# Time as a Structure

**At the beginning of the semester (H01), you wrote a program to add and subtract time.** This was harder than expected, because you didn't have a Time type; you did everything with integers. Let's rectify that now by creating a `Time` structure, with hours and minutes. Assume a 24-hour clock, so you don't need an indicator for AM/PM.

```
struct Time
{
    int hours;
    int minutes;
};
```

Now, you can create a `Time` **object** that bundles that data:

```
int main()
{
    Time lunch = = {11, 15};
    cout << lunch.hours << ":" << lunch.minutes << endl;

    return 0;
}
```

## Type Invariants

**The definition of `Time` is straightforward, but, it will cause problems. There** are **certain restrictions** on what values members of a `Time` object may and may not have. Given our specification, for instance, `hours` must be between `0..23` and `minutes` must be between `0..59`. We call these the type's **invariants**.

But, with structures, we have **no means of enforcing those restrictions**. There is **nothing to prevent** someone, (most likely you, if you aren't careful), from constructing a bogus `Time` object like this:

```
Time bed_time = {27, 95};
```

Both values supplied here makes the `Time` object, `bed_time`, **invalid**. But, the code compiles fine; everything is perfectly legal C++, and the compiler has no idea that something bad might happen in the future.

# A Scenario

**To give an idea of what could go wrong, suppose that you have** a **Radiation Therapy Machine** like the Therac-25. If the software controlling the machine used a `Time` object to specify **how long a therapy session should last**, the machine would be **intrinsically unsafe**. Think about what will happen if you write the following code using the (non-buggy) `run()` function:

```
Time treatmentTime = {0, -2};
run(treatmentTime);
. . .
void run(Time& t)
{
    auto elapsed = t.hours * 360 + t.minutes * 60;
    while (elapsed > 0)
    {
        pulseBeam();
        --elapsed;
    }
}
```

As Peg and Cat point out, you now have a fairly serious problem. Even though the `run()` function is reasonable, it relies on the `Time& t` parameter **being correctly initialized**. Because `minutes` was, (accidentally), set to a negative number, the loop will supply **not** two minutes of radiation, but **billions of pulses** instead.

## The Problem

**If you're lucky, the code will have some extra checking to catch this, and** report an error. If you are unlucky and the code actually sends too much radiation to the patient, then they would die, just as in the original Therac-25 incident.

In other words, because the user of the `Time` structure set a single field to a nonsensical value, it's possible that your program could cause real injury. **This is clearly unacceptable, and you will need to do something about it.**

There are two problems with implementing `Time` as a `struct`.

- **Structures do not enforce invariants**. Structures use "naked" variables to represent data, so **any part of the program** can modify those variables without any validation. `Time` **expects** certain relationships between its data members, **but cannot enforce those relationships**.
- `Time` is **represented in a particular way**, as two `int` members. We say that code which uses the `Time` data type is **tightly coupled to that implementation**.

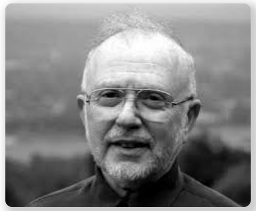
Both of these are real problems, and this is what C++ programming is like with raw structures. Code is **brittle**, bugs are more likely, and **changes are more difficult**. So, in the next lesson, let's change gears and represent `Time` in a slightly different way.

# Information Hiding

**The `Time` structure from the last lesson did not enforce its invariants.** **Structures** use "naked" variables to represent data, so **any part of the program** can modify those variables without any validation. `Time` **expects** certain relationships between its data members, **but cannot enforce those relationships**.

In addition, `Time` is **represented in a particular way**, as two `int` data members. Thus code that uses the `Time` type is **tightly coupled to that implementation**.

These problems with structured data were noticed in the early 1970s by a Canadian Computer Scientist named David Parnas, who developed a theory of **information hiding**, and who led the drive towards **modular programming** with his famous 1971 paper, *On the Criteria to be used in Decomposing Systems*, written at Carnegie Mellon University.



## The Public Interface

**The goal of information hiding is to make it possible for clients to use `Time`** objects **without** ever directly accessing the data members themselves.

To do that, you create a collection of **functions**, which provides **a public interface** for your type. These functions are specially dedicated to an individual class, so they are called **member functions**.

Your interface should be as **small as possible**, yet comprehensive enough to meet all of your client's needs. What features might those be? (Let's consider **`H01`**).

- **Arithmetic**: calculate the **difference** and **sum** of two `Time` objects
- **Input and Output**: read into and write out a `Time` object

Here is your `Time` structure, including the above interface:

```
struct Time
{
    int hours;
    int minutes;

    Time sum(const Time& rhs);
    Time difference(const Time& rhs);

    std::istream& read(std::istream& in);
    std::ostream& write(std::ostream& out);
};
```

# Member Functions

**Here is your `Time` structure, including the above interface:**

```
struct Time
{
    int hours;
    int minutes;

    Time sum(const Time& rhs);
    Time difference(const Time& rhs);

    std::istream& read(std::istream& in);
    std::ostream& write(std::ostream& out);
};
```

The member functions are written **as prototypes**, inside the structure definition. This is a user-defined type, so you will normally put it inside a header file. Because of that, the **library types**, `istream` and `ostream` are **fully qualified**.

- The `read()` and `write()` functions both take a stream as an argument, so that you can use either the console or a file. Stream arguments must **always** be passed (and returned) by reference.
- The `sum()` and `difference()` member functions will return a new `Time` object.

# Proving the Interface

**When designing an interface, its often useful to write a client program, just to** try it out. This is called **proving your interface**. You may find that you need additional member functions. Or, you might find that the prototypes for the functions are not exactly what you need to complete your task, and you can change them at this stage.

Here's the `run()` function which will act as the client for your new `Time` type. This is a revision of `H01`, using member functions.

```
int run()
{
    printHeading(); // already written

    Time startTime;
    Time duration;

    // Prompt and read the input
    cout << "     Time: ";
    if (! startTime.read(cin)) { return die(); }
    cout << "     Duration: ";
    if (! duration.read(cin))  { return die(); }

    // Processing section
    Time after = startTime.sum(duration);
    Time before = startTime.difference(duration);

    // Output section
    duration.write(cout) << " hours after, and before, ";
    startTime.write(cout) << " is [";
    after.write(cout) << ", ";
    before.write(cout) << "]" << endl;

    return 0;
}
```

The interface looks OK, so let's go ahead and **implement** the member functions.

## The Implementation File

**The header file defines the instance variables used to store the attributes or** properties. The implementation file, which typically uses the structure name with a `.cpp` extension, **provides a definition** for each **member function** defined in the interface.

Here's a starter for the implementation file:

```
#include "time.h"
#include <iostream>
using namespace std;
```

1. `#include` the header file with the class definition. If you don't, the compiler will flag all of the member functions as errors.
2. Surround the header name in "double quotes" **`not`** <angle brackets>, which the preprocessor sees as instructions to look for standard library files.
3. `#include` any standard libraries that the implementation uses. Here that is the `<iostream>` library, which is used for the `read()` and `write()` member functions.

Because this is an implementation, not an interface file, you may include a `using namespace std;` preprocessor directive.

# Implementing Member Functions

**To define a member function, specify the name of the function preceded by the name of the structure that it belongs to**. To implement write(), for instance, write:

```
ostream& Time::write(ostream& out)
{
    // format and print output here
    return out;
}
```

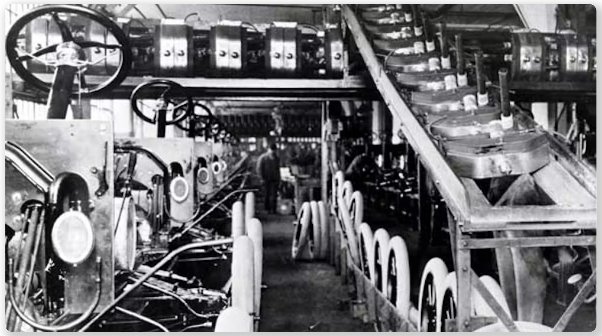
The name of the member function is `Time::write`; the double-colon operator (`::`) is called the **scope resolution operator** and tells C++ where to look for the function.

You can think of the syntax `X::Y` as meaning *"look inside **X** for **Y**."* It is important to use the **fully-qualified name** of the function when implementing it. The code shown below may compile, but C++ thinks you are implementing a regular (or **free**) function named `write()` that has **no relationship whatsoever** to the `Time` class.

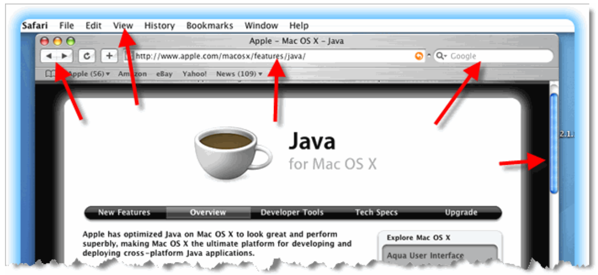
```
ostream& write(ostream& out)
{
    // Error... not a member function
    return out;
}
```

# Introducing OOP

**Procedural (aka structured) programming works well when applied to linear** problems like processing the monthly payroll or sending out a set of utility bills. You feed a list of employees and their hours into one end of a program and get a pile of paychecks out the other. You organize your programs as **assembly lines which processes data**. This works very well.



But how do you apply the assembly line model to your Web browser? Where is the beginning? Where's the end?


Safari File Edit View History Bookmarks Window Help
Apple – Mac OS X – Java
http://www.apple.com/macosx/features/java/
Google
Apple (56) ▾ Amazon eBay Yahoo! News (109) ▾
Java
for Mac OS X
New Features Overview Developer Tools Tech Specs Upgrade
Explore Mac OS X
Aqua User Interface

Apple has optimized Java on Mac OS X to look great and perform superbly, making Mac OS X the ultimate platform for developing and deploying cross-platform Java applications.

When it comes to **interactive** software, a better method of organization is needed, and that's where **OOP** (**O**bject-**O**riented **P**rogramming) comes in. In an object-oriented program, the basic "building-block" is not the function, but the **object**. OOP programs look **more like communities** than assembly lines. Each object has its own attributes and behavior, and your program "runs" as the objects interact with one another.

## What are Objects?

**Objects are simply variables of programmer-defined types. Objects can** represent **real things**, like employees or automobile parts or apartment buildings; objects can also represent more abstract concepts, such as relationships, times, numbers, or black holes.

Regardless of what kind of objects you use, you'll want to be able to recognize an object when you meet one. That's easy to do, because all objects have three properties:

- **Identity**: who the object is
- **State**: the characteristics of the object
- **Behavior**: what the object can do

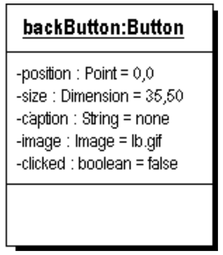
## Object Identity

In C and C++, we often use the term **object** to refer to a "bucket in memory" that contains **data values** of a particular type. In this sense, regular variables are objects:

```
int little = -4;
int big = 1795321;
int& tiny = little;
```

The names little and big are different areas of memory storing integer values. Changing little, it won't affect the variable big; the variables have **different identities**. But, tiny is **not** another variable but a **different name**; tiny and little have **the same identity**; they are different names for the same object.

## State and Behavior

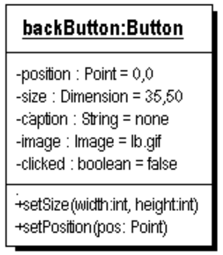


**The second property is object state. The state of an object** includes all the information about the object **at a particular time**. Take a look at the "Back" button on your Web browser. The **UML** (**U**nified **M**odeling **L**anguage) **object diagram** to the right displays a `Button` object named `backButton`. A `Button` object might have attributes like:

- **Position**: where the button is located on the screen
- **Size**: the button's width and height
- **Caption**: any text, such as the word "Back," that the button displays
- **Image**: any icon or image that is displayed on the button's surface
- **Clicked**: whether or not the button is currently selected (pressed)

The state of the object is represented by the **values of those attributes**. The `backButton` may display an arrow image but no text, and, if you click on the button with your mouse, its `clicked` state may change from `false` to `true`.

## Object Behavior



The third property shared by all objects, and what differentiates them from structure types, is **behavior**. The behavior of an object consists of the messages that it responds to.

In the UML diagram on the right, the behavior is implemented by the **member functions** appearing in the bottom portion of the box. You ask the `backButton` object to change its size, for instance, by sending it a message with the desired size like this:

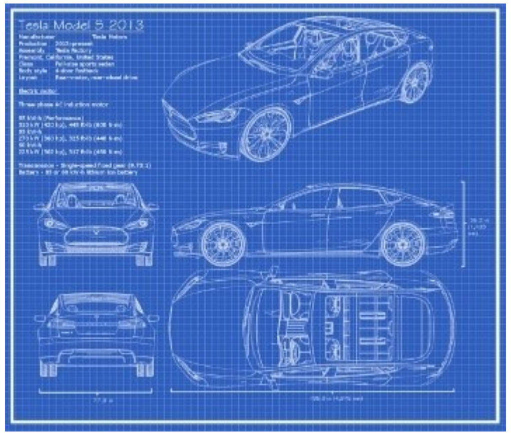
```
backButton.setSize(300, 100);
```

Since the `backButton` object has a `setSize()` member function, (as shown in the UML diagram), it does as you've asked.

# What are Classes?

**A class represents the definition—the blueprint if you like—used to create** objects. Objects are simply variables, created (or **instantiated**) from this blueprint.

Like a structure, a class describes the **attributes of an object**: the kinds of data it stores internally. To design a Car class, you specify the physical characteristics that car shares: its serial number, body type, color, type of interior, engine size, etc.



Such attributes are stored as the object's **data-members** ( **instance variables** in Java).

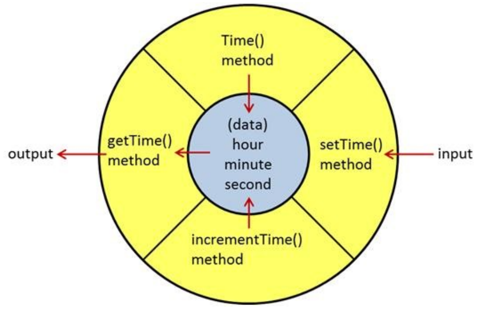
A class also **describes and implements the behaviors of its objects**: the kinds of operations that each object in the class can perform. When you define a class, you need to specify **what the object can do**, providing an explicit list of its possible behaviors.

These are specified as **embedded functions**, called **member functions** in C++; in Java these are called **methods**. Member functions contain the **interface** (as prototypes in the class definition), and the instructions (appearing in the **implementation**) that tells each particular object how to complete a particular task.

To ask an object to perform some action, **invoke** or **call** one of its member functions. To emphasize the fact that objects represent fairly self-contained, autonomous units, the process of invoking a method is often called **sending a message** to the object.

# What is Encapsulation?

**With structures, the functions that make up the program, and the data the** functions operate on, **are separate**. In an object-oriented program, **they are combined**. This process of wrapping up procedures and data together is called **encapsulation**.



Encapsulation is used to enforce the principle of **data hiding**, and, to allow your objects to **enforce their own invariants**, as we saw in the last chapter. With encapsulation, the data members defined inside a class are accessible to all the member functions defined inside the same class, but cannot be accessed by methods outside that class.

As you saw with the `Time` structure, making the data publicly accessible risks accidental data corruption as a result of a bug in someone else's code. The `struct` version of the `Time` type provides **no abstraction** and **no encapsulation**.

The `Time` **interface is its implementation**–the operations that clients can perform on the `Time` object are simply **direct manipulation** of its data. Changing the implementation thus **changes the interface**, which is why changing the data members breaks existing code.

# Encapsulation in the "Real" World

**You might find encapsulation a strange idea; why make it harder for your** programs to access their data? In fact, out in the real world, all of us are familiar with the ideas of encapsulation. Let me give you a few examples.

- **Today's automobiles** are more complex than Henry Ford's original car. Despite this, **driving** the latest Tesla is similar to, if not simpler than, driving a Model-T. Why, because, as cars got more complex, that complexity was **hidden** behind a **simpler** interface: the ignition (key), steering wheel, accelerator and brakes. These internal changes don't require drivers to take a new "how to drive" course. The **implementation details** are **encapsulated**.
- **Your computer** is another marvel of complexity. Unless you are a hardware tech, though, you never open up the system unit and try to operate it by shorting the circuits with a paper-clip. Instead, you use the **interface**—the mouse, and keyboard — to control the complex working parts that it contains.
- Finally, think about **your TV**. It's probably at least as complex as your car or your computer, but you don't need a license or a degree to operate one. Thanks to the magic of encapsulation, exemplified by the **remote control**, every child in the country can harness that power, although if you are a parent or grandparent, you might wish that were not true.

Just as with automobiles, computers and TVs, when it comes to programming, instead of making things more difficult, encapsulation makes objects safer **and** easier to use.

Encapsulation is one of the pillars underlying OOP or Object-Oriented Programming. We'll cover the other two, **inheritance** and **polymorphism**, next week.