More Classes

James Daly
CSE 232
Michigan State University
Abstraction

- 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 the details.
Abstraction helps change

- What happens if I remove a Volkswagen engine and replace it with a Porsche engine?
- 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

- Imaging that you make a class that implements a company inventory
  - You make the class and you use a vector 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)
Special variable

- C++ marks / remembers the calling object in a method call
  
  ```c++
  Clock my_c;
  my_c.add_minutes(5);
  ```

- In the member function `add_minutes`, the variable `this` points to `my_c`
This

```cpp
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
- `this` is a `pointer`!
Clock Class

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

#include <string>
using std::string

struct Clock {
    int minutes;
    int hours;
    string period;
    void add_minutes(int);
};

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

Naked damnate members are assumed to be associated with this

minutes + min is equivalent to
this->minutes + min

#ifndef CLOCK_H
#define CLOCK_H

#endif
Constructors

- Example 16.1
What is a constructor

- We’ve seen these special methods before (in Python or Java)
- These are the methods responsible for creating / initializing a user defined struct / class
Really more like initializers

- Constructors are really more initializers than “creators” as they are part of a pipeline
- Your constructor fits into the creation process pipeline allowing you to initialize elements of your data struct
Pipeline

Create a variable → Apply class template → Initialize with Constructor → Object initialized
Default /synthetic constructor

- If you do not provide a constructor, C++ will provide one
- The *synthesized constructor* will initialize each data member to its default value
  - long: 0
  - double: 0.0
  - string: “”
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
  - Points are not initialized to a usable 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
    - 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

Clock::Clock() {
    minutes = 0;
    hours = 0;
    period = “AM”;
}

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

struct Clock {
    int minutes;
    int hours;
    string period;
}

Clock();
Clock(int m, int h, string s);

void add_minutes(int);
Main

Clock my_clk;  // Call to default constructor, no args
                    // Not even empty parents!!!
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 any 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 (no-argument) constructor, you have to provide it
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);
};
Get the C++ default back

- We said that if you define any constructor, the C++ default (the no-arg constructor) can no longer be used.
- If you are interested in using the C+ default, you can by using the = default designator on your no-arg constructor
Default uses default data member values

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

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

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

- 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 parameters used to set that data member.
- The empty {} is **required** at the end
  - Could provide code here if you chose, 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** the order of parameters in the initializer constructor
  - You’ll get a warning if the param order and the 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
- Example 16.2
There are two senses of cast

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

```
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) {};

    // to-cast
    Clock(string)
    // explicit Clock(string);

    void add_minutes(int);
};
```

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];
}

**Explicit**

- 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.
Conversion

```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

```cpp
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.
- This requires two conversions: char* -> string -> Clock
Default params in ctor can be a problem

- Slightly modified .h file

```cpp
clock(int m=0, int h=0, string s="") :
    minutes(m), hours(h), period(s) {};
```

```cpp
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