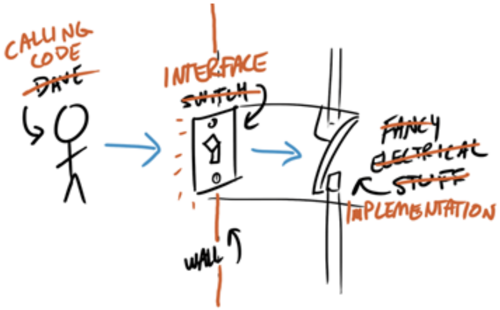
The Wall of Abstraction

A class is an interface paired with an implementation, similar to the `Time` structure you created in the lesson on Information Hiding. This is called the **wall of abstraction**, illustrated by the comic below.



The **public interface** specifies how clients interact with objects, and the **private implementation** specifies how the functions in the public interface are implemented.

The `Time` struct (even when paired with an interface, so the data is hidden), still allows users to **directly manipulate** its data members. Classes take a different approach; with classes, the data inside your objects will **only be accessible by the member functions**, **forcing** the client to access and modify data in a safe way.

Class Definition Syntax

▶ *The Time Class Definition*

To create a class definition for `Time`, similar to the structure from last week, follow these rules:

1. Instead of `struct`, use the `class` keyword. There is no `public` in front of this as with Java.
2. The `public` keyword, followed by a colon, indicates the start of the **public interface**. Here we **prototype** the member functions it contains.
3. The member functions `hours()`, `minutes()`, `sum()`, `difference()` and `write()`, all access the `hours` and `minutes` data members **without changing them**. When this is the case, add the `const` keyword **after the argument list**. We say these functions are **accessors**.
4. The `read()` member function does **modify** the `Time` object. This is called a **mutator**.
5. The class definition **ends with a semicolon**, just like a structure. This is not optional.

Data Members

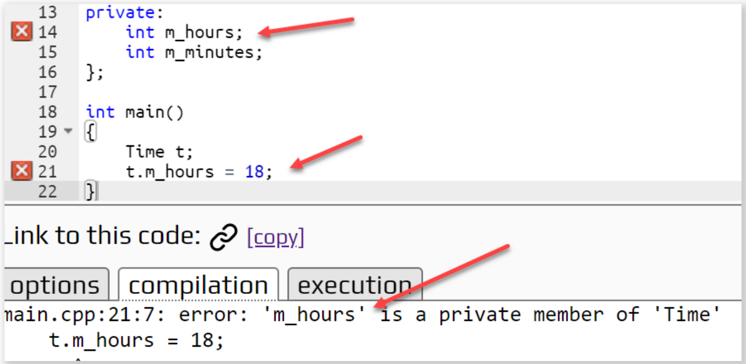
Most of the implementation will appear inside a `.cpp` file. **Defining the data members** which store **object state**, is **written inside the header file instead**. A common practice is to use a special indicator like `m_` to show that it is a data member.

The `Time` `struct` used two individual data members: one for `hours` and one for `minutes`. This is fine; it allows you to store all of information needed. By adding `private`, you can **prevent clients of `Time` from accessing the data members directly.**

Public and Private

So, what do `public` and `private` mean in C++? If a member of a class is `public`, then **any part of your code** can access and manipulate it directly. If you have a **`public`** member function, any code can **call it using an object of that type**. If a data member is marked **`private`**, then **only** member functions of the class can access it.

The **`public`** and **`private`** keywords are the C++ mechanism for **defining interfaces and enforcing encapsulation**. Once you add **`private`**, the compiler enforces the **appropriate encapsulation**.



By prohibiting clients from directly accessing **`private`** data, the implementation can assume that all access to that data goes through the **`public`** interface (unlike the **`Time struct`** of last week, where clients **should use the member functions**, but **were not prohibited** from directly accessing the data members **`m_hours`** and **`m_minutes`**.)

> *Actually, the only **real** difference between **`class`** and **`struct`** in C++ is that with a **`struct`**, the members are **`public`** by default; with a **`class`** they are **`private`**. By convention, we will use **`struct`** for **POD** (plain-old-data) data types, and **`class`** for encapsulated types.*

The Implicit Parameter

Consider the implementation of the `hours()` member function of the `Time` class shown here:

```
int Time::hours() const
{
    return m_hours;
}

int main()
{
    Time t;                             // a Time object
    cout << t.hours() << endl;          // value of t::m_hours
}
```

The **`hours()`** member function **does not** contain a local variable named **`m_hours`**. But, the function still compiles and runs correctly. Why?

In a **member function**, you may **directly access and manipulate** any or all of the class's **data members** by referring to them by name. You don't need to indicate that **`m_hours`** is a data member, nor do you specify **which** `Time` object it belongs to.

C++ assumes that all data members **are the data members of the receiver object**, and so the line **`return m_hours`** means *"return the value of the **`m_hours`** data member of the object on which this function was invoked."* In such a case, **the receiver object is known as the implicit parameter**, passed to every member function.

The Pointer this

Behind the scenes, the implicit parameter is a pointer to the calling object. Every member function has an implicit parameter. Thus the effective signature for the `hours()` function is as if you had declared it like this:

```
int hours(const Time* const this);
```

The keyword `this` is the **name which is automatically supplied** for the implicit parameter. The `const` following the `Time*` means that the value inside the pointer can never be changed; it always points to the block of data containing the object's data members. The `const` following the member function header means that the implicit parameter is a pointer to a `const Time` object.

If you wish, you can **explicitly** use the pointer when calling other member functions, or accessing data members:

```
int Time::hours() const
{
    return this->m_hours;
};
```

Initializing this

When you **call a member function** like this:

```
Time t;     // a Time object
cout << t.hours() << endl;           // value of t::m_hours
```

That call is **implicitly translated** into code that acts as if you had written:

```
Time t;     // a Time object
cout << hours(&t) << endl;           // value of t::m_hours
```

Because of this call, the `this` pointer is initialized to the **address of the calling object**.

The sum Member Function

Let's examine the behavior of `this` and `const` a little more closely, by considering the **`sum()`** member function from **`Time`**:

```
class Time
{
public:
    Time sum(const Time& rhs) const;
    . . .
};
```

When you add two **`Time`** objects (**`a`** + **`b`**) together like this:

```
Time after = a.sum(b);
```

The **caller** (the implicit parameter) is the left-hand-side of the expression **`a`** + **`b`**. Thus, the effective implicit prototype for the function is similar to this:

```
Time sum(const Time* lhs, const Time& rhs);
```

In the implementation, however, instead of the **explicit `lhs` parameter** shown here, you'd use the keyword **`this`** to access the data members.

```
Time Time::sum(const Time& rhs) const
{
    auto tMinutes = this->m_hours * 60 + this->m_minutes;
    auto dMinutes = rhs.m_hours * 60 + rhs.m_minutes;
    . . .
}
```

If you leave off the keyword **`this`**, it is assumed. Notice that when you implement a **`const`** member function, you **repeat** the word **`const`** in the implementation.

Setters

In Java, many classes have member functions that start with `set`. These are called **mutators**, since they **change the state** of the object. Mutators should **validate data** written to the object to **enforce the class invariants**.

With properly written mutators, the errors described in earlier lessons **cannot occur**. Consider your `Time` class. If you were to add `setHours()` and `setMinutes()` members to the class, you would have to enforce these restrictions:

- `m_hours` must be between `0` and `23` inclusive.
- `m_minutes` must be between `0` and `59` inclusive.

Unlike the `read()` member function, where you could put the stream into a failed state, if these conditions were not met, in a mutator you need to `throw` an exception like this:

```
void Time::setHours(int h)
{
    if (h < 0 || h > 23) throw out_of_range("...");
    m_hours = h;
}
```

Getter & Setter Patterns

The pattern of pairing a "getter" along with a "setter" function is common, and you will see it in any major C++ project you work on. Unlike Java, the actual `get*` and `set*` name pattern is not as common. Instead, what programmers often do is write **a pair of overloaded functions**.

Instead of the name `getHours()` or `setHours()`, use the name `hours()` for both of them.

- The accessor is `const` and returns a value.
- The non-`const` mutator returns a reference, which can be assigned to.

```
Time Time::hours() const { return m_hours; }  // accessor
Time& Time::hours()                           // mutator
{
    if (h < 0 || h > 23) throw out_of_range("...");
    return m_hours;
}
```

In general, it is safer to allow clients to **read the values** of the data members than it is allow them to **change** those values. As a result, "setters" are far less common than "getters" in class design. Classes with no mutators at all, are called **immutable classes**.

Constructors

Initializing object data is the responsibility of the constructor, which always has **the same name as the class** and which **never has a return type**.

```
class Time
{
public:
    Time();     // default constructor
    ...
};
```

A constructor is a member function which **initializes an object into a well-formed state** before clients start manipulating it. When C++ creates an object from a class:

1. It **allocates a block of memory** large enough to store the data elements.
2. It passes **the address of that block** of memory to the constructor function. The address is the this pointer inside the constructor function.

The constructor **is called automatically** whenever an object is created. If you have a class that defines a constructor, that constructor is **guaranteed to execute** whenever you create an object of the class type.

Default Constructors

The **default constructor** is the constructor which takes **no arguments** and which should initialize **all of its data members** to an appropriate **default** value. Alternatively, since C++11, you may **provide an initial value** when defining the data members, just as in Java.

If you do not provide a constructor, the compiler will "write" one for you. This is called the **synthesized default constructor**. If you use in-definition initializers, then this is perfect.

Member Initialization

In C++, all constructors must initialize all primitive types. A C++ constructor does not need to initialize any object members (like `string` or `vector`).

This is **exactly the opposite from Java**, where you must initialize all of the object instance variables, or they are set to `null` (an invalid object). In Java, all primitive instance variables are automatically initialized to `0` like this:

```
public class Point
{
    private String name;
    int x, y;
    public Point() {}
}

Point p = new Point(); // x,y->0, name is null (invalid)
```

In C++, if you fail to initialize a primitive data member, then it assumes **whatever random value** was in memory; if you don't initialize an object, such as `string` or `vector`, its default constructor will **automatically** run, and it is still a valid object.

```
class Point
{
public:
    Point() {}
private:
    string name;
    int x, y;
};

Point p; // x,y->random, name is valid empty string
```

Of course, if you provide in-definition initializers for your primitive data members, they **will** automatically be initialized, even if your construct does not explicitly initialize them.

Working Constructor

With the `Time` class we might like to have another, overloaded constructor which takes hours and minutes. This is generally known as the **working constructor**. Here is the public interface of `Time` with both of these constructors.

```
class Time
{
public:
    Time();                    // default
    Time(int h, int m);        // working
    . . .
};
```

Unfortunately, if you have **any** explicit constructors, the synthesized one **is deleted**, so you have to add an **explicit default constructor** as done here. In C++11, however, you can just add the phrase `=default;` to the end of the prototype in the class header, and the compiler will **retain** the synthesized constructor that it normally writes.

The **implementation** of the constructors goes into the `.cpp` file along with the other member functions. The job of the constructor is to **initialize the data members**, so in the `Time` class, you might have code that looks something like this.

```
Time::Time() { m_hours = m_minutes = 0; }
Time::Time(int hours, int minutes)
{
    m_hours = h;
    m_minutes = m;
}
```

Assignment vs. Initialization

Before we talk about constructors, look at these two statements:

```
string a = "Bob", b;       // initialization
b = "Bill";                // assignment
```

1. Two `string` objects are created and initialized on line one; `a` is initialized using the C-String `"Bob"`, and `b` is initialized to the empty `string` by running the default constructor.
2. The `string` object `b` is destroyed (its destructor is run), a new `string` object is initialized with `"Bill"`, and that new `string` object replaces the `string` object originally held by `b`.

The variable `b` is first initialized, then destroyed, then assigned. **This is inefficient**.

Assignment in a Constructor

The body of the constructor is executed **after** the data members have been initialized. You may use **assignment** to place a new value into these data members. For primitive types, the cost of doing this is negligible, but for object types, such assignments mean that **data members are constructed twice**—once at initialization and once at assignement. Here's an example. (The implementation is inline to shorten the code.)

```
class Person
{
public:
  Person(const string& name) { m_name = name; }
private:
    string m_name;
};
```

When you write `Person p("Fred")`, the `m_name` data member first calls the default constructor to create an empty `string` object. Then, in the body of the constructor, the default-constructed `string` is destroyed when assigning `name` to `m_name`. **This is inefficient**, and you want to avoid it.

The Initializer List

We can instruct the compiler to initialize the individual data members before the body of the constructor is entered instead. This is called the **initializer list**:

- It follows the parameter list and is preceded by a colon (`:`)
- It is followed by a list of member names and their initializers.
- Initialization occurs **in the order the members are declared in the class**.

In C++98 the initializers are placed in parentheses; in C++11 use either parentheses or braces. You cannot use the assignment operator. Here is the same class using the initializer list. In this case, the `name` data member is **only constructed once**:

```
class Person
{
public:
  Person(const string& name) : m_name(name) { } // empty body
private:
    string m_name;
};
```

The Working Constructor

As you saw in the last lesson, working constructor is the short-hand description of a constructor that takes as many user-supplied arguments as possible. In the `Time` class the working constructor looks like this:

```
Time(int hours, int minutes);
```

In the `.cpp` file, you might have code that looks something like this, using the initializer list, also from the last lesson:

```
Time::Time(int hours, int minutes)
  : m_hours(hours), m_minutes(minutes)
{
    // validate the constructor arguments
    assert(hours >= 0 && hours < 24);
    assert(minutes >= 0 && minutes < 60);
}
```

It is important that your constructor intitializes your object so that it is **in a valid state**. In the example shown here, we've used `assert()` on the assumption that it is a programming error if an invalid `Time` is constructed. If, however, you were constructing `Time` objects using external data, it is possible you would want to `throw` and `catch` an exception instead.

Conversion Constructors

A conversion constructor is a constructor (usually 1-argument) that implicitly converts between one type and another. Here's an overloaded conversion constructor that converts between fractional hours and hours and minutes:

```
Time(double hours);
```

The implementation of the constructor (converting first to seconds, then extracting the hours and minutes) could look like this:

```
Time::Time(double hours)
{
    assert(hours >= 0 && hours < 24);
    const int kSecondsPerHour = 3600;
    const int kHoursPerMinute = 60;
    auto time = static_cast<int>(hours * kSecondsPerHour);
    m_hours = time / kSecondsPerHour;
    m_minutes = time % kSecondsPerHour / kHoursPerMinute;
}
```

Implicit vs. Explicit

Conversion constructors can be implicit (which is the default), or explicit. The implicit conversion constructor is called any time the compiler needs a `Time` object, but finds a `double` that it can convert. Consider this fragment of code:

```
1 Time bed_time(23, 30);      // 11:30
2 bed_time = 5.2;             // WHAAAAT?
```

You would **expect** that line 2 would be a syntax error, but, surprisingly, it is not. Instead, **the conversion constructor is silently (implicitly) called**, and `bed_time` is changed to **5:12 am**. Probably not what you expected.

You can add `explicit` as a modifier to the prototype to prevent this:

```
explicit Time(double hours);  // 7.51 -> 7:35
```

The keyword `explicit` **only** goes in the class definition. It **is not repeated** in the `.cpp` file. Sometimes, as you'll see when you cover symmetric overloaded operators in CS 250, you'll **want** to allow implicit conversion. For instance the `string(const char*)` constructor is **not explicit**. Most of the time, however, `explicit` is preferred.