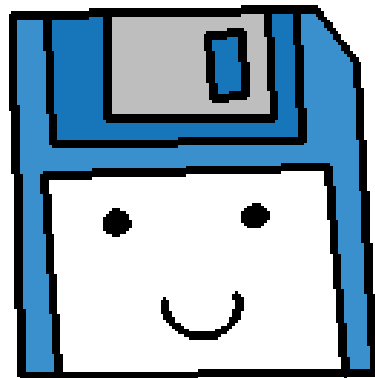


Classes

Part 2

We've covered some basics of classes,
But let's also talk about some more
intermediate topics.

There are a lot of features for classes in
programming, so you need to get a good
handle of these topics.



Classes are the backbone
of languages like C++, C#,
and Java.

Constructors & Destructors



Constructors & Destructors

Constructors and Destructors are a special type of **function** that classes can have.

Constructors are run automatically and immediately once you **instantiate** (declare) a variable of that object type.

Destructors are run automatically and immediately once that variable goes out of scope – in other words, just before it gets **destroyed**.

These are good for **initializing** and **de-initializing** our objects.

Constructors & Destructors

```
struct Timestamp
{
    int h, m, s;
};
```

Remember how uninitialized variables may contain garbage? There is no default value for variables in C++.

If your class contains ints, floats, chars, or other data types as member variables, we don't know what they will store when the object is first created.

```
class Fraction
{
    private:
        int numerator, denominator;

    public:
        // Don't forget to call setup
        // immediately after creation!!
        void Setup( int n, int d )
        {
            numerator = n;
            denominator = d;
        }
}
```

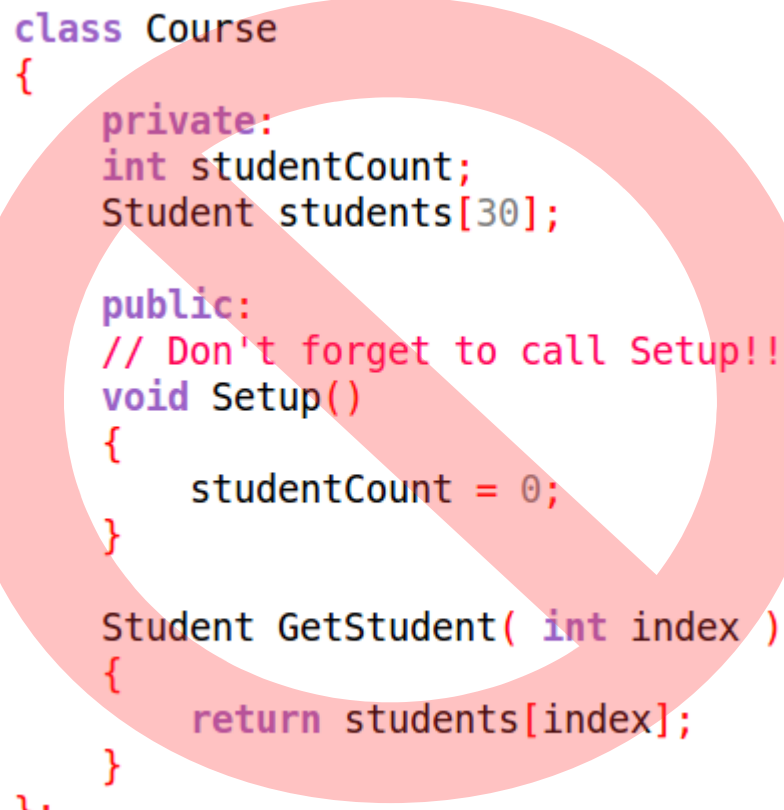
When we declare our object, we have to **remember** to initialize all its internal member variables.

Relying on “remembering” is a good way to get bugs, because somebody is going to eventually forget!

Constructors & Destructors

Using constructors means that we
DO NOT HAVE TO REMEMBER!

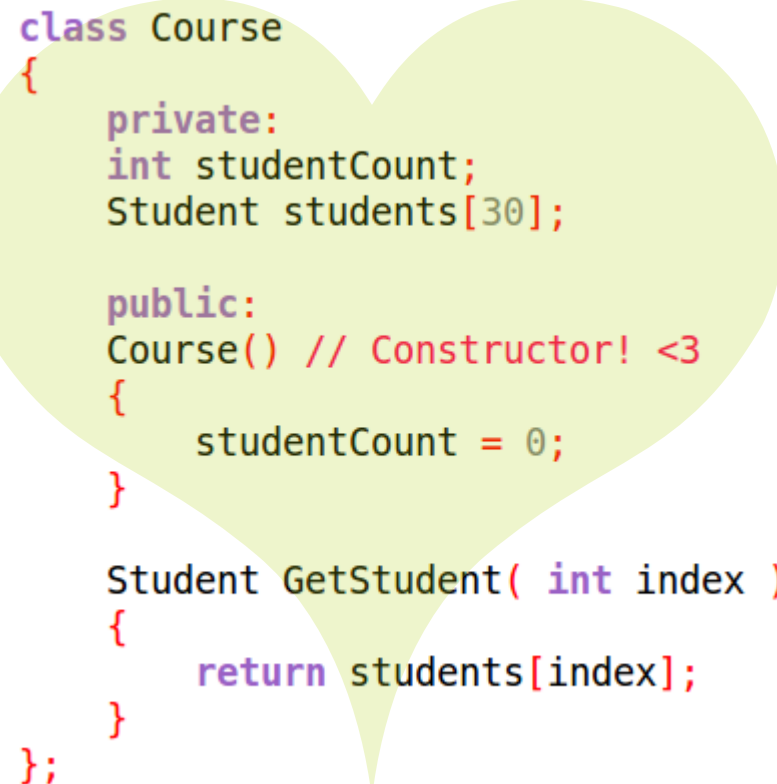
We can make sure the class initializes its member variables to some “safe” value...



```
class Course
{
    private:
    int studentCount;
    Student students[30];

    public:
    // Don't forget to call Setup!!
    void Setup()
    {
        studentCount = 0;
    }

    Student GetStudent( int index )
    {
        return students[index];
    }
};
```



```
class Course
{
    private:
    int studentCount;
    Student students[30];

    public:
    Course() // Constructor! <3
    {
        studentCount = 0;
    }

    Student GetStudent( int index )
    {
        return students[index];
    }
};
```

Constructors & Destructors

```
class Course
{
    private:
        int studentCount;
        Student students[30];

    public:
        // Don't forget to call Setup!!
        void Setup()
        {
            studentCount = 0;
        }
}
```

Relying on the user, another programmer, or even yourself to call the **Setup()** function after creating an object is very unreliable.

With this Course class, if **studentCount** is not initialized, it may store a garbage value (any number!), which means when we try to access the array, we might go outside of its declared size!

Constructors & Destructors

```
class Course
{
    private:
        int studentCount;
        Student students[30];

    public:
        // Don't forget to call Setup!!
        void Setup()
        {
            studentCount = 0;
        }
}
```

Relying on the user, another programmer, or even yourself to call the **Setup()** function after creating an object is very unreliable.

With this Course class, if **studentCount** is not initialized, it may store a garbage value (any number!), which means when we try to access the array, we might go outside of its declared size!

```
class Course
{
    private:
        int studentCount;
        Student students[30];

    public:
        Course() // Constructor! <3
        {
            studentCount = 0;
        }
}
```

The constructor is called **automatically**, so any member variables that might cause problems can be initialized right off the bat!

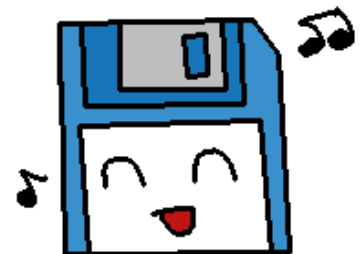
Constructors & Destructors

Additionally, you can also **overload your constructors** with different parameters, to initialize your object differently based on what information is available when it is declared...

```
class Assignment
{
    private:
        float totalPoints;
        float receivedPoints;
        float weight;

    public:
        Assignment()
        {
            totalPoints = 100;
            receivedPoints = 0;
            weight = 0.20;
        }

        Assignment( float totalPoints, float weight )
        {
            this->totalPoints = totalPoints;
            this->weight = weight;
            receivedPoints = 0;
        }
};
```



Constructors & Destructors

```
class Assignment  
{
```

```
private:
```

```
float totalPoints;
```

```
float receivedPoints;
```

```
float weight;
```

```
public:
```

```
Assignment()  
{
```

```
totalPoints = 100;
```

```
receivedPoints = 0;
```

```
weight = 0.20;
```

```
}
```

```
Assignment( float totalPoints, float weight )  
{
```

```
this->totalPoints = totalPoints;
```

```
this->weight = weight;
```

```
receivedPoints = 0;
```

```
}
```

```
};
```

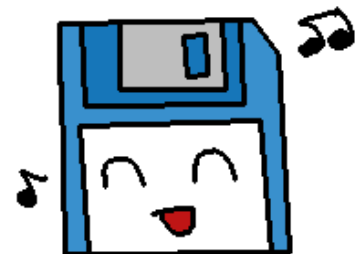
```
int main()  
{
```

```
Assignment assignment1;
```

```
Assignment assignment2( 100, 0.30 );
```

```
return 0;  
}
```

If a variable of this type **Assignment** is declared like `assignment1`, the constructor with no parameters is called, and everything receives a default value.



Constructors & Destructors

```
class Assignment
{
    private:
        float totalPoints;
        float receivedPoints;
        float weight;
```

```
    public:
        Assignment()
        {
            totalPoints = 100;
            receivedPoints = 0;
            weight = 0.20;
        }
```

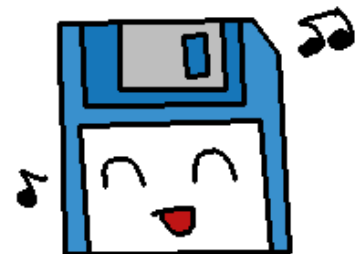
```
    Assignment( float totalPoints, float weight )
    {
        this->totalPoints = totalPoints;
        this->weight = weight;
        receivedPoints = 0;
    }
};
```

```
int main()
{
    Assignment assignment1;

    Assignment assignment2( 100, 0.30 );

    return 0;
}
```

If we declare a variable of type **Assignment**, but pass in arguments, the constructor with the matching argument types will be called instead.



Constructors & Destructors

```
class Assignment
{
    private:
        float totalPoints;
        float receivedPoints;
        float weight;

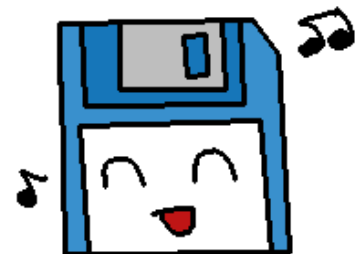
    public:
        Assignment()
        {
            totalPoints = 0;
            receivedPoints = 0;
            weight = 0;
        }

        Assignment( float totalPoints, float weight )
        {
            this->totalPoints = totalPoints;
            this->weight = weight;
            receivedPoints = 0;
        }
};
```

Note that if we had
Assignment(float totalPoints, float weight)

But not
Assignment()...

When we declare our object, it **must** have parameters, and would not accept a declaration without the parameters.



Constructors & Destructors

With **Destructors**, there can only be one – it does not take any parameters like a constructor can.

Destructors are called when the object variable goes out of scope and is destroyed.

This is especially useful once we get into memory management.

Constructors & Destructors

```
class FileWriter
{
    private:
        ofstream outfile;

    public:
        FileWriter( string filename ) // Constructor
        {
            outfile.open( filename.c_str() );
        }

        ~FileWriter() // Destructor
        {
            outfile.close();
        }

        void WriteText( string text )
        {
            outfile << text << endl;
        }
};
```

A good example of something we would want to **clean up** before an object is destroyed would be File I/O!

With this class, it requires that you pass in a file name to open a file when the object is created.

When the object is destroyed, it closes the output file stream automatically, so we don't have to worry about it.

If we had forgot to close the file before the program quit, we could get corrupted data in our file.

Constructors & Destructors

```
class FileWriter
{
    private:
        ofstream outfile;

    public:
        FileWriter( string filename ) // Constructor
        {
            outfile.open( filename.c_str() );
        }

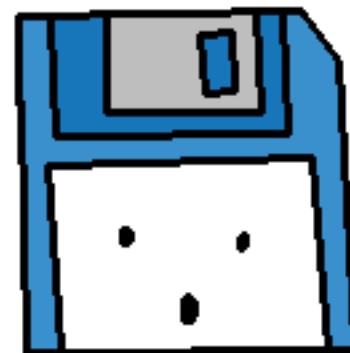
        ~FileWriter() // Destructor
        {
            outfile.close();
        }

        void WriteText( string text )
        {
            outfile << text << endl;
        }
};
```

The constructor **always** has the name of the object, no “return type”, and it may or may not have a parameter list. (Still needs parenthesis, though)

The destructor **always** has the name of the object, but with a tilde (~) before-hand. Its parameter list is **always** empty.

Static Members

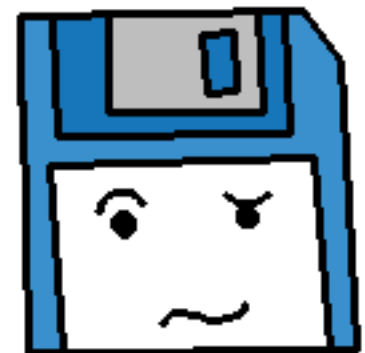


Static Members

Static members are an interesting feature of classes.

I had a hard time grasping them at first, but these days I use them relatively frequently for a very specific type of class.

It's up to you to use static members in a class or not, but let's go over what they are, and some uses for them.



Static Members

With normal member variables, every time you **instantiate** (create) a variable of that object type, it gets its own set of those variables.



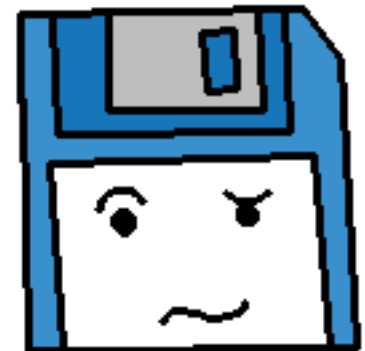
```
Rectangle rectA;  
Rectangle rectB;
```

```
rectA.x = 50;  
rectB.x = 30;
```

But if you declared a member variable as **static**, that variable would be shared among ALL instantiations of that object.

This might be useful for keeping count for all instances of the object,

Or something else that they all have in common.



Static Members

OK, stay with me
for a second.

Let's say that we're writing a class
that “wraps” the file stream object
from the standard library.

We want to take advantage of
constructors and **destructors** to
make sure the file is opened and
closed safely, automatically.

```
class File
{
public:
    File( const string& filename )
    {
        output.open( filename.c_str() );
    }

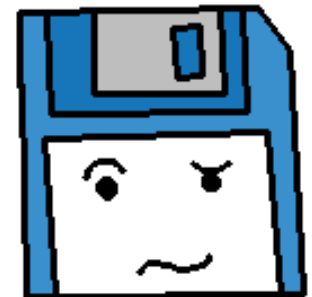
    ~File()
    {
        output.close();
    }

private:
    ofstream output;
};
```

File opened when
object is created

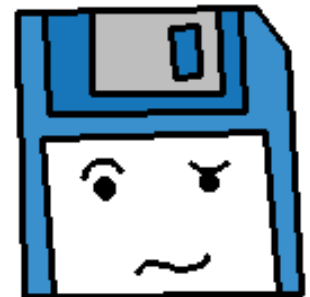
File closed when
object is destroyed

This object “wraps” **ofstream** to
add functionality.



Static Members

Show
programming



Static Members

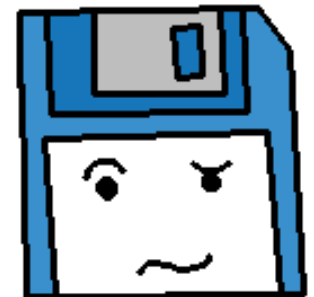
```
class OutputFile
{
    private:
        ofstream file;
        static int fileCount;

    // ...
}
```

So, we created a **static** member variable by adding the **static** keyword before the variable declaration.

Any time **fileCount** is accessed within any **OutputFile** function call, the values will be the same.

We initialize it at 0, and any time we add or subtract, that value is shared between all **OutputFiles**.



Static Members

```
class OutputFile
```

```
{
```

```
// ...
```

```
static int GetFileCount()
```

```
{
```

```
    return fileCount;
```

```
}
```

```
private:
```

```
ofstream file;
```

```
static int fileCount;
```

```
// These ought to be private so the  
// user isn't calling these.
```

```
void IncFileCount()
```

```
{
```

```
    fileCount++;
```

```
}
```

```
void DecFileCount()
```

```
{
```

```
    fileCount--;
```

```
}
```

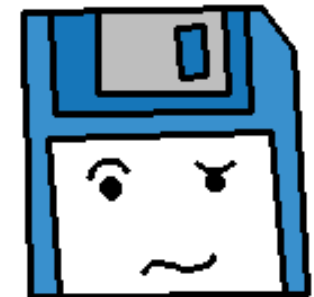
```
};
```

We created some **static** functions as well.

Though I've adjusted this to be designed better.

If we want to call a function without creating any instances of this class, we want to make it static.

Otherwise, normal functions will do. These two are only accessed within OutputFile, so they've been made private and not-static.

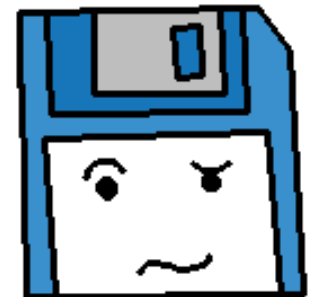


Static Members

With static members, we also have to **define** the variable, in addition to **declaring** it.

Outside of the class, we define it by specifying the
Data type,
Class it belongs to,
And **variable name**.

We can also initialize it here.



```
5  class OutputFile
6  {
7      public:
8          OutputFile( const string& filename )
9          {
10
11
12
13
14          ~OutputFile()
15          {
16
17
18
19
20          void WriteText( const string& text )
21          {
22
23
24
25          static int GetFileCount()
26          {
27
28
29
30      private:
31
32          void IncFileCount()
33          {
34
35
36
37
38          void DecFileCount()
39          {
40
41
42
43
44
45
46
47          ofstream file;
48          // Declaration
49          static int fileCount;
50  };
51
52  // Definition
53  int OutputFile::fileCount = 0;
```


Static Members

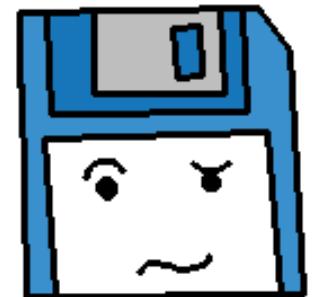
When accessing a static member, we can either access it via the object name directly:

```
cout << " Files: " << OutputFile::GetFileCount() << endl;
```

Or we can access it via an instantiation of the object:

```
OutputFile programLog( "Log.txt" );  
cout << " Files: " << programLog.GetFileCount() << endl;
```

This function returns the value of the **static int fileCount**,
So calling it either way will give us the appropriate value.

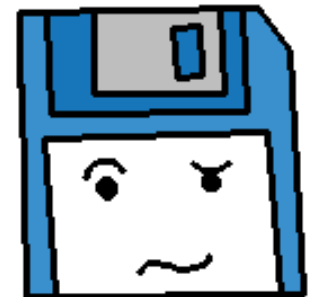


Static Members

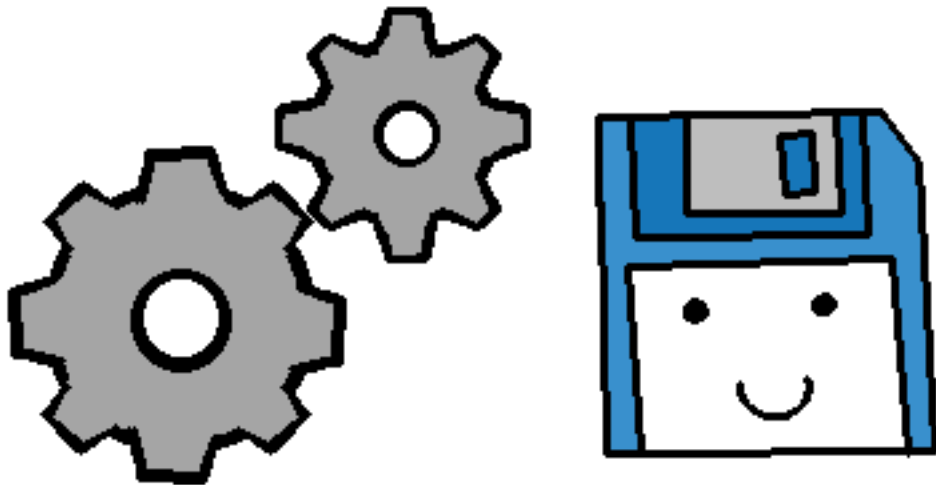
It is good to know about static members,
Even if they don't make complete sense right now.

As with anything, it takes practice.

And if you encounter a **static** member in somebody else's
code, you have a rough idea of how it works.



Const Functions



Const Functions

We have used the **const** keyword in the past to create a variable flag that doesn't change, as well as to create function parameters that are guaranteed to not change.

We can also ensure that an object's member function **cannot change the value of member variables** by marking the function as **const**.

Const Functions

```
class Rectangle
{
    private:
    int x, y, width, height;

    public:
    // This const function is now guaranteed
    // to not be able to change any member variables
    // of this class.
    void Display() const ←
    {
        cout << "(" << x << ", " << y << ")\t"
              << width << " x " << height << endl;
    }

    void SetX( int val )    { x = val; }
    void SetY( int val )    { y = val; }
    void SetWidth( int val ) { width = val; }
    void SetHeight( int val ) { height = val; }
};
```

If we put **const** at the end of our **function signature**, this means that **within the function**, no **members of the class** will change their values.

Const Functions

Similar to passing values by reference but not wanting those values to change, **const** lets us write functions that operate on some data, but does not “corrupt” that data in the process.

```
13
14
15 void Display() const
16 {
17     cout << "(" << x << ", " << y << ")\t"
18         << width << " x " << height << endl;
19     x = y = 10;
20 }
```

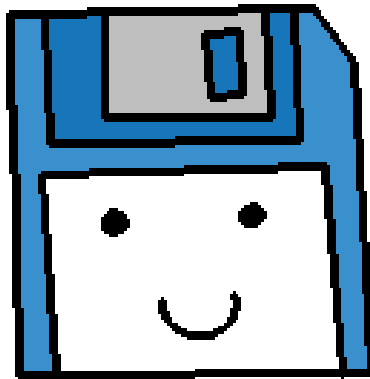
Trying to make a modification within a const function will result in a **build error**.

It is always easier to fix a **build error** than a **logic error**!

File	Line	Message
=== Build: Debug in Const Member Function (compiler: GNU GCC Compiler) ===		
/home/rejcx/PROJE...		In member function 'void Rectangle::Display() const':
/home/rejcx/PROJE... 17		error: assignment of member 'Rectangle::y' in read-only object
/home/rejcx/PROJE... 17		error: assignment of member 'Rectangle::x' in read-only object
=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===		

How are you doing with classes?

We've covered a lot of important topics!



How about this time, I give you
some practice problems to solve?

You'll find the solutions in the
course repository.

Class “Homework”

#1

Write a program that has a **struct** called **CoordPair**.

CoordPair has the following member variables:

- `x` and `y`, both floats.

And the following member functions:

- `GetUserInput()`, return type `void`, which has the user input values for the `x` and `y` member variables.
- `Display()`, return type `void`, which displays the `x` and `y` values as a coordinate pair, like this: `(2, 3)`

Test it out by creating some `CoordPair` objects within `main()`, having the user enter both, and finding the slope between these coordinate pairs.

Remember that structs make their members **public** by default!

Class “Homework”

#2

Write a program that has a **class** named **Student**.

Member variables (private):

- **name**, type string
- **classNames**, array of size 3, strings

Member functions (public):

- **void SetName(const string& value)** - set the name member to the value passed in.
- **void SetClassName(int index, const string& value)** - set the name of the class at index *index* to the value passed in.
- **void DisplayInfo()** - Display the student name and all 3 class names.

In main, create a few students, set their names, and all of their classes. Then use `DisplayInfo()` to view all the information.

Class “Homework”

#3

Write a program that has a class called MyClass. It has the following public members:

- **MyClass()** constructor – output the text “Constructor”
- **MyClass(const string& value)** constructor – Output the text “Constructor”, then the value of the parameter.
- **~MyClass()** destructor – Output the text “Destructor”

```
int main()
{
    MyClass classA;

    if ( true )
    {
        MyClass classB( "B!" );
        MyClass classC;
    }

    return 0;
}
```

In main, create a series of MyClass objects to see the constructors and destructors working.