Instance variables are unique to an instance of a class.

Static variables are shared by all instance of a class. These don't require an instance of the class to be used.

Static member variables are shared by all classes. Static member variables are stored separately from instances of a class. These exist even if no instance of a class exists. You access static member variables the same way as instance variables.

```
Joe.GPA = 2.13;

Joe.School = "UC";

Student::School = "University of Cincinnati";
```

Static member functions - like static member variables. These are functions that are shared by all instances and don't require an instance to be called. Because they can be called before an instance is created, you can't access instance variables from inside a static member function.

Friends of classes

A friend of a class is a function or class that isn't a member of a given class but still has access to the private parts of that given class.

```
class Student{
        private:
                double GPA;
        public:
        friend string Honors(student &stu);
        friend class Professor;
        static void IsHere(){return true;}
};
string Honors(Student &stu){
        if(stu.GPA > 3.5) return "Highest";
        …
}


Student Joe;
Joe.IsHere();
Student::IsHere();
string s = Honors(Joe);
Joe.Honors(); //not legal
```

Making an entire class a friend should be done with care as all members of the class now have access to the friend's data.

Checkpoint page 829
14.1 – static doesn't require an instance to run. Instance member functions has access to information about that specific instance.
14.2 – Usually outside the class in your .cpp file.
14.3 – before.
14.4 – Can't access instance variables. Can't access non-static member functions
14.5 – call function without an instance of the class.
14.6 – no.
14.7 – Only in X

Memberwise assignment.
If we want to copy an instance of a class to another instance of a class, we can't use a single assignment operator.

```
Student Joe, Mary;
Joe = Mary; //This will not copy all of Mary's values to Joe's instance.
```

Instead, we have to assign them member by member.

```
Joe.Name = Mary.Name;
Joe.GPA = Mary.GPA;
```

A copy constructor can be created to allow assignments with a single statement.
Copy constructor is a special constructor that is called with assgnments like Joe = Mary;.
This is helpful as it allows you to pick and choose which member to assign.

```
className (className &varName){}

Student(Student &stu){…}
```

To make sure your copy constructor doesn't update the original, you can use the const keyword.

```
Student(const Student &stu){…}
```

Operator overloading allows you to redefine how an operator works. For example, if you have a class that handles speed and a function that adds to the current speed, you might say:

```
Speed s;
s.Add(12);
```

With an operator overload, we can instead say:

```
s = s + 12;
```

To overload an operator the syntax is:

```
returnType operator symbol(rightSideParameters)
void operator +(int speed);
```

You can also overload the = to change how the memberwise assignment works.

You can call the overloaded operator with a regular function call:

```
s.operator+(6);
```

This pointer – "this" is a special built in pointer available to a class's member functions. "this" refers to the current instance that is being used. It is passed as a hidden argument in all non-static member functions.

```
Speed Speed::Add(int i){
        this.velocity +=i;
```

```
}
```

An overloaded operator can't change the number operands it uses.

=, +, -, *, etc… will always have 2 operands.

++, --, +, - will always have 1 operand. No parameter is required for pre but post uses a dummy parameter.

The object on the left side of the operator is the object who's operator overload is called.

The two objects don't need to be the same type.

```
Pizza p;
Topping t;
p += t;
```

Checkpoint

14.14 –

```
void pet::operator =(pet other);
```

14.15 –

```
dog.operator =(cat);
```

14.16 – you can't chain the equal signs together.

```
a=b=c=d; //This is where they are chained.
```

14.17 – to access fields with the same name as a class parameter function.

14.18 – non-static member functions

14.19 – cat calls + with tiger as a parameter.

14.20 – post fix operation such as i++

Object conversion

For built in data types, conversion happens automatically.

You can overload a class to do something similar.

```
class MyTime{
        private: int hours, int minutes;
        public:
                operator int(); //overload the equal when left side is an int.
};
MyTime::operator int(){
        int result = hours * 60;
        result += minutes;
        return result;
}

MyTime race1;
…
int totalTime = race1; //call the overload int conversion.
```

Note: conversion overloads don't specify parameters and don't specify overloads.