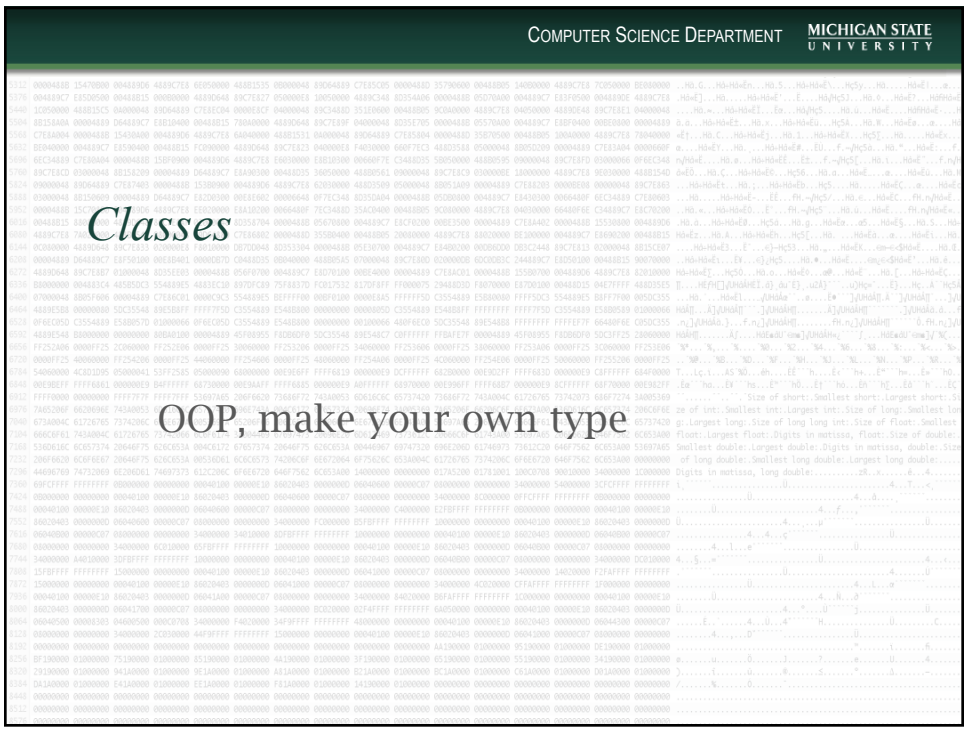


Object Oriented Programming

You know this drill. C++ introduced OOP to C-type programming.

OOP, what is that again?

FAQ (1)

1. What is Object Oriented Programming (OOP)?

OOP is a view of how both data and functions that work with that data can be grouped together as a single programming entity. This organization is typically called a **class**.

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

FAQ (2)

2. Why do we need it?

Complexity is the biggest problem faced by a programmer. OOP is one way to control complexity.

3. How does OOP help complexity?

A class is created for other programmers by a **class designer**. The designer creates a class to manipulate class data in a more "natural" way, above the details of implementation, such that the class is: easy to use, reliable, secure, efficient etc.

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

FAQ(4)

4. How is a class implemented in C++?

The easiest way to think about it is that C++ uses a `class` (or `struct`) to organize data and functions as a new **type**. Once created by the class designer, other programmers can use this type

FAQ(5)

5. What principles are embraced by OOP in C++?

There is not firm agreement on all aspects that an OOP/class system should have. Different languages in fact take different approaches. However, here are some principles that most would agree on and which do show up in C++.

FAQ(6)

- Composition
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Composition

1st property

A type with parts

type mailbox

underlying data in mailbox

what is a type again

A ***type*** has a number of aspects

1. the elements that are part of a type
   1. example: fraction has a numerator and a denominator
   2. the size and number of elements in a type determine its size
2. functions, really methods, that can be applied to the new type

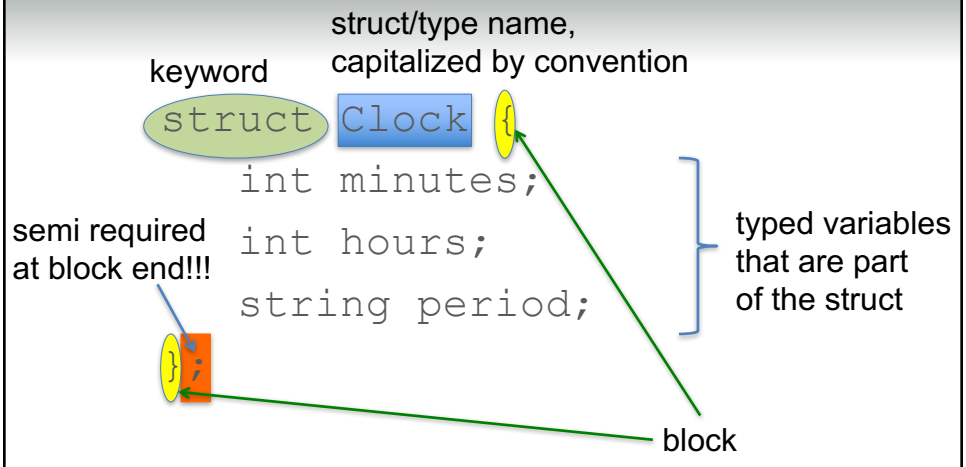
a struct

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;
};
```


struct breakdown



Clock is now a type

The `struct Clock` is now a type. We can use it 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 so…

definition in a header file

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

```
// clock.h
struct Clock{
   int minutes;
   int hours;
   string period;
};
```


Instance vs Class

Remember this discussion?

- an instance (here `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/variables to follow

access
using data members
Ex 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/variable.

Every variable of type `Clock` has:

- and integer `minutes` variable
- integer `hours` variable
- string `period` variable

proper term, *member*

In fact, 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`.

In general, two kinds of members

Broadly speaking, a `struct` can have two general types of members:

- data members
- function members

We'll start with the data members we've already seen.

member access

This is the same as it was in Python (if you remember):

The statement:

```
my_c.hours
```

refers to the `hours` member of the variable of type `Clock` called `my_c`

data member access: `var.member`

```
// 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.

also refs and ptrs

`Clock` is a type like any other type. So we can make references and pointers just like we could for any other type.

```
#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
```

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

remember -> syntax

Remember:

```
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

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


First clock

Ex 15.1

main and header

```
#include<iostream>
#include<string>
using std::cout; using std::endl;
using std::string;

#include "15.1-clock.h"

int main (){
  Clock my_c;
  Clock &ref_c = my_c;
  Clock *ptr_c = &my_c;

  my_c.hours = 10;
  ref_c.minutes = 10;
  ptr_c->period = "A.M";

  cout <<"My_C:"<<print_clk(my_c)
       <<endl;
}
```

```
#ifndef CLOCK_H
#define CLOCK_H

#include<string>
using std::string;

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

string print_clk(const Clock &c);

#endif
```


functions working with Clock

We put functions that work with Clock, or are a part of Clock, in a separate implementation file.

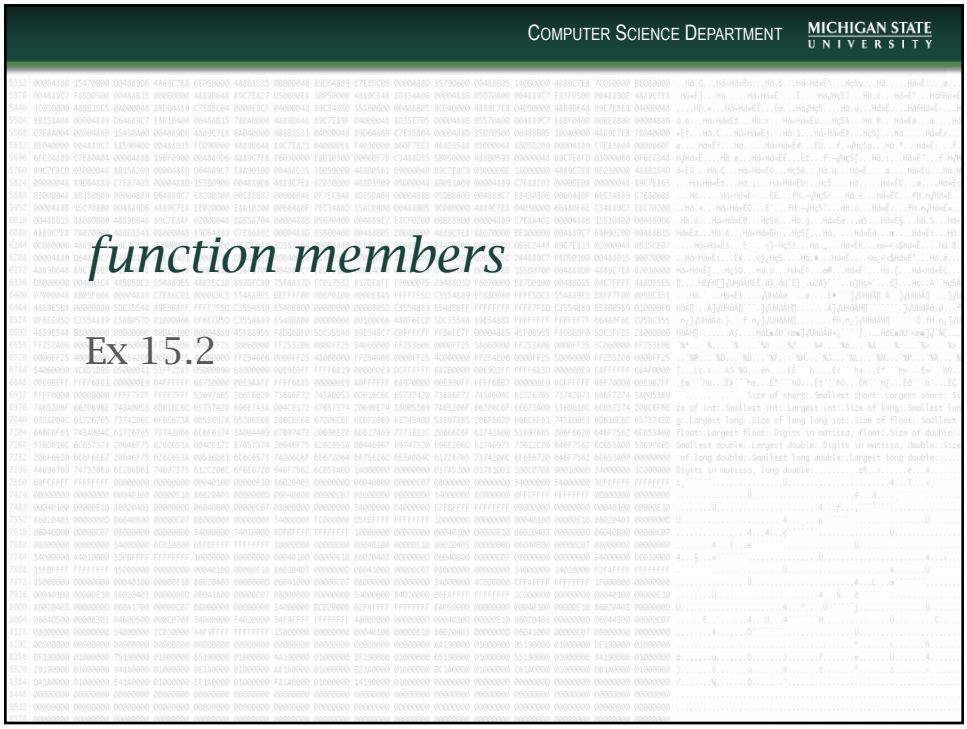
implementation

```
#include<string>
#include<sstream>

using std::string;
using std::ostringstream;

#include "15.1-clock.h"

// a function!
string clk_to_string(const Clock &c){
  ostringstream oss;
  oss << "Hours:"<<c.hours<<", Minutes:"
      <<c.minutes<<", Period:"<<c.period;
  return oss.str();
}
```

function members

Ex 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

Without saying how to write one, how we call a method is something we do all the time. We use a `'.'` to call a method *in the context* of an object

```
Clock my_c;
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

interpretation

```
Clock my_c;
my_c.add_minutes(5);
```

This would mean:

"In the context of `my_c`, call the `add_minutes` method with the arg `5`"

You would guess it means to add 5 minutes to `my_c`

methods are specific to type

Because of the way they are called, methods are specific to the struct/class/type they are associated with:

- we can call `add_minutes` on a `Clock`. `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 the `struct`

- indicates it is part of the `struct`
- this is only the declaration
  - still need a definition

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

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

  void add_minutes(int min);
};
string clk_to_string(const Clock &c);

#endif
```

want `add_minutes` to be a *member* of Clock. Inside!

want `print_clk` to be just a regular function. Outside!

definition add_minutes

```
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

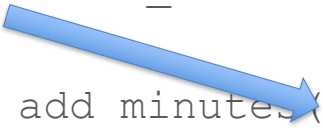
```
clk_to_string(clk);
```

Clear in function(2nd) how a `Clock` instance is passed, how is it passed in the function member (1st)?

self?

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

```
my_clk.add_minutes(5)
```



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

Is there a `self` here?

the special variable this

There is no "first parameter" in every method. Rather, C++ creates a special variable named `this` which is used in a method call

- unfortunate name really, confusing to say things like "this this" or "that this".
  - life is hard.

```
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! Yeah!

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 (no `object.` in front of method) 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 typed the below (which you can even do if you like, no difference)

```
auto temp = (*this).minutes + min;
```

or better

```
auto temp = this->minutes + min;
```