Classes 2

constructors

Abstraction

2nd property
We want to provide an **interface** to our class.

- an interface is a simple, user-oriented way to access the functionality represented by our class
- the methods we define are that interface
information hiding

By abstraction, we are "hiding" the details of how a struct/class is implemented.

We design the interface, the methods, so that the user can access the functionality without worrying about details.

abstraction helps change

What if I take out that volkswagen engine and replace it with a porsche engine.
Much change, same interface

If
- the interface is well designed
- is respected by the people doing the changes

Then the user access to the underlying object should be the same

Data structures

Imagine that you make a class that implements a company inventory.
- you make the class and you use vectors for the underlying implementation
- you decide later to change the implementation to a map
- users should not care!!! Works the same for them (if you did it right)
Remember this

You must remember this, a kiss is just a kiss, a sigh is just a sigh.

C++ marks/remembers the calling object in a member function call

Clock my_c;
my_c.add_minutes(5);

in the member function add_minutes, the variable this points to my_c
my_clk.add_minutes(5)
   this
void add_minutes(int min)

On a method call, C++ automatically binds a variable named this to the calling object
It is a pointer!

```cpp
#include <string>
using std::string;

struct Clock{
    int minutes;
    int hours;
    string period;
    void add_minutes(int);
    string print_clk(const Clock &c);
};

void Clock::add_minutes(int min){
    auto temp = minutes + min;
    if (minutes >= 60){
        minutes = temp % 60;
        hours = hours + (temp / 60);
    } else
        minutes = temp;
}

naked data members in a member function are assumed to be associated with this
minutes + min is equivalent to this->minutes + min or (*this).minutes + min
```

```cpp
#ifndef CLOCK_H
#define CLOCK_H

#include <string>
using std::string;

struct Clock{
    int minutes;
    int hours;
    string period;
    void add_minutes(int);
    string print_clk(const Clock &c);
};
#endif
```
Constructors

Ex 16.1

**Constructors**

**what is a constructor**

We've seen these special member functions before (in Python, in Java).

These are the member functions responsible for creating/initializing a user defined struct/class.
Really more like initialize

Constructors are really more initializers than "creators", as they are part of a pipeline.

Your constructor fits into the pipeline, the creation process, allowing you to initialize elements of your data struct.
Default/synthesize constructor

So if you do not provide a constructor, C++ will synthesize a constructor. The *synthesized constructor* will initialize each data member to its default value:

- long $\leftarrow 0$
- double $\leftarrow 0.0$
- string $\leftarrow ""$

problems

- default constructor takes no arguments. A user cannot change the initial data members of a variable
- default value for each data member is OK for most types, but there are exceptions:
  - pointers are not initialized to a "useable" value
  - user defined types must have a default
constructor

Constructor is a function member with the same name as the class itself:

- there is **no return** from a constructor as it is already part of a pipeline
- not a `void` return, no return (no type)
- unlike Python, the constructor can be **overloaded** based on parameters
- many different constructors depending on parameters

clock constructors

```cpp
Clock::Clock(){
    minutes=0;
    hours=0;
    period="AM";
}

Clock::Clock(int min,
             int hr,
             string prd){
    minutes=min;
    hours=hr;
    period=prd;
}
```

```cpp
struct Clock{
    int minutes;
    int hours;
    string period;
    
    Clock();
    Clock(int m, int h,
          string s);
    
    void add_minutes(int);
    
    string print_clk(const Clock &c);
}
```
main

Clock my_clk; // call to default constructor, no args
// not even empty parens!!
Clock a_clk(1, 1, "PM"); // call to 3 arg constructor

• first declaration is a call to the user-defined default constructor
• second is a call to the 3-arg constructor

all or nothing

If you define any constructor, then C++ no longer provides a synthesized default constructor
• when you define a constructor, it is up to you to provide all the constructors necessary for your class.
• if you still want a default constructor (a no-argument constructor), you have to provide it.
construction shortcuts

Ex 16.2

```cpp
#ifndef CLOCK_H
#define CLOCK_H
#include<string>
using std::string;

struct Clock{
    int minutes = 0;
    int hours = 0;
    string period;

    Clock()=default;
    Clock(int m, int h, string s) : minutes(m), hours(h), period(s) {};

    void add_minutes(int);
};

string print_clk(const Clock &c);
#endif
```
We said that if you define any constructor, the C++ default (the no arg constructor) can no longer be used.

However, if you're interested in using the C++ default, you can by using the = default designator on your no-arg constructor.

If you declare the no-param constructor (the default constructor) =default, it will respect default data member values.

```cpp
struct Clock{
  int minutes = 0;
  int hours = 0;
  string period;

  Clock()=default;
  ...
}
```

the default ctor will assign
minutes \( \leftarrow 0 \)
hours \( \leftarrow 0 \)
period (call the std::string default ctor)
initializer list

If all you are doing (as we are doing in the Clock example) is setting a data member directly to some parameter, there is a shortcut.

This is called the initializer list.

format

Clock(int m, int h, string s) : minutes(m), hours(h), period(s) {};

- colon indicates what follows is an init list
- each comma separated phrase afterwards is the name of a data member and, in parens, the name of the parameter used to set that data member.
- the empty {} is **required** at the end.
  - could provide code here if you choose, but it should be short!
order depends on declaration

The order of initialization of data members from an initialization list goes in the **order of declaration** in the class, not on the order of parameters in the initializer constructor.

- you'll get a warning if the param order and declaration order differ.
- it could matter to the code as well!

.h vs .cpp

You can put the constructor in the .h or the .cpp. Traditionally:

- initializer list constructors go in the header
- constructors that "do work" i.e. require a function body to do something, go in .cpp
.h means inline

if you put the constructor in the .h, then that means these constructors will be **inlined**.

Instead of creating a function, everywhere that the constructor occurs is physically **replaced** with the appropriate code to do construction

- should be simple, as this could be an expensive process.

---

**well, kind of anyway**

Inlining is an interesting process, whose consequences are difficult to easily ascertain. However, the compiler is free to do as it wishes with inlining

- so even though it is inlined, it may in fact be turned into a regular function by the compiler
- compilers are free to optimize things as they choose 😊
advertising vs implementation

You try to keep implementation out of the header when possible. Remember:
• the header is the ad for the class. This is what the class does
• the implementation file is how the class does what is advertised.

type conversion

"to" conversion
Ex 16.2
there are two senses of cast

- **to**-casting: cast a known type to a new a variable of your class type
- **from**-casting: cast a variable of your class type to a known type

to-casting is easy!

to casting is construction

If you write a constructor with a single parameter, then that constitutes a to-cast.

- when C++ sees a type that, when passed to a constructor, creates the required type, it will call that constructor and do the conversion.
```cpp
#ifndef CLOCK_H
#define CLOCK_H
#include<string>
using std::string;

struct Clock{
    int minutes;
    int hours;
    string period;
    Clock()=default;
    Clock(int m, int h, string s) : minutes(m), hours(h), period(s) {} // to-cast
    // explicit Clock(string);
    void add_minutes(int);
};

string print_clk(const Clock &c);
#endif
```

A to-cast, from string to Clock
if explicit, then compiler cannot call it implicitly, but programmer can call explicitly

ctor with string param, expects "hr:min:period"

Clock::Clock(string s){
    // format is hr:min:period
    vector<string> fields;
    split(s, fields, ':');
    hours = stol(fields[0]);
    minutes = stol(fields[1]);
    period = fields[2];
}
```
The call to the one-string parameter could be used by C++ *implicitly*, that is without being explicitly called by the user (like a long → double conversion in mixed math, done by the compiler)

The keyword `explicit` in front of a constructor means that it will not be called implicitly by C++, but can be called explicitly by the user

```cpp
string clk_to_string(const Clock &c) {
    ostringstream oss;
    oss << "Hours:" << c.hours <<", Minutes:" << c.minutes <<", Period:" << c.period;
    return oss.str();
}
```
only one conversion at a time

```
string s="12:12:PM";
cout << clk_to_string(s) << endl;
cout << clk_to_string(string("11:11:PM"));
// cout << clk_to_string("11:11:PM");
```

Last one won't work. A literal character string is not an STL string object. So this requires two conversions:

```
char* → string → Clock
```

default params in ctor can be a problem

Slightly modified .h file

```
Clock(int m=0, int h=0, string s="") :
    minutes(m), hours(h), period(s) {};
Clock(string s);
Clock("11:11:PM");
Which one??
```
default constructor

In fact, a constructor that defaults all of its parameters is defining the default constructor

• could call it with no args, so default.