14.6 — Access functions

In previous lesson 14.5 -- Public and private members and access specifiers, we discussed the public and private access levels. As a reminder, classes typically make their data members private, and private members can not be directly accessed by the public.

Consider the following `Date` class:

```
#include <iostream>

class Date
{
private:
    int m_year{ 2020 };
    int m_month{ 10 };
    int m_day{ 14 };

public:
    void print() const
    {
        std::cout << m_year << '/' << m_month << '/' << m_day << '\n';
    }
};

int main()
{
    Date d{};  // create a Date object
    d.print(); // print the date

    return 0;
}
```

While this class provides a `print()` member function to print the entire date, this may not be sufficient for what the user wants to do. For example, what if the user of a `Date` object wanted to get the year? Or to change the year to a different value? They would be unable to do so, as `m_year` is private (and thus can't be directly accessed by the public).

For some classes, it can be appropriate (in the context of what the class does) for us to be able to get or set the value of a private member variable.

Access functions

An **access function** is a trivial public member function whose job is to retrieve or change the value of a private member variable.

Access functions come in two flavors: getters and setters. **Getters** (also sometimes called **accessors**) are public member functions that return the value of a private member variable. **Setters** (also sometimes called **mutators**) are public member functions that set the value of a private member variable.

Getters are usually made const, so they can be called on both const and non-const objects. Setters should be non-const, so they can modify the data members.

For illustrative purposes, let's update our `Date` class to have a full set of getters and setters:

```
#include <iostream>

class Date
{
private:
    int m_year { 2020 };
    int m_month { 10 };
    int m_day { 14 };

public:
    void print()
    {
        std::cout << m_year << '/' << m_month << '/' << m_day << '\n';
    }

    int getYear() const { return m_year; }        // getter for year
    void setYear(int year) { m_year = year; }     // setter for year

    int getMonth() const  { return m_month; }     // getter for month
    void setMonth(int month) { m_month = month; } // setter for month

    int getDay() const { return m_day; }          // getter for day
    void setDay(int day) { m_day = day; }         // setter for day
};

int main()
{
    Date d{};
    d.setYear(2021);
    std::cout << "The year is: " << d.getYear() << '\n';

    return 0;
}
```

This prints:

```
The year is: 2021
```

Access function naming

There is no common convention for naming access functions. However, there are a few naming conventions that are more popular than others.

Prefixed with “get” and “set”:

```
int getDay() const { return m_day; }  // getter
void setDay(int day) { m_day = day; } // setter
```

The advantage of using “get” and “set” prefixes is that it makes it clear that these are access functions (and should be inexpensive to call).

No prefix:

```
int day() const { return m_day; }  // getter
void day(int day) { m_day = day; } // setter
```

This style is more concise, and uses the same name for both the getter and setter (relying on function overloading to differentiate the two). The C++ standard library uses this convention.

The downside of the no-prefix convention is that it is not particularly obvious that this is setting the value of the day member:

```
d.day(5); // does this look like it's setting the day member to 5?
```

Key insight

One of the best reasons to prefix private data members with “m_” is to avoid having data members and getters with the same name (something C++ doesn’t support, although other languages like Java do).

“set” prefix only:

```
int day() const { return m_day; }     // getter
void setDay(int day) { m_day = day; } // setter
```

Which of the above you choose is a matter of personal preference. However, we highly recommend using the “set” prefix for setters. Getters can use either a “get” prefix or no prefix.

Tip

Use a “set” prefix on your setters to make it more obvious that they are changing the state of the object.

Getters should return by value or by const lvalue reference

Getters should provide “read-only” access to data. Therefore, the best practice is that they should return by either value (if making a copy of the member is inexpensive) or by const lvalue reference (if making a copy of the member is expensive).

Because returning data members by reference is a non-trivial topic, we’ll cover that topic in more detail in lesson 14.7 -- Member functions returning references to data members.

Access functions concerns

There is a fair bit of discussion around cases in which access functions should be used or avoided. Many developers would argue that use of access functions violates good class design (a topic that could easily fill an entire book).

For now, we’ll recommend a pragmatic approach. As you create your classes, consider the following:

- If your class has no invariants and requires a lot of access functions, consider using a struct (whose data members are public) and providing direct access to members instead.
- Prefer implementing behaviors or actions instead of access functions. For example, instead of a `setAlive(bool)` setter, implement a `kill()` and a `revive()` function.
- Only provide access functions in cases where the public would reasonably need to get or set the value of an individual member.

Why make data private if we’re going to provide a public access function to it?

So glad you asked. We’ll answer this question in upcoming lesson 14.8 -- The benefits of data hiding (encapsulation).