Encapsulation

principle 3
operator overloads
copy, assign, destroy (rule of three)

Encapsulation

Ex 14.6

enforce access

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?

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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 designer provides and, without access, cannot step out of that role

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 only 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.

That's it!! It's why I struggle with struct/class, they are basically the same.

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Structs_vs._Classes

Use a struct only for passive objects that carry data; everything else is a class.

The struct and class keywords behave almost identically in C++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

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 only access to the compiled implementation (no source)
- the class user, because 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 (data or method) 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!

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;
  …
```

OK, constructor is part of the interface

problem 1

problem 2

problem 2 first

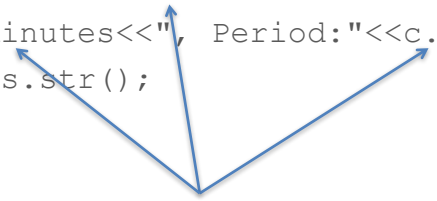
User can no longer access the private members of a class (which includes `hours`, `minutes` and `period`).

What to do? We, class designers, need to provide methods for this:

- win: we control what goes in and out
- loss: we have more work to do

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!

big picture

```
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 &);
};
```

name change

accessors

different names, accessors and data

C++ (for various reasons) does not allow an accessor member function to have the same name as a data member

- changed the data members to have an '_' underline at back
  - Google standard

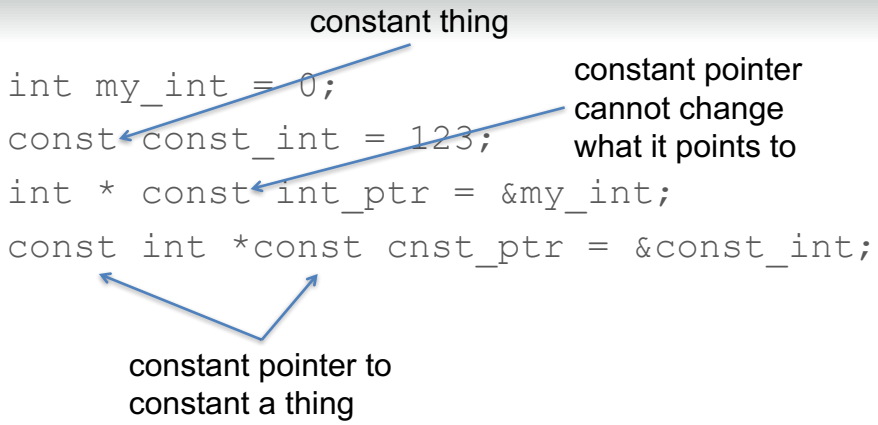
accessor in header

```
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?



the this pointer

The `this` pointer is a constant pointer:

- C++ sets the `this` pointer when a function member 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, that is 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

```
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!

you gotta like this

A `friend` function (`friend` is a keyword) 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.

Class design

is this a good design?

Ex 14.7

friend in class header

You must "declare" the function as a friend in the class header:

- that is, 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, for example:

- hours < 12, minutes < 60, period equal to "AM" or "PM"
- providing the interface we have just makes the "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 opt-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" things, we could:

- indicate errors when the user 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 better but it shows off some programming.

yet another header

```
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, int, string);
  Clock(string s);

  // getters
  int hours() const {return hours_;}
  int minutes() const {return minutes_;}
  string period() const {return period_;}
  // setters, defined elsewhere
  void minutes(int);
  void hours(int);
  void period(string);

  // members
  void add_minutes(int);
  friend string clk_to_string(const Clock &);
};
```

private member!

leave the getters, do more for the setters

```
void Clock::adjust_clock(int mins, int hrs,
                         string prd){
  int hrs_remainder;
  minutes_ = minutes_ + mins;
  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 underline indicates data member. Helpful!

made some decisions here, are they the right ones???

```
Clock::Clock(int mins, int hrs, string prd){
  minutes_=0;
  hours_=0;
  period_="";
  adjust_clock(mins,hrs,prd);
  // this->adjust_clock(mins,hrs,prd);
}

void Clock::hours(int val){
  int temp = hours_ + val;
  adjust_clock(0, temp, "");
  // this->adjust_clock(0, temp, "");
}
```

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

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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

Table 14.1: Operators

Operators That May Be Overloaded

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

Operators That Cannot Be Overloaded

::	.*	.	?:

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

really ops are just "sugar" for 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
- symmetric/commutative should be functions
- I/O should be functions
  - more on I/O in a few slides

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Operator_Overloading

- It can fool our intuition into thinking that expensive operations are cheap, built-in operations.
- It is much harder to find the call sites for overloaded operators. Searching for `Equals()` is much easier than searching for relevant invocations of ==.
- Some operators work on pointers too, making it easy to introduce bugs. `Foo + 4` may do one thing, while `&Foo + 4` does something totally different. The compiler does not complain for either of these, making this very hard to debug.

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?


```
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?


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

Goes as pairs. First

```
cout << my_clock
```

- should return an ostream so the next call works
- `(cout<<my_clock) << "right now"`

a friend

Since it is a function, it is a legitimate friend since we should have access to the private data members.

Destructors

Bhagavad Gita: "Now I am become Death, the destroyer of worlds."

Ex 14.6

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 is something destroyed

- variable goes out of scope
- data members when container is destroyed
- elements of a container when the container is destroyed
- dynamically allocated objects when delete is called (next lecture!)
- when a temp object when expression ends.

name starts with tilde(~)

~Clock()

The name of the destructor is the same as the name of the class, prepended with the tilde character

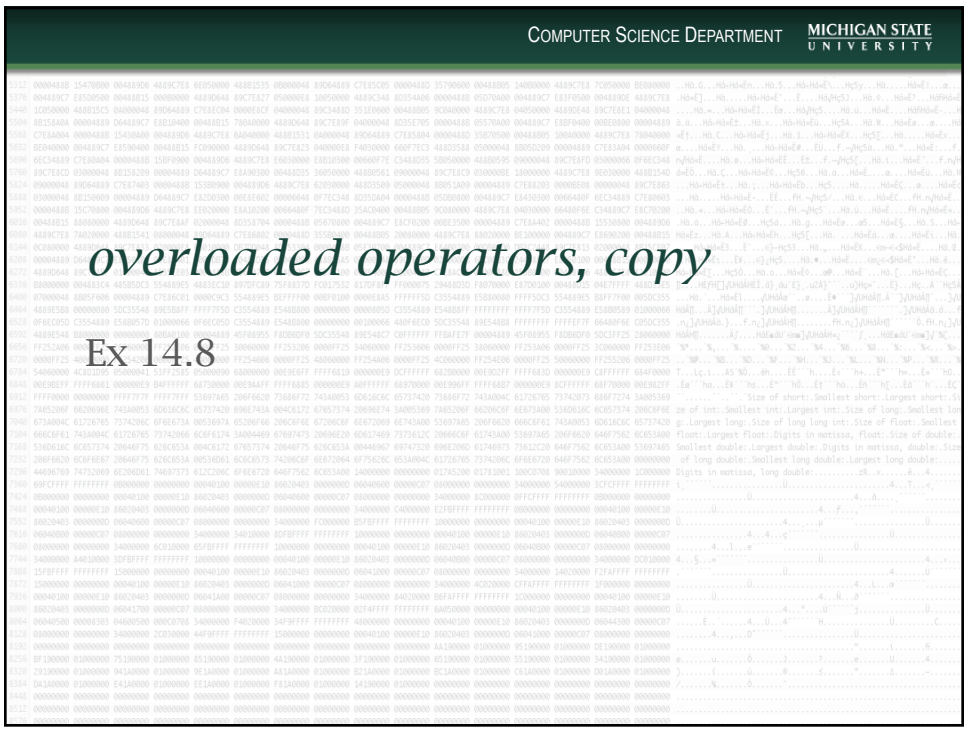
Like a constructor, if you don't define one it is automatically provided

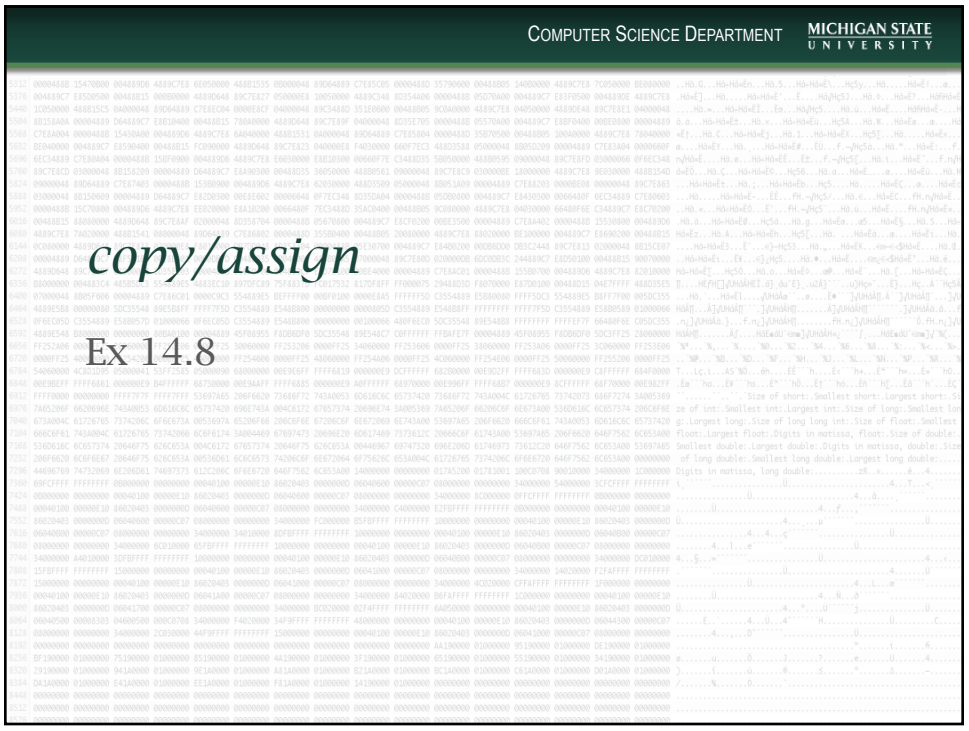
no reason until dynamic memory

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 the next lecture.


COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
overloaded operators, copy
Ex 14.8


COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
copy/assign
Ex 14.8

rule of three

In fact, the rule of three is 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 really need all three!

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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).
  - if you define one, C++ will define the other two (but remember the rule of three)
- Unless you are doing dynamic memory, you 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 of a pointer.

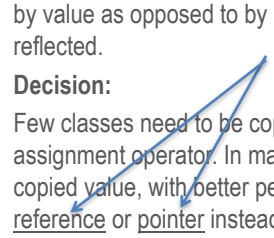
member to member copy

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 if it is STL) to make a copy.
- except for pointers (copy of a pointer may not be what you want) that is good enough!

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Copy_Constructors

- Implicit copying of objects in C++ is a rich source of bugs and of performance problems. It also reduces readability, as it becomes hard to track which objects are being passed around by value as opposed to by reference, and therefore where changes to an object are reflected.
- **Decision:**
- Few classes need to be copyable. Most should have neither a copy constructor nor an assignment operator. In many situations, a pointer or reference will work just as well as a copied value, with better performance. For example, you can pass function parameters by reference or pointer instead of by value, and you can store pointers rather than objects in an STL container.
- If your class needs to be copyable, prefer providing a copy method, such as CopyFrom() orClone(), rather than a copy constructor, because such methods cannot be invoked implicitly. If a copy method is insufficient in your situation (e.g. for performance reasons, or because your class needs to be stored by value in an STL container), provide both a copy constructor and assignment operator.



copy and assign do 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 s);
  Clock(const Clock&); // copy
  ~Clock() {};

  // getters
  int hours() const {return hours_;}
  int minutes() const {return minutes_;}
  string period() const {return period_;}

  // setters, defined in implementation file
   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 &);
   friend ostream &operator<<(ostream&, const Clock&);
};

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

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

```
// assign
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?

```
// 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

```
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,"");
  return new_c;
}
```

To note:

- it is not a member function
  - how can you tell
  - why (hard question)
- return is a local Clock
  - out of scope, how does that work?

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

}
```