Putting the Class Definitions into Separate Files

In the previous concept, you saw how to create a `Car` classs and use a constructor. At the end of that concept, your code looked like this:

```
#include <iostream>
#include <string>
using std::string;
using std::cout;

class Car {
  public:
    void PrintCarData()
    {
        cout << "The distance that the " << color << " car " << number << " has traveled is: " << distance << "\n";
    }

    void IncrementDistance()
    {
        distance++;
    }

    // Adding a constructor here:
    Car(string c, int n)
    {
        // Setting the class attributes with
        // The values passed into the constructor.
        color = c;
        number = n;
    }

    string color;
    int distance = 0;
    int number;
};

int main()
{
    // Create class instances for each car.
    Car car_1 = Car("green", 1);
    Car car_2 = Car("red", 2);
    Car car_3 = Car("blue", 3);

    // Increment car_1's position by 1.
    car_1.IncrementDistance();

    // Print out the position and color of each car.
    car_1.PrintCarData();
    car_2.PrintCarData();
    car_3.PrintCarData();

}
```

If you were planning to build a larger program, at this point it might be good to put your class definition and function declarations into a separate file. Just as when we discussed header files before, putting the class definition into a separate header helps to organize your code, and prevents problems with trying to use class objects before the class is defined.

There are two things to note in the code below.

1. When the class methods are defined outside the class, the *scope resolution operator* `::` must be used to indicate which class the method belongs to. For example, in the definition of the `PrintCarData` method you see:

   ```
   void Car::PrintCarData()
   ```

   This prevents any compiler issues if there are are two classes with methods that have the same name.

2. We have changed how the constructor initializes the variables. Instead of the previous constructor:

   ```
   Car(string c, int n) {
      color = c;
      number = n;
   }
   ```

   the constructor now uses an *initializer list* (https://en.cppreference.com/w/cpp/language/initializer_list):

   ```
   Car(string c, int n) : color(c), number(n) {}
   ```

   Here, the class members are initialized before the body of the constructor (which is now empty). Initializer lists are a quick way to initialize many class attributes in the constructor. Additionally, the compiler treats attributes initialized in the list slightly differently than if they are initialized in the constructor body. For reasons beyond the scope of this course, if a class attribute is a reference, it must be initialized using an initializer list.
3. Variables that don't need to be visible outside of the class are set as `private`. This means that they can not be accessed outside of the class, which prevents them from being accidentally changed (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rc-private).

Check out the cells below to see this code in practice. In this code, we have separated the class into declarations and definitions, with declarations being in the `.h` file and definitions being in `.cpp`. Note that only the `.h` file needs to be included in any other file where the definitions are used.

car.h and car.cpp

In [ ]:

```
#ifndef CAR_H
#define CAR_H

#include <string>
using std::string;
using std::cout;

class Car {
  public:
    void PrintCarData();
    void IncrementDistance();

    // Using a constructor list in the constructor:
    Car(string c, int n) : color(c), number(n) {}

  // The variables do not need to be accessed outside of
  // functions from this class, so we can set them to private.
  private:
    string color;
    int distance = 0;
    int number;
};

#endif
```

In [ ]:

```
#include <iostream>
#include "car.h"

// Method definitions for the Car class.
void Car::PrintCarData()
{
    cout << "The distance that the " << color << " car " << number << " has traveled is: " << distance << "\n";
}

void Car::IncrementDistance()
{
    distance++;
}
```

car_main.cpp

In [ ]:

```
#include <iostream>
#include <string>
#include "car.h"
using std::string;
using std::cout;

int main()
{
    // Create class instances for each car.
    Car car_1 = Car("green", 1);
    Car car_2 = Car("red", 2);
    Car car_3 = Car("blue", 3);

    // Increment car_1's position by 1.
    car_1.IncrementDistance();

    // Print out the position and color of each car.
    car_1.PrintCarData();
    car_2.PrintCarData();
    car_3.PrintCarData();

}
```

Compile & Execute Explain

Loading terminal (id_2ry0tt5), please wait...

Scaling Up

In this concept and the previous one, you took code without classes and converted it into an object-oriented format.

In case you aren't convinced that organizing the code using OOP saved you some trouble, the next cell redefines `main.cpp` to generate 100 cars with different colors, move each, and print data about each. This would have been extremely difficult to do if you had to manually create new variables for each car!

There is a lot going on in the code to unpack, including the `new` keyword and the `->` operator. The arrow operator `->` is used to simultaneously

- dereference a pointer to an object and
- access an attribute or method.

For example, in the code below, `cp` is a pointer to a Car object, and the following two are equivalent:

```
// Simultaneously dereference the pointer and
// access IncrementDistance().
cp->IncrementDistance();

// Dereference the pointer using *, then
// access IncrementDistance() with traditional
// dot notation.
(*cp).IncrementDistance();
```

The `new` operator allocates memory on the "heap" for a new Car. In general, this memory must be manually managed (deallocated) to avoid memory leaks in your program. Memory management is the primary focus of one of the later courses in this Nanodegree program, so we won't go into greater depth about the difference between stack and heap in this lesson.

Click on the explanation button for a discussion of the code.

Note: This `main.cpp` uses the class files defined above, so be sure you have run the previous example before running this one.

In [ ]:

```
#include <iostream>
#include <string>
#include <vector>
#include "car.h"
using std::string;
using std::cout;
using std::vector;

int main() {
    // Create an empty vector of pointers to Cars
    // and a null pointer to a car.
    vector<Car*> car_vect;
    Car* cp = nullptr;

    // The vector of colors for the cars:
    vector<string> colors {"red", "blue", "green"};

    // Create 100 cars with different colors and
    // push pointers to each of those cars into the vector.
    for (int i=0; i < 100; i++) {;
        cp = new Car(colors[i%3], i+1);
        car_vect.push_back(cp);
    }

    // Move each car forward by 1.
    for (Car* cp: car_vect) {
        cp->IncrementDistance();
    }

    // Print data about each car.
    for (Car* cp: car_vect) {
        cp->PrintCarData();
    }
}
```

Compile & Execute Explain

Loading terminal (id_y7nkua3), please wait...