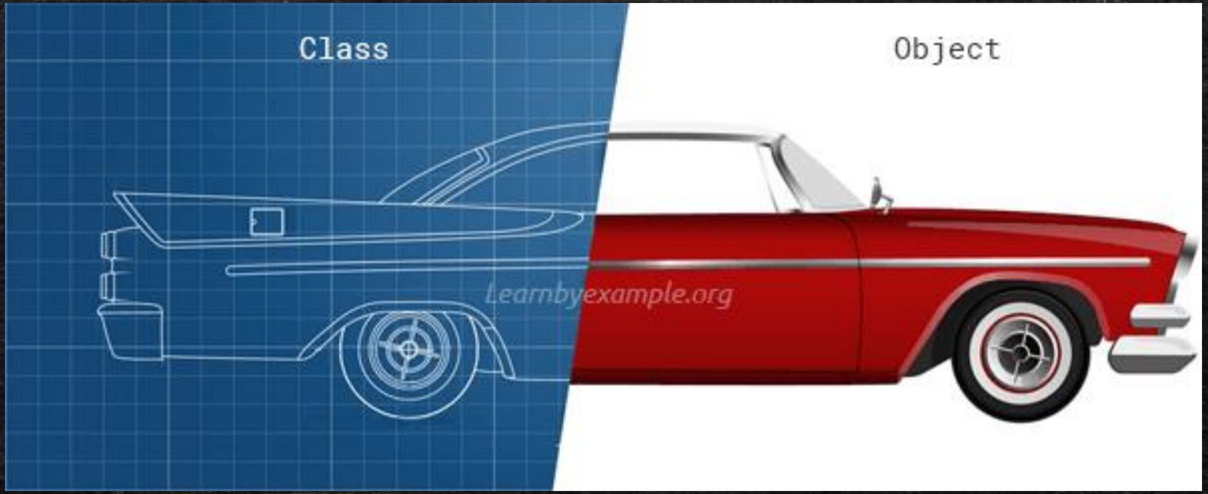
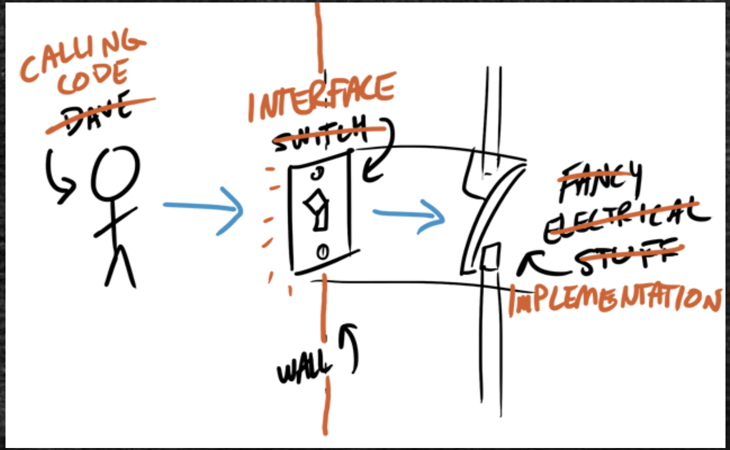
Objects & Classes



CS 150 – C++ Programming I
Lecture 25

The Wall of Abstraction



```
struct Date
{
    int day;
    int month;
    int year;
};
```

```
Date d1 = {2, 2, 1950};
d1.day = 75;
```

```
struct Date {
  long long daysFromZero;
};
```

- Abstraction is interaction through an interface
  - With structures, the implementation is the interface
- Since you can directly access the data members it is inherently unsafe, error prone, and inflexible

Classes & the Wall of Abstraction

- A class is an interface paired with an implementation
- The interface will contain the public facing portion
  - The part that users interact with (like the switch on the wall)
  - In C++, we put this into a header file in the class definition
- The implementation is the hidden or private portion
  - This includes the data members which are encapsulated
  - It also includes the member function definitions
- In C++, a `struct` is a class with default `public` members
  - But, in CS 150, we'll use `struct` only for POD types

Class Definition Syntax

```
class Date
{
public:
    Date(int d, Month m, int year);
    void addDays(int days);
    Month month() const;
    string toString() const;
private:
    . . .
};
```

Public Interface

- ✓ Members accessible by users
- ✓ Prototypes
- ✓ Constructor initializes object
- ✓ Mutators change object
- ✓ Accessors (const) cannot

Private Implementation

- ✓ Not accessible by users
- ✓ Data members and helpers
- ✓ Can change implementation

- Here is `Date` written as a `class`
  - Client uses `public` interface not data members
  - Unlike `struct`, constructor ensures all objects initialized & valid
  - `private` data members are encapsulated inside class
- Exercise: define the class specified

Implementing Member Functions

- Member functions are implemented in a .cpp file

```
#include "date.h"         // class definition
#include <string>         // used in implementation
using namespace std;      // OK in implementation

Month Date::month() const    // qualified prototype
{
    return . . .;    // whatever represents month
}
```

  – Separately compiled and then linked when used

Stubbing the Member Functions

- Memorize: should be second nature (place in `.cpp`)
  - 1. `#include` the header file for your class
    - May include library and `using namespace std;`
  - 2. Copy the prototypes from interface section
    - Don't copy the preprocessor directives
  - 3. Qualify each prototype. `Class-name::function`
    - Don't put it before the return type
  - 4. Remove the semicolon and supply a body
  - 5. Provide a return type and return before going on
- Exercise: implement members with stubs

Representing State

- Next, we need to decide what data members to use
  - Should we use int and Month?
  - How do we provide a default value (today)?
- The standard library has a header <ctime>
  - Provides types and support for hardware time
  - time_t – number of seconds since Jan 1, 1970
  - Makes calculations and printing easier
- Exercise: add data member to date
  - Use time_t as type, name cur_time
  - (Just because my test code uses that name)

```
class Date
{    . . .
private:
    int m_day;
    Month m_month;
    int m_year;
};
```

Default Constructor

- Objects should always be in a valid state; all members initialized
  - Constructors can automatically initialize every object
  - Unlike built-in types or structs which allow uninitialized objects

  -
    ```
    int a;          // uninitialized
    string b;       // constructor automatically called
    ```
- No-argument constructor sets default values for data members
  - Same name as the class: `Date::Date() {. . .}`
  - No return type: **not** `void Date::Date() {. . .}`
  - Used like `Date today;` not `Date today(); //` *`prototype`*

Default Constructor & toString

- Exercise: complete the default constructor and toString

- Default constructor sets the Date to the hardware time
  - Use time(0) to initialize cur_time data member
  - Returns the current time in Universal Time Coordinates (UTC)

- To implement the toString member function call the strftime and gmtime functions in <ctime>
  - "String formatted time", "Greenwich Mean Time" (UTC)

  -
    ```
    char buf[100];              // temporary buffer
    strftime(buf, sizeof(buf),
       "%u %d %m", gmtime(&cur_time));
    ```

Working Constructor

```
class Date
{
public:
    Date(int d, Month m, int year);
};
```

- When you want to customize all parts of an object
- struct tm is <ctime> calendar type
  - tm temp = *gmtime(&cur_time) // time_t->tm
  - cur_time = mktime(&temp) // tm->time_t
  - temp.tm_mon: // month [0-11]
  - temp.tm_mday: // day [1-31]
  - temp.tm_year: // year - 1900
- Exercise: complete working constructor