Classes

James Daly
CSE 232
Michigan State University
Object-Oriented Programming

- Object-oriented programming (OOP) is a view of how both data and function that work with that data can be grouped together as a single programming entity.
- This organization is typically called a class
- Complexity is the biggest problem faced by a programmer. OOP is one way to control complexity
Class Implementation

- A class (or struct) contains two things
  - data
  - functions that operate on that data
- Organized together so it is easier to keep track of
OOP Principles

- General Principles
  - Composition
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

- Different languages take different approaches to these principles
Composition

- A type with parts
- (Example: mailbox to cubbies?)
What is a type again?

- A type has a number of aspects
  - The elements that are part of a type
    - Ex: a fraction has a numerator and a denominator
    - The size and number of elements in a type determine its size
  - Methods (member functions) that can be applied to the type
A struct (short for structure) is a way to compose a new type (that we can declare, that we can pass to a function, etc) where we can decide what the underlying parts of the type consist of
Clock

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

**Keyword**

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

Type name, Capitalized by convention

Typed variables
Part of the struct

Semicolon required!

Block
Clock is now a type

- The `struct Clock` is now a type
- We can use it to declare a variable of type `Clock`
Separate declaration and definition

- Typically, we place the structure definition in the header file and then any functions associated with the structure in an implementation file
- No functions yet, just the declaration
Definition in a header file

```c
// main
#include "clock.h"
int main() {
    Clock my_c;
}

// Clock.h
struct Clock {
    int minutes;
    int hours;
    string period;
};
```
Instance vs Class

- An instance (such as my_c) is a variable created from the Clock pattern
  - An instance / variable is what we typically manipulate
- The type / class is the pattern we want all instances to follow
Access

- Using data members
- Example 14.1
How to access the struct elements

- Once we create the variable `my_c` of type `Clock`, we can manipulate the elements that are present in every `Clock` instance.

- Every variable of type `Clock` has
  - an integer `minutes` variable
  - an integer `hours` variable
  - a string `period` variable
Proper term, member

- The proper term for the elements present in a variable of a `struct` is `data member`.
- A variable of type `Clock` has 3 `data members`: `minutes`, `hours`, `string`.
- We defined those three in the `struct`.
Two types of members

▪ Broadly speaking, a struct has two general types of members
  ▪ data members
  ▪ function members (methods)
Member access

- This is the same as it was in Python
- `my_c.hours`
  - Refers to the `hours` member of the variable of type `Clock` called `my_c`
Data member access

// main
#include "clock.h"
int main() {
    Clock my_c;
    my_c.hours = 10;
    cout << my_c.hours << endl;
}

// Clock.h
struct Clock {
    int minutes;
    int hours;
    string period;
};
More access

- As a programmer you can:
  - Access the value of a data member
  - Set the value of a data member
  - Just like you can any other variable
Refs and Ptrs

- **clock** is a type just like **int or string**.
- We can make references and pointers to it just like we could for any other type.
Example

```cpp
#include "clock.h"
int main() {
    Clock my_c;
    Clock &ref_c = my_c;
    Clock *ptr_c = &my_c;
    my_c.hours = 10;
    ref_c.minutes = 20;
    ptr_c->period = "A.M.";
    cout << my_c.hours << endl;
}
```
Remember -> syntax

- Remember

```c
Clock *ptr_c = &my_c;
(*ptr_c).hours = 10;
ptr_c->hours = 10;
```

- Last two statements mean exactly the same thing
  - deref pointer
  - set member of deref
Pass a Clock var to a function

```cpp
string print_clk(const Clock &c) { 
    ostringstream oss;
    oss << "Hours:" << c.hours << ", Minutes:" << c.minutes << ", Period:" << c.period;
    return oss.str();
}
```
First Clock

- Example 15.1
Functions working with Clock

- We put functions that work with Clock or are a part of Clock in a separate implementation file
Function Members

- Example 15.2
Function members -> Methods

- Besides **data** members, we can also have **function** members
  - Better name: **methods**
- Methods have some special properties
  - Called in context of an object
  - Special privileges
How called

- We use a . to call a method in the context of an object
  - Ex: my_c.add_minutes(5);
  - Call the method add_minutes in the context of the my_c variable of type Clock passing 5 as an argument
Methods are specific to a type

- Methods are specific to the struct / class / type they are associated with
  - We can call `add_minutes` on a `Clock` as `add_minutes` is part of `Clock`
  - Can’t call `add_minutes` on a string. No such method is defined for use by a string
Declare method inside of struct block

- To make a method, we declare the method **inside** of the block of **struct**
  - Indicates it is part of the **struct**
  - This is only the declaration
    - Still need a definition
Definition add_minutes

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

- `Clock::add_minutes(int min) { ...`
- Scope resolution operator
- The method `add_minutes` is in the scope of the `Clock` struct when it is defined.
Can call as a member

- By declaring `add_minutes` to be part of `Clock`, we can call it as we indicated, as a member function of a `Clock` variable
  - `Clock clk;`
  - `clk.add_minutes(5);`
- Not so for `clk_to_string`, just a function
  - `clk_to_string(clk);`
How is calling object passed?

Clock clk;
clk.add_minutes(5);
vs
clk_to_string(clk);

- Clear in function (2\textsuperscript{nd}) how a Clock instance is passed.
- How is it passed to the method (1\textsuperscript{st})?
Self

- In Python, we said that the first parameter to every method was the calling object. We always called it `self`.

```python
my_clk.add_minutes(5);
```

```c
void add_minutes(???, int min)
```

- Is there a `self` here?
The special variable this

- There is no “first parameter” in every method.
- Instead, C++ creates a special variable name `this` which is used in a method call.
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**!
implicit pointer for members

Clock::add_minutes(int min) {
    auto temp = minutes + min;
...

- In the above, minutes is a member of the struct.
- In the context of a method, it is assumed that using a “naked” data member means: “the data member associated with the variable this”
Rephrase

Clock::add_minutes(int min) {
    auto temp = minutes + min;
    ...
    ▪ It is as if you had type the below
    auto temp = (*this).minutes + min;
    ▪ or
    auto temp = this->minutes + min;
    ▪ All three are equivalent (you can do whichever you like)