Classes: Overloading, Copy Constructors, and More

Thomas Merkh
tmerkh55@gmail.com

Section 2, Week 7

May 14, 2020
Our Example Project: MathVector

Operator Overloading

The Copy Constructor
Last Time

Last time we introduced classes. As a refresher:

- The fundamental syntax associated with classes was covered
- The public/private qualifiers indicate whether or not a member is accessible from outside of the class
- All classes need at least one constructor
Last time we introduced classes. As a refresher:

- The fundamental syntax associated with classes was covered
- The `public/private` qualifiers indicate whether or not a member is accessible from outside of the class
- All classes need at least one constructor

Today, we will cover operator overloading and copy constructors by developing a `MathVector` class, which will function similar to vectors in mathematics.
1. Our Example Project: MathVector

2. Operator Overloading

3. The Copy Constructor
```cpp
#include <vector>
#include <iostream>
#include <initializer_list>
using namespace std;

class MathVector
{
    public:
        // Null constructor
        MathVector() : vData() {
            vData = {};
        }

        // Copy constructor
        MathVector(const MathVector& V) : vData(V.vData) {
            vData = V.vData;
        }

        // Values list initializer
        MathVector(initializer_list<double> init) : vData(init) {
            vData = init;
        }
};
```
/ Destructor
virtual ~MathVector() {};

MathVector& operator=(const MathVector& V)
{
    vData = V.vData;
    return *this;
}

// Incremental vector addition : (*this) = (*this) + V
void operator += (const MathVector& V)
{
    for (size_t i = 0; i < this->size(); ++i)
    {
        this->vData[i] += V.vData[i];
    }
}

private:
    vector<double> vData;
};
This example class has many aspects which you do not need to worry about:

- The `:`'s after the constructors and the arguments that follow. These are called initialization lists, and are out of the scope of this class.
- The `virtual` qualifier before the destructor. This prevents errors from occurring when dealing with class inheritance.
- `this->` operations, or `(this*)` operations. These deal with pointers (PIC 10B).

What *are* we going to worry about?
Our Example Project: MathVector

Operator Overloading

The Copy Constructor
Operator overloading is when we give additional meanings to operators. For example:

- The operator + has an intuitive meaning for numbers, e.g. \( a = 6 + 4 \); means that \( a = 10 \).
- The operator + also has an additional (and natural) meaning for strings. \( \text{string name} = \"To\" + \"M\"; \) assigns \text{name} equal to \"ToM\".

- How can we give + a meaning for a class that we create?

Answer: Operator overloading.
Here is an example of **overloading the `+=` operator**:

```cpp
void operator += (const MathVector& V) {
    for (size_t i = 0; i < this->size(); ++i) {
        this->vData[i] += V.vData[i];
    }
}
```

Now, this looks complicated. So let's break it down: `void operator += (const MathVector& V)` is declaring that the `+=` operator for this class will operate on another `MathVector` type (adding one to the other) `is passing the object by reference to prevent copying the object`.
Here is an example of **overloading the `+=` operator**:

```cpp
void operator += (const MathVector& V) {
    for (size_t i = 0; i < this->size(); ++i) {
        this->vData[i] += V.vData[i];
    }
}
```

Now, this *looks* complicated. So let’s break it down:

- `void operator += (const MathVector& V)` is declaring that the `+=` operator for this class will operate on another `MathVector` type object (adding one to the other)
- `(const MathVector& V)` is passing the object by reference to prevent copying the object
```cpp
void operator += (const MathVector& V)
{
    for (size_t i = 0; i < this->size(); ++i)
    {
        this->vData[i] += V.vData[i];
    }
}
```

- `for(size_t i = 0; i < this->size(); ++i)` is an ordinary for loop, where `size_t` could equivalently be written as an `int`
- `this->size()` retrieves the size of the object we are incrementing
Then, we could run a code like:

```cpp
int main()
{
    MathVector A = {1, 2, 3};
    MathVector B = {4, 5, 6};
    B += A;
    cout << B << endl;
    // Outputs: {5, 7, 9}
}
```
Then, we could run a code like:

```cpp
int main() {
    MathVector A = {1, 2, 3};
    MathVector B = {4, 5, 6};
    B += A;
    cout << B << endl;
    // Outputs: {5, 7, 9}
}
```

Again, most of the complicated details have been passed over or hidden, but this demonstrates that operators like `+=` can be given new meanings which are class-specific.
There are many uses for operator overloading:

- Some classes are complicated to print out. It's useful to overload « so one can simply cout the object
- Often there is a natural interpretation of what + would mean between classes. With overloading the plus operator, the code becomes significantly easier to read and understand
- It is easier to remember $A + B$ compared to giving this operation a “real name” which is easily forgotten. Think: “Was it $A\text{.add}(B)$, $A\text{.Add}(B)$, $\text{add}(A,B)$, $A\text{.plus}(B)$..?”
Operator overloading is also an extremely useful concept to understand in general. Python uses operator overloading so frequently and fluidly that it actually seems “easy” to learners because it's so natural.

- An example is that for basically any object in Python, (print(object)) will almost always print the object in an understandable form to the console.
- Technically, Python does this through “special methods”, which look like __add__ or __sub__ and are essentially operator overloading.
Syntax for Overloading

For completeness, I’ve included common syntax for operator overloading for the case of the MathVector class:

MathVector& operator=(const MathVector& V)
MathVector operator+(const MathVector& V)
MathVector operator-(const MathVector& V)
void operator += (const MathVector& V)
void operator -= (const MathVector& V)
void operator *= (double alpha)
void operator /= (double alpha)
MathVector operator*(double alpha)
MathVector operator/(double alpha)

This syntax will have to be updated for whatever assignment/project you are working on.
1. Our Example Project: MathVector

2. Operator Overloading

3. The Copy Constructor
In order to instantiate an instance of a class, or create an object with the class type, we need to call the class’s constructor.

In the main code block, this looks like `MyClass objname;`.

In the class code block, this function may look like:

```cpp
MyClass()
{
    //initialization of data members
}
```

More than one constructor can be defined, as long as the input list varies between definitions.
Consider the code:

```cpp
class Person {
public:
    Person() {
        fname = "No";
        lname = "Name";
        age = 0;
        bornday = 0;
    }

    Person(string f, string l, int a, int b) {
        fname = f;
        lname = l;
        age = a;
        bornday = b;
    }

private:
    string fname;
    string lname;
    unsigned int age;
    int bornday;
};
```
Here there were two constructors defined, the *default* constructor which initializes all of the data members to generic values, and an additional constructor which initializes the data members to whatever values are fed into the constructor call. What do you think the following calls will do?

- Person one;
- Person two("Betty", "Anderson", 30, 5);
Default Constructor

Here there were two constructors defined, the *default* constructor which initializes all of the data members to generic values, and an additional constructor which initializes the data members to whatever values are fed into the constructor call. What do you think the following calls will do?

- Person one;
- Person two("Betty", "Anderson", 30, 5);

You may have guessed that the object *two* will have *fname* “Betty” associated with it, whereas *one* will have *fname* “No” associated with it.
The copy constructor is a particular useful idea which will allow for new objects to be constructed as copies of already existing objects. For example, if one is a Person, then can we create a copy of one called three?
The copy constructor is a particular useful idea which will allow for new objects to be constructed as copies of already existing objects. For example, if one is a Person, then can we create a copy of one called three?

Not yet, but with a copy constructor we can. Here, we want to take in a class object as an input to the constructor. This is demonstrated on the following slide.
class Person {
public:
    Person() {
        fname = "No";
        lname = "Name";
        age = 0;
        bornday = 0;
    }
    Person(Person& P) {
        fname = P.fname;
        lname = P.lname;
        age = P.age;
        bornday = P.bornday;
    }

private:
    string fname;
    string lname;
    unsigned int age;
    int bornday;
};
Copy Constructor

Since we know that this constructor is supposed to copy another Person object, the action of the following code should be pretty predictable:

```cpp
int main() {
    Person mich(“Tom”, “Merkh”, 100, 273);
    Person myself(mich);
    return 0;
}
```