Learning Objectives - Getters and Setters

- **Define the terms getter, setter, and data validation**
- **Demonstrate how to access private attributes with getters**
- **Demonstrate how to change private attributes with setters**
- **Demonstrate validating data with setters**

Getters

Getters

While all attributes are private, that does not mean that the user will not need to access the values of these attributes. A public function can be used to access a private attribute. This type of function has a special name, a **getter** (also called an accessor). The whole purpose of a getter is to return an attribute. These functions get their name because they always start with `Get` followed by the attribute name.

```
//add class definitions below this line

class Phone {
  public:
    Phone(string mo, int s, int me) {
      model = mo;
      storage = s;
      megapixels = me;
    }

    string GetModel() {
      return model;
    }

  private:
    string model;
    int storage;
    int megapixels;
};

//add class definitions above this line
```

The function `GetModel` is an example of a getter. Getters are very simple, straightforward functions that do only one thing — return a private attribute. Getters can be treated just as you would treat the attribute (except for changing its value). To test a getter, you can call in `main` to see if it returns what is expected.

```
//add code below this line

Phone my_phone("iPhone", 256, 12);
cout << my_phone.GetModel() << endl;

//add code above this line
```

challenge

Try this variation:

Create getters for the `storage` and `megapixels` attributes. Then call them within `main`.

▼ **Possible Solution**

```
//class definitions
public:
  int GetStorage() {
    return storage;
  }

  int GetMegapixels() {
    return megapixels;
  }
```

```
//main function
cout << my_phone.GetStorage() << endl;
cout << my_phone.GetMegapixels() << endl;
```

Benefits of Getters

Using a getter is the same thing as accessing a public attribute. Why not make the attribute public? That would mean writing less code. Is that not a good thing? A public attribute makes no distinction between accessing its value and changing its value. If you can access it, you can change it (or vice versa). Using a getter with a private attribute makes this distinction clear; you can access the value, but you cannot change it.

```
//add code below this line

Phone my_phone("iPhone", 256, 12);
cout << my_phone.GetModel() << endl;
my_phone.model = "Pixel 5";

//add code above this line
```

The code above generates an error because an instance cannot alter a private attribute. Using a getter allows limited access to an attribute, which is preferable to the full access a public access modifier allows.

Setters

Setters

Setters are the compliment to getters in that they allow you to set the value of a private attribute. Setter functions are also called mutators. Use the `Phone` class from the previous page.

```
//add class definitions below this line

class Phone {
  public:
    Phone(string mo, int s, int me) {
      model = mo;
      storage = s;
      megapixels = me;
    }

    string GetModel() {
      return model;
    }

    int GetStorage() {
      return storage;
    }

    int GetMegapixels() {
      return megapixels;
    }

  private:
    string model;
    int storage;
    int megapixels;
};

//add class definitions above this line
```

Add the `SetModel` method to the `Phone` class. As this is a setter method, start the name with `Set` followed by the name of the attribute. Setters do not return anything. Finally, setters have a parameter — the new value for the attribute.

```
public:
  // Setter
  void SetModel(string new_model) {
    model = new_model;
  }
```

Now that you are implementing both getters and setters, you should now be able to access and modify private attributes.

```
//add code below this line

Phone my_phone("iPhone", 256, 12);
cout << my_phone.GetModel() << endl;
my_phone.SetModel("XR");
cout << my_phone.GetModel() << endl;

//add code above this line
```

challenge

Try this variation:

Create setters for the `storage` and `megapixels` attributes. Then call them within `main`.

▼ **Possible Solution**

```
//class definitions
public:
  void SetStorage(int new_storage) {
    storage = new_storage;
  }

  void SetMegapixels(int new_megapixels) {
    megapixels = new_megapixels;
  }
```

```
//main function
my_phone.SetStorage(128);
cout << my_phone.GetStorage() << endl;
my_phone.SetMegapixels(6);
cout << my_phone.GetMegapixels() << endl;
```

Comparing Getters and Setters

Getters and setters have a lot in common. Their names are similar, they have the same number of lines of code, etc. However, getters and setters also differ in a few important ways. The table below highlights these similarities and differences.

Category	Getters	Setters
Has `public` keyword	X	X
Has `private` keyword	-	-
Has `return` statement	X	-
Has `void` type	-	X
Performs only 1 task	X	X
Has parameter	-	X

Data Validation

C++ already has a type system which will flag errors when a boolean value is passed to a function that takes an integer. However, just because a function takes an integer does not mean that all integers are valid for that function. Data validation is the process of asking if this data is appropriate for its intended use. Take a look at the `Person` class.

```
//add class definitions below this line

class Person {
  public :
    Person(string n, int a) {
      name = n;
      age = a;
    }

    string GetName() {
      return name;
    }

     void SetName(string new_name) {
      name = new_name;
    }

     int GetAge() {
      return age;
    }

     void SetAge(int new_age) {
      age = new_age;
    }

  private:
    string name;
    int age;
};

//add class definitions above this line
```

The `SetAge` function will assign any value to the attribute `age` as long as the value is an integer. There are some integers which make no sense when thought of as an age attribute. The code sample below sets the age of

`my_person` to `-100`. `-100` is a valid integer, but it is not a valid age. This is why data validation is important. C++'s compiler is not sufficient to catch all errors.

```
//add code below this line


Person my_person("Calvin", 6);
cout << my_person.GetName() << " is " << my_person.GetAge() <<
      " years old." << endl;
my_person.SetAge(-100);
cout << my_person.GetName() << " is " << my_person.GetAge() <<
      " years old." << endl;


//add code above this line
```

Another benefit of using setters is that data validation can take place before the new value is assigned to the attribute. Modify `SetAge` so that it will only update the `age` attribute if the new value is greater than or equal to 0. This way you can ensure that the `Person` object always has a valid age.

```
void SetAge(int new_age) {
  if (new_age >= 0) {
    age = new_age;
  }
}
```

challenge

Try these variations:

- Change the data validation for the `SetAge` function so that values over `200` or less than `0` are not valid.

▼ **Possible Solution**

Here is one possible solution.

```
void SetAge(int new_age) {
  if (new_age >= 0 && new_age <= 200) {
    age = new_age;
  }
}
```

- Add data validation to the `SetName` function so that all strings with one or more characters are valid.

▼ **Possible Solution**

Here is one possible solution.

```
void SetName(string new_name) {
  if (new_name != "") {
    name = new_name;
  }
}
```