Encapsulation

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

Data Structures

- Imagine 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)

Does vector have any data members?

- Yes, it has multiple data members
- No, it has only function members
- It likely has data members, but it doesn't share them with others
- I don't know

Special variable

- C++ marks / remembers the calling object in a method call

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

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

This

```
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**!

Enforce

- Our Clock struct does not prevent a user from changing a value, even if it is a wrong value.

```
Clock my_c(11, 11, “PM”);
my_c.hours = 100; // stupid
```

- How can we be sure that what we set up is properly used?

Provide Protection

- The way we can “save the users from themselves” is to protect aspects of the class.
- Divide the world into two parts:
  - Class designer. Full access to everything
  - Class user. Only gets to use the interface the design provides

public vs private

- As part of the class declaration, we can declare parts of the class `public` or `private`
  - `public`: parts of the class to be used by everyone
  - `private`: parts of the class to be used by *other members of the class*

Clock Class

```
class Clock {
    private:
        int minutes;
        int hours;
        string period;

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

Class members that follow are private until something says otherwise. Note the colon.

Class members that follow are public until something says otherwise. Note the colon.

struct vs class

- Only one, very small difference
  - `struct`: if you don’t say otherwise, everything is assumed to be public
  - `class`: if you don’t say otherwise, everything is assumed to be private

google.github.io/styleguide/cppguide.html#Structs_vs._Classes

- Use a struct only for passive objects that carry data; everything else is a class.
  - structs should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behavior but should only be used to set up the data members, e.g., constructor, destructor, Initialize(), Reset(), Validate().

Implies separation of file

- If class designer vs class user is going to work we need to separate the files:
  - The class user includes (has access to) the provided header, but only has access to the compiled implementation (no source)
  - The class user, becauses he has no access to the class definitions, cannot change his access!

Different from Python

- Python had a saying: “We are all adults here”. Would warn you about changing things you shouldn’t but would let you
- C++ is made more for large groups of interacting people. Need to enforce access to keep everybody coordinated
- That’s the story anyway

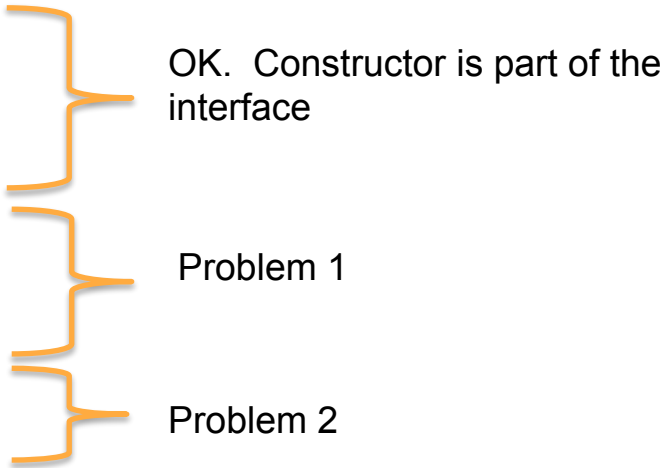
What effects does this have?

- No effect on the code in the class (members)
- private members can be accessed by members of the class
- Big effect on the design of the class. Class user cannot do many things we assumed
  - Class designer is fully in control and must provide interface access as they see fit
  - Must anticipate user needs!

Const Member Functions

Part of old main

```
int main() {
        Clock my_clk;
        Clock a_clk(1, 1, “PM”);
        Clock some_clk(“10:15:AM”);
        cout << clk_to_string(my_clk) << endl;
        cout << clk_to_string(a_clk) << endl;
        cout << clk_to_string(some_clk) << endl;
        my_clk.hours = 1;
        my_clk.minutes = 55;
        …
}
```



Problem 1

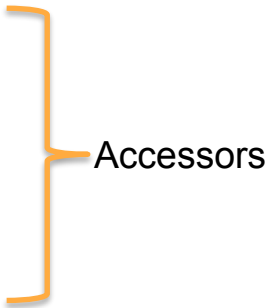
```
string clk_to_string(const Clock &c) {
    ostringstream oss;
    oss << “Hours:” << c.hours << “, Minutes: “
          << c.minutes << “, Period:” << c.period;
    return oss.str();
}
```

- function is no good anymore. Assumes it can access private data members and it cannot!
- Up to us to fix

Problem 2 first

- User can no longer access the private members of a class (which includes `hours`, `minutes`, and `period`)
- What to do?
- We need to provide methods for this:
  - Win: we control what goes in and out
  - Loss: We have more work to do

```
class Clock {
 private:
  int minutes_;
  int hours_;
  string period_;
 public:
  // Constructors
  Clock() : hours_(0), minutes_(0), period_("") {};
  Clock(int m, int h, string s) : minutes_(m), hours_(h), period_(s) {};
  Clock(string s);
  // Accessors
  int hours() const { return hours_; }
  void hours(int val) { hours_ = val; }
  int minutes() const { return minutes_; }
  void minutes(int val) { minutes_ = val; }
  string period() const { return period_; }
  void period(string s) { period_ = s; }
  // members
  void add_minutes(int);
  friend string clk_to_string(const Clock &);
}
```



Different names, accessors and data

- C++ (for various reasons) does not allow an accessor method to have the same name as a data member
  - Changed the data members to have an ‘_’ underscore at back
    - Google standard

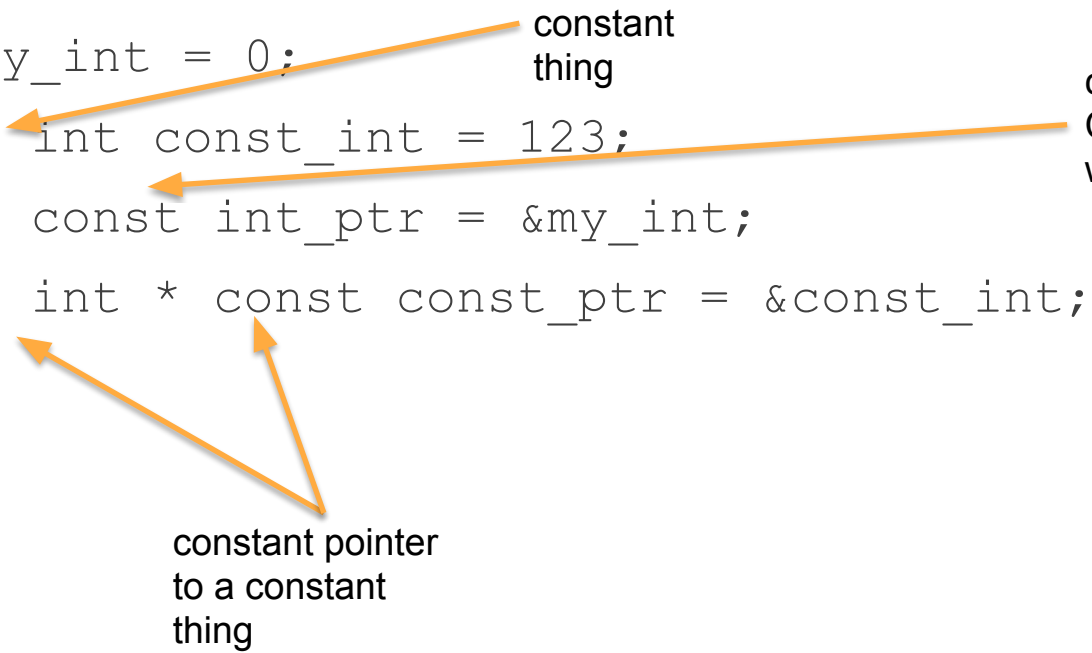
Accessor in header

```
int hours() const { return hours_; }
void hours(int val) { hours_ = val; }
```

- Couple of things here
  - If the code is simple, you can inline it here in the header
    - Note the `{}` with the statements within
    - Only simple stuff!
  - Can be overloaded (as is here)
  - What's the `const` there?

Remember this?

```
int my_int = 0;
const int const_int = 123;
int * const int_ptr = &my_int;
const int * const const_ptr = &const_int;
```



The this pointer

- The `this` pointer is a constant pointer
  - C++ sets the `this` pointer when a method is called and you cannot change what it points to in the member function
  - What if you want the `this` pointer to point to a constant thing?
    - You don’t want the member function to change a value in the object it points to

const at the end of the member function

```
int hours() const { return hours_; }
```

- This `const` means that the `this` pointer is a pointer to a constant thing
- You cannot change any aspect (any member) of what `this` points to in this method
  - Remember you can add `const` but you cannot take it away

Friend Functions

friend functions

```
string clk_to_string(const Clock &c) {
  ostringstream oss;
  oss << “Hours:” << c.hours_ << “, Minutes:”
  << c.minutes_ << “, Period:” << c.period_;
  return oss.str();
}
```

- Two choices
  - Rewrite function using accessors
  - Make function a friend

friends

- A `friend` function is a regular function that still has access to `private` member stuff
  - Calling a `friend` is like calling a regular function
    - No `this` pointer
    - Must pass class instance if you need it

Friend in class header

- You must “declare” the function as a friend in the class header:
  - The class gives friendship to the function, not the other way around
  - You must still declare / define the function, the friend designation is an access specification only

Can do vs should do

```
int hours() const { return hours_; }
void hours(int val) { hours_ = val; }
int minutes() const { return minutes_; }
void minutes(int val) { minutes_ = val; }
string period() const { return period_; }
void period(string s) { period_ = s; }
```

- Getters are fine (if you want user to have access)
- What about setters?

Setters should be more complex

- With setters you have an opportunity to do some sanity checking
  - hours < 12, minutes < 60, period equal to “AM” or “PM”
  - providing the interface we have just makes “public” access of structs more complicated
    - No real value added!

Who are your friends?

- If you are not careful, over use of friend turns into a kind of cop out
  - The heck with all this access control stuff, I need to get work done!
- You have to buy into the process that C++ provides

Design Decisions

- Now we are getting to the good stuff
  - As class designers, we are trying to make good decisions, especially when considering access vs complexity
  - We want to design our class to be like the picture
    - Easy to access
    - Functional, updateable, testable, portable

A proposal

- When doing “Clock” thing we could:
  - Indicate errors when the users screws up
  - “Fix it” for them so that it makes sense
- Both have their advantages.
- We’ll do the “fix it”, not because it is necessarily better but because it shows off some more programming

```
class Clock {
private:
      int minutes_;
      int hours_;
      string period_;
      void adjust_clock(int, int, string);
public:
      // Constructors
      Clock() : hours_(0), minutes_(0), period_("") {};
      Clock(int m, int h, string s) : minutes_(m), hours_(h), period_(s) {};
      Clock(string s);
      // Getters
      int hours() const { return hours_; }
      int minutes() const { return minutes_; }
      string period() const { return period_; }
      // Setters
      void hours(int val);
      void minutes(int val);
      void period(string s);
      // member function
      void add_minutes(int);
      // friend function
      friend string clk_to_string(const Clock &);
}
```

private members

Leave the getters, do more for the setters

```
void Clock::adjust_clock(int mins, int hrs, string prd) {
    minutes_ = minutes_ + mins;
    int hrs_remainder = minutes_ / 60;
    minutes_ %= 60;

    hours_ = hours_ + hrs + hrs_remainder;
    hours_ %= 12;
    if (prd != "AM" && prd != "PM")
        period_ = "AM";
    else
        period_ = prd;
}
```

Remember, trailing underscore indicates data member

Made some decisions here. Are they the right ones?

```
void Clock::Clock(int mins, int hrs, string prd) {
    minutes_ = 0;
    hours_ = 0;
    period_ = “”;
    adjust_clock(mins, hrs, prd);
}
void Clock::hours(int val) {
    adjust_clock(0, val, period_);
}
```

Call the this pointer on a Clock member

- After setting, need to call the `adjust_clock` member function
- On what object
  - The calling object
  - `this` pointer
- How to do
  - `this->adjust_clock()`
  - `(*this).adjust_clock()`
  - `adjust_clock()` // Easiest

Overloaded Operators

Operators

- When you define a class, you can also define **overloaded operators**
  - Allows for both unary and binary operators
  - Can be sensible for a class, but be aware of the issues

Operators

+	-	*	/	%	^
&	|	~	~	,	=
<	>	<=	>=	++	
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Really operators are just “syntactic sugar” for a function call

```
Clock c1, c2, c_sum;
c_sum = c1 + c2;
```

```
c_sum = c1.operator+(c2); // if member
c_sum = operator+(c1, c2); // if function
```

- Depends is `operator+` a member function or not?

Rules

- Assign (=), subscript([]), call (), member(->) required to be members
- Compound assign should be members
  - Anything that changes object state
- Commutative (symmetric) operators should be functions
- I/O should be functions

Overloading << or >>

- Imagine we want to use the typical cout statement with our new class
- The name of the operator would be operator<<
- Should it be a method or a function?

Overloading << or >>

```
cout << my_clock << “ right now” << endl;
```

- method: `cout.operator<<(Clock)`
- function: `operator<<(ostream, Clock)`

Function

- We cannot, should not, access the ostream class to add our class as a method.
- Needs to be a function
  - Need to pass the ostream by reference
- What does it return?

Return?

- `cout << my_clock << " right now." << endl;`

- Goes by pairs.
- First `cout << my_clock`
- Should return an `ostream` so the next call works
- `(cout << my_clock) << " right now."`

```
// Overload output
ostream & operator<<(ostream &out, const Clock &c) {
    out << “Hours:” << c.hours_ << “, Minutes:”
        << c.minutes_ << “, Period:” << c.period_;
    return out;
}
```

- Couple things to note:
  - It isn’t a member function
    - How can you tell?
    - Why is that?
  - It returns the stream
    - How?
    - Why?

A friend

- Since it is tied closely to the class, it is a legitimate friend and thus has access to the private data members.

```
Clock operator+(const Clock &c1, const Clock &c2) {
    Clock new_c;
    new_c.minutes_ = c1.minutes_ + c2.minutes_;
    new_c.hours = c1.hours + c2.hours_;
    new_c.period_ = c1.period_;
    new_c.adjust_clock(0, 0, c1.period_);
    return new_c;
}
```

- It isn’t a member function
  - How can you tell?
  - Why is that? (hard question)

Rule of Three

Destructor

- If you can construct a class, can insert the class designer’s will on the creation of variables of the class type, you should also be able to insert your approach to destruction

When something is destroyed

- Variable goes out of scope
- Elements of a container when the container is destroyed
- Dynamically allocated objects when `delete` is called (Covered Later!)

Destructor = tilde (~) + Class Name

- ~Clock()
- The name of the destructor is the same as the name of the class, prepended with the tilde character
- Like a constructor, if you don’t provide one, it is automatically provided

No reason until dynamic memory

- No reason to define a destructor for stuff you would typically do
  - Built in types are “destroyed” correctly
  - STL types are also destroyed correctly
- However, dynamic memory is another issue. We’ll come back to this in a later lecture

Rule of Three

- Used for any object that dynamically allocates memory.
- In this case you probably
  - Define a copy constructor
  - Define an assign operator
  - Define a destructor
- **Rule**: If you need one (**really** need one), then you likely really need all three!

Defaults are fine for non-dynamic memory

- You **do not have** to write any of these member operations
  - If you do not, C++ provides them for you (destructor, copy, assign)
- Unless you are doing dynamic memory, you likely don't **need** this, but you can do it if there is a good reason

=delete

- Like `=default` which sets a method to use the C++ default, you can set a method (like a copy) to be `=delete`, meaning it does not exist and won't run
- In this way you force the user to use either a reference or a pointer

copy constructor (A constructor that takes one parameter, a reference to the class itself)

- C++ by default does mostly the right thing: member to member copy
  - For each data member in the class, a copy is made (calling the copy constructor of that class) to make a copy
  - Except for pointers (copy of a pointer may not be what you want), that is usually good enough

Copy and Assign do pretty much the same thing

- If we want to control how things get copied, then we probably want to control how things get assigned.
  - They pretty much do the same thing
  - They could be exactly the same except for chaining behavior of assign

Clock header (in two parts)

```
class Clock {
private:
     int minutes_;
     int hours_;
     string period_;
     void adjust_clock(int, int, string);
public:
     // constructors, destructor
     Clock() : hours(0), minutes_(0), period("") {};
     Clock(int, int, string);
     Clock(string);
     Clock(const Clock&); // Copy
     ~Clock() {};
     // getters
     int hours() const { return hours_; }
     int minutes() const { return minutes_; }
     string period() const { return period_; }
     // setters
     void minutes(int);
     void hours(int);
     void period(string);
     // members
     void add_minutes(int);
     Clock &operator=(const Clock&);
     friend Clock operator+(const Clock&,
          const Clock&);
};

// Regular functions
Clock operator+(const Clock&, const Clock&);
ostream& operator<<(ostream&, const Clock&);
void split(const string&, vector<string>&,
          char);
```

```
// Copy Constructor
Clock::Clock(const Clock &c) {
    minutes_ = c.minutes_;
    hours_ = c.hours_;
    period_ = c.period_;
}
```

```
// Assignment Operator
Clock& Clock::operator=(const Clock &c)
{
    minutes_ = c.minutes_;
    hours_ = c.hours_;
    period_ = c.period_;
    return *this;
}
```

- Couple things to note:
  - Both pass the parameter (a Clock) by reference
    - Why?
  - The operator= returns a Clock&
    - How to do that?
      - What does return *this do?
      - Why do that?

Spiffy main now

```
int main() {
    Clock my_clk;
    Clock a_clk(2, 2, “AM”);
    Clock some_clk(a_clk); // direct call to copy

    cout << “Copy:” << some_clk << endl;
    my_clk = a_clk = some_clk; // assign op
    cout << “Assign result:” << my_clk << endl; // << op
    my_clk = a_clk + a_clk;
    cout << “Add result:” << my_clk << endl; // add op
}
```