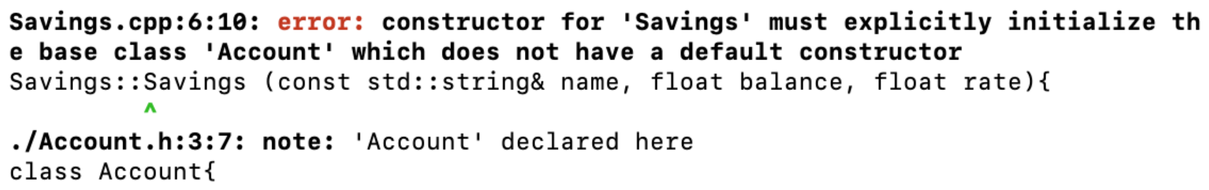
ERROR MESSAGES and Lecture NOTES

Lecture 100: Account and Savings Classes



```
Savings.cpp:6:10: error: constructor for 'Savings' must explicitly initialize th
e base class 'Account' which does not have a default constructor
Savings::Savings (const std::string& name, float balance, float rate){
         ^
./Account.h:3:7: note: 'Account' declared here
class Account{
```

This means that we are not invoking the Account class constructor from the child class. If you don't specify a constructor to be used, it is going to try to use the Default constructor of the Account class (base class), that doesn't exist after we've defined the parameterized constructor.

Lecture 101: Assignment I – Implement the Checkings class.

Lecture 102: Project (Part III) – Inheriting Constructors

In C++, there are 3 member functions that are **NOT inherited:**

1) Constructors
2) Destructors
3) Assignment Operator.

So that's why we have to implement the constructors and destructors manually in Checkings and Savings Account.

In C++11, there is a feature called **Inheriting Constructors.** With this feature, you can inherit the constructor of a base class, so you won't have to manually implement the constructor in the Child.

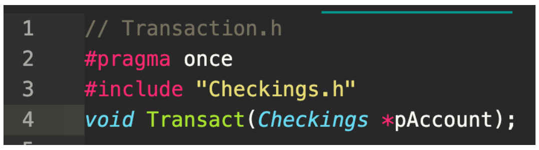
What would happen if we don't implement a constructor in the Checkings class?
In C++ 11, we can inherit the base class constructor so that the compiler will **implicitly generate** the constructors in the child that will call the base and let's first comment this out.

To inherit the constructors from the base, you have to use the 'using' keyword.
`using Base_Class::Constructor` inherits constructors from the base class.

It can be seen from disassembly that the compiler **implicitly generates** the Checkings constructor. And you can see that it makes a **call** to the Account() constructor.

Lecture 103: Project (Part IV) – Virtual Keyword

Assume there is a module in this application that uses these account classes to perform some transactions (Higher level module).



```
// Transaction.h
#pragma once
#include "Checkings.h"
void Transact(Checkings *pAccount);
```

The problem with this implementation is that the transaction can be performed only on Checking account. What if we want to perform transactions on Savings account? We cannot do that. this transaction module is hardcoded to perform transactions only on Checking Accounts because that's what is specified as argument above. If we want to perform transactions on the Savings Account, we cannot do that.

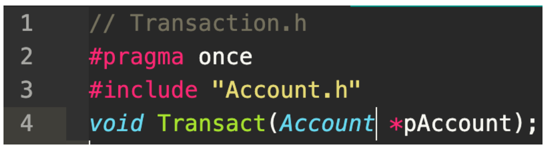
The other thing is that – this transaction module is **tightly coupled with the Checking Account** because it includes the header file of Checkings. **That means if we make any changes to the Checking class, those changes are going to affect the transaction module as well.**

This is an example of a **higher level module depended on the implementation details of the lower level module, which is a violation of the Dependency Inversion Principle.**

That is why it is NOT a good idea to specify a Checking pointer here. So what should it be? We know, in inheritance, the base class represents all its child classes, it will contain the common behavior of all its children in the same way a base class object can point to any of its child classes.

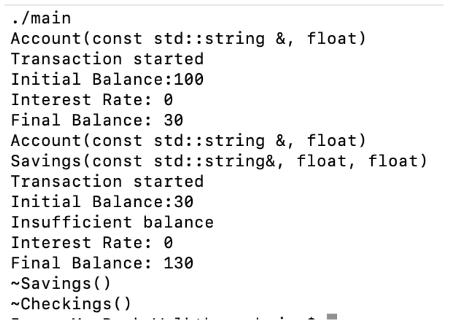
But that object needs to be a pointer or a reference.

This means a base class pointer or a reference can point to any of the child class objects. So that's why, we will replace this Checking * with Account*. So now, this transaction module is dependent ONLY on the base class, it is not dependent on the Child classes. It doesn't know anything about the Child classes.



```cpp
// Transaction.h
#pragma once
#include "Account.h"
void Transact(Account *pAccount);
```

Now, we can go back to main() and we can pass a Savings class object:



```
./main
Account(const std::string &, float)
Transaction started
Initial Balance:100
Interest Rate: 0
Final Balance: 30
Account(const std::string &, float)
Savings(const std::string&, float, float)
Transaction started
Initial Balance:30
Insufficient balance
Interest Rate: 0
Final Balance: 130
~Savings()
~Checkings()
```

This looks wrong – even though this is a savings account, the interest rate is 0, and the final balance doesn't seem to contain any interest accumulated within it.

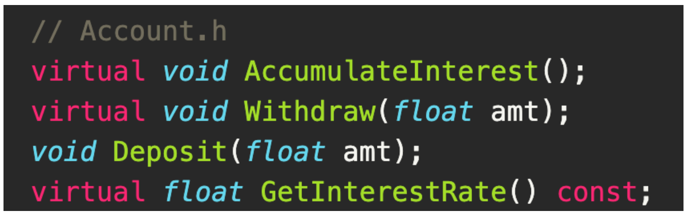
Even though we have implemented the GetInterestRate() in the savings class, the interest rate is still 0, and no interest is calculated.

When we go into Disassembly, we see that the compiler generated a call to AccumulateInterest() of **Account** – while we were expecting it to invoke AccumulateInterest() of **Savings**. In the same way, if we look at assembly code of the GetInterestRate(), the compiler generated a call to GetInterestRate() of **Account** AND NOT of **Savings.** The problem here is when the compiler encounters these function calls, it generates the calls to these functions **based on the type of the pointer.** The compiler doesn't care what the pointer is pointing at.

The Virtual Keyword

To tell the compiler to generate the calls to the object being pointed at, such that the functions are invoked on the real object that the pointer is pointing at, we will have to mark such functions with a special keyword: **virtual.** AccumulateInterest(), Withdraw() and

GetInterestRate() are reimplemented by the child classes, that means the child classes are **overriding** the implementation of these functions of the base. And if these functions are invoked to the base class, pointer or reference, then the correct function should be invoked.



```cpp
// Account.h
virtual void AccumulateInterest();
virtual void Withdraw(float amt);
void Deposit(float amt);
virtual float GetInterestRate() const;
```

Now the compiler will generate the calls to these functions based on the type of the object that the pointer pAccount is pointing at.

BUT, the compiler **will not know AT COMPILE TIME** what object pAccount is pointing at. That is why it will generate special code that will decide **at runtime** which objects function should be invoked.

Now it works: we can go into disassembly to see the magic.

Going into disassembly you don't see the call to AccumulateInterest() anywhere, but instead you see a call on the **eax register.** This means that **eax register** has some address which is invoked. (We'll explain this in the next lecture) But the concept we have implemented here is **POLYMORPHISM.**

Polymorphism: you a send a message to an object that represents different objects and the message will **automatically go to the correct subtype of** that object, and **give the right behaviour.**

So these functions that we marked with virtual, these are **POLYMORPHIC functions** , that means when these functions are invoked, they will be invoked on the correct object that pAccount is pointing at.

The advantage of polymorphism is the code that uses polymorphism **doesn't have to know** the actual object on which the functions are invoked. It only needs to know about the **base. This way, we can add more account classes in the future, and Transact() module will even work with those new Account classes,** because it doesn't know ANYTHING about the child classes at all. This adheres to the **Dependency Inversion Principle.**

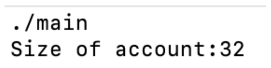
In the next video, we'll examine how the virtual mechanism is internally implemented.

Lecture 105: Virtual Mechanism internals – Part II
We know that when a function is marked as virtual in a class, then a compiler automatically **adds a hidden pointer to the class** and that pointer is called a **virtual pointer.** This **virtual pointer** is going to **increase** the size of the class and the objects **by the size of the pointer.**

- In 32 bit platforms it will increase by 4 bytes.
- In 64 bit platforms it will increase by 8 bytes.

Let's verify that!

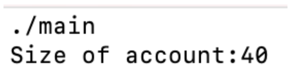
Size of Account without Virtual Functions



```
./main
Size of account:32
```

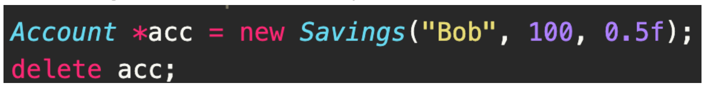
This means my comp has a 64 bit platform.

Size of account with 1 virtual Function

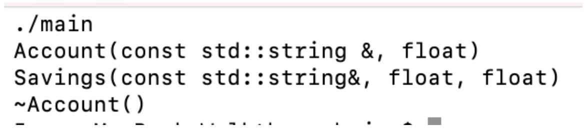


```
./main
Size of account:40
```

Recall that a base class pointer can point to any of its child class pointers – so consider the following code and the output.



```
Account *acc = new Savings("Bob", 100, 0.5f);
delete acc;
```



```
./main
Account(const std::string &, float)
Savings(const std::string&, float, float)
~Account()
```

There is something amiss in the output, because you see 2 constructor calls for both classes in the hierarchy, but you see the destructor call only for the base class Account(). It has not invoked the destructor of the Savings class. **This is because the compiler generates the Constructor / Destructor calls** for this pointer based on its type and this type is 'Account()', but what we want is the call to the destructor should be generated based on the type of the object the Account() pointer is pointing at, and obviously this will be decided at runtime. That is why **the compiler should generate a polymorphic call to the destructor of the object, and for this reason, we need to make the destructor of the base class VIRTUAL as well.**

In the next video, we'll look at some new keywords that C++ 11 introduced which are used with inheritance.

Lecture 106: C++11's Override and Final

Using C++11's FINAL
If a class is NOT MEANT to be inherited, the users of this class can prevent users from inheriting it, and this can be done by marking the class with the FINAL keyword, and if you try to inherit from this class, then that would be a compiler error: **A final class type cannot be used as a base class.**

If you create a base class which is **not meant to be inherited,** then you can mark that class with the **Final** specifier.

```
class Zip final {
…
};// Class body.
```

Using C++11's OVERRIDE
Let's say we have a class called Document which has a virtual function called Serialise(), and different subclasses would have their own definition of serialise(). For inheriting from the base class, **the function signature must be exactly the same.**

If the signature is different, it will invoke the method of the **base class** and **not of the subtype.** This may lead to bugs, and the compiler does not tell you about it.

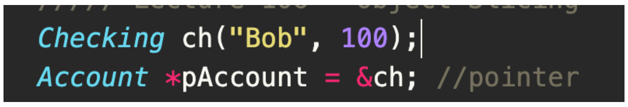
C++11 introduced a new keyword called override, and that keyword is used to **tell the compiler** that a function is being **overridden.** When we use this specifier, the compiler will check whether a function **by the same name AND the same signature** exists in the base or not. In our case, the signature is different. So that's why this **forces** a compiler error. This way we immediately know that something is wrong. If we use the Override keyword, we can avoid some runtime errors. So it is a **good practice** to use the **override** specifier when you override a base class function.

If we use override on a function that **isn't virtual,** the override keyword will give an error as well: Member with override specifier 'override' did not override any base class methods…

If there is a specific member function that you don't want users to override, then you can mark the function as **FINAL.** If you do so, you will get an error: **Cannot override the final function.**

Notice that C++11 has added a lot of new features that help us write more robust code, and it also prevents us from making mistakes. So take advantage of the new C++11 and C++14 features in your code.

Lecture 107: Object Slicing
In this lecture we'll examine **upcasting** and **downcasting.**



```
Checking ch("Bob", 100);
Account *pAccount = &ch; //pointer
```

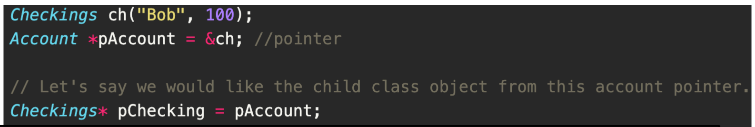
If I simply assign the Checking object to Account object, the Checking object will get automatically converted or upcasted to account object. **The compiler will perform object slicing to slice the child class object so that it fits into the base class object.**

What is Object Slicing?

- A situation in which the compiler will **deliberately remove** some part of an object.
- This occurs **when a child class object** is assigned directly to a **concrete base class object** OR you may pass a Child class object **by value** into a function that accepts a base class object.
- Since the child class may contain more attributes, it may require more memory, so its size will be larger than its base, that is why when it is assigned to the base object, some part of the memory after the base object may get overwritten.
- This will lead to memory corruption.
- In order to prevent this, the **compiler slices the child object, effectively removing the memory that contains the child class attributes.**
- Then, the child class object is then copied into the base object.

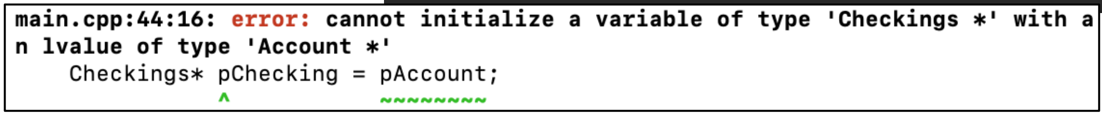
So remember, if you create a base class object **and you want to invoke the child class member functions through that object,** ensure that **it is either a pointer or a reference!**

After doing the upcast, if we want to get the child class object back from this Account() pointer, so let's create a pointer, and then assign the Account pointer to it: but this does not compile and there is an error. The reason for the error is this pointer represents an **Account**



```
Checkings ch("Bob", 100);
Account *pAccount = &ch; //pointer

// Let's say we would like the child class object from this account pointer.
Checkings* pChecking = pAccount;
```



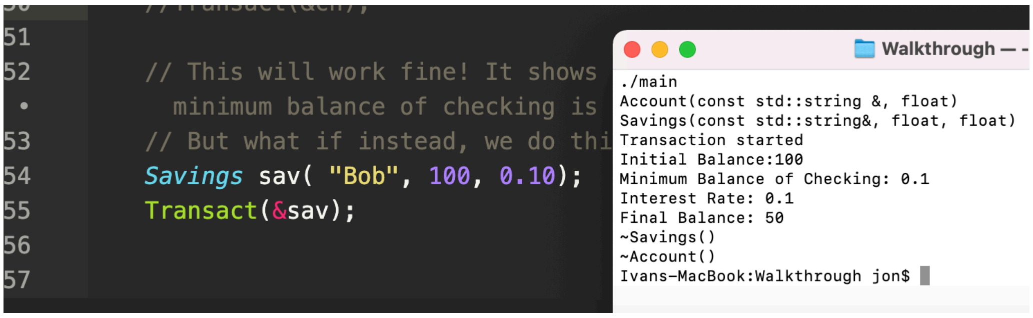
```
main.cpp:44:16: error: cannot initialize a variable of type 'Checkings *' with a
n lvalue of type 'Account *'
    Checkings* pChecking = pAccount;
               ^           ~~~~~~~~
```

and not all Accounts can be Checking Accounts! So the compiler will not allow you to assign the Checkings* pointer to this pointer. To do so anyway, we would have to manually specify a cast like this:
Checkings * pchecking = static_cast<Checkings*> (pAccount);

And this is known as **downcasting.** So we have **downcasted** a base class pointer to a child. So remember, **upcasting does not require any program or intervention,** it happens automatically. **Downcasting** on the other hand, will require you to perform manually specify the cast.

Lecture 108: TypeId Operator

In this video we'll stat the discussion on RTTI. Remember in Checkings Account, the balance should not fall below 50, but **we don't want to hardcode this value in the class. We want this to be set from outside.** We also want to **query for this value outside.**



```
    // This will work fine! It shows
      minimum balance of checking is
    // But what if instead, we do thi
    Savings sav( "Bob", 100, 0.10);
    Transact(&sav);
```

```
./main
Account(const std::string &, float)
Savings(const std::string&, float, float)
Transaction started
Initial Balance:100
Minimum Balance of Checking: 0.1
Interest Rate: 0.1
Final Balance: 50
~Savings()
~Account()
Ivans-MacBook:Walkthrough jon$
```

It shows the Minimum Balance of checking to be the third parameter – **which is incorrect.** The reason being we are **blindly downcasting from an Account pointer to a Checking pointer,** the account pointer may not even be pointing to Checking, it may be pointing to some other Child class of Account.

So even though the typecast may work, **you will not get the correct result.** In other cases, it may even crash. That is why before we downcast, the Account pointer to checking, we should first check whether it is actually a Checking object or not.

This is where the RTTI concept of C++ will be useful.

Some information regarding Type ID:

- If you use the information on non-polymorphic types, then it gathers type information at **compile time.**
- If you use it on polymorphic types, like in our case, we are using it on the account ptr, then typeID will work at **runtime.**
- It makes sense to use it only on **polymorphic types.**
- Maybe just getting the name of the type is not enough, maybe we want to take some decision based on the type of object that p is pointing at.

In this case, use comparison operators to check for equality.

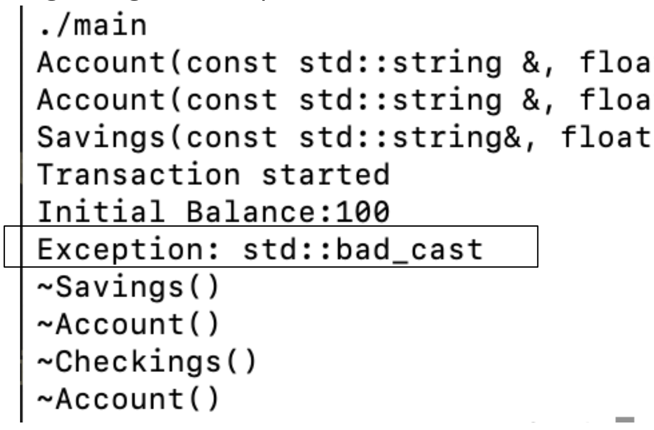
Lecture 109: Dynamic Casting

Instead of using **typeid** to check for the type and then performing the typecast, we could have used the **dynamic casting operator.**

- Dynamic Cast will check whether the typecast can be performed or not:
  1. If the typecast can be performed, it will return the type casted pointer.
  2. Otherwise, it will return nullptr.

- Dynamic Cast can also work with references. We can make an overload of the Transact() function to show that the function can work by reference.
- If dyanamic cast is **unable to perform a cast, AND unable to return NULL,** dynamic cast **throws an exception.** The exception is called " Bad Cast". We would have to create a try-catch block for this – within the 'Try' block, we write code that can throw an exception at runtime and immediately following the try block we have a catch block and the catch block always accepts an **exception type.**

- Now, in this case, I'm using **std::exception** as a type which is the base class of all standard exceptions. The contains a method called 'what()'' that contains a message regarding the exception.



```
./main
Account(const std::string &, float)
Account(const std::string &, float)
Savings(const std::string&, float, float)
Transaction started
Initial Balance:100
Exception: std::bad_cast
~Savings()
~Account()
~Checkings()
~Account()
```

Checkings and Savings Types are **not compatible. But they shouldn't be compatible anyway.**

Ideally, we should avoid using dynamic cast **OR** typeid. To use these, compiler had to add some **extra information for the polymorphic classes.**

So during compile time, the compiler creates the type information of the object as a type_info object, and that is stored along with the class **vtable**. At runtime, when we use typeID or dynamic cast, these operators will query that information, so RTTI will **impose overhead on the program.** That is why it should be **avoided,** especially dynamic cast, because dynamic cast will have to run through the hierarchy to check if the cast can be performed or not. So it is xlower than typeid. The bottom line, as far as possible, avoid RTTI.

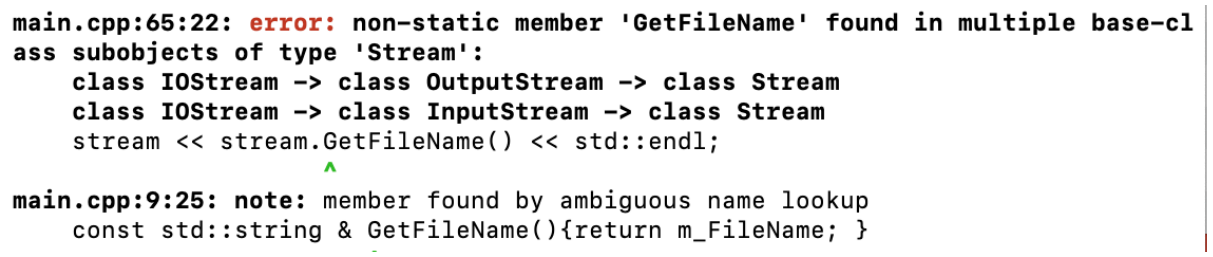
Instead, rely on polymorphism so that the correct behaviour is invoked polymorphically without having to use RTTI.

Lecture 110: Abstract Classes and Pure Virtual Functions

Pure Virtual Functions

- This is done by adding a '= 0' to the end of a virtual function. This function will not have any implementation. But the child classes will have to provide an implementation for this serialise() function.
- By marking a function as pure virtual, this makes the **Document class** an **ABSTRACT class.** An abstract class has at least **1 pure virtual function,**
- Abstract classes **cannot be instantiated.** But you **can create their pointers or references.**
- In the XML class, we have to provide an implementation for all the functions declared as pure virtual.

Lecture 111: Diamond Inheritance



```
main.cpp:65:22: error: non-static member 'GetFileName' found in multiple base-cl
ass subobjects of type 'Stream':
    class IOStream -> class OutputStream -> class Stream
    class IOStream -> class InputStream -> class Stream
    stream << stream.GetFileName() << std::endl;
                     ^
main.cpp:9:25: note: member found by ambiguous name lookup
    const std::string & GetFileName(){return m_FileName; }
```

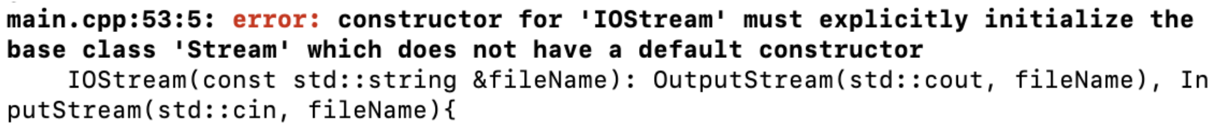
Refer to the code in iostream. There are 2 instances of the stream object in the iostream object, and when you try to invoke the function that is inherited twice, the **compiler gives an AMBIGUOUS ACCESS error** for this function. To resolve this error, we need to ensure that there is only 1 instance of the stream object in the IOStream and we can do this by using **virtual inheritance.**

So inputstream and outputstream will both inherit **virtually from the base class,** and this will ensure, that in the iostream object, there is **only 1 instance of the Stream object.**
This is done by using

Class InputStream : virtual public Stream{…}

And the same for the outputstream.

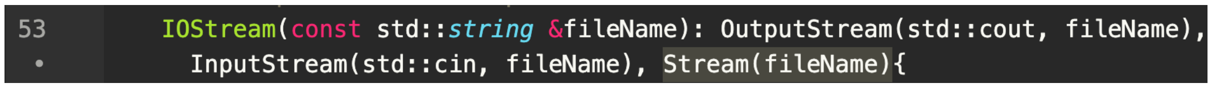
Upon compiling now, we get another error that suggest there is no appropriate default constructor available:



```
main.cpp:53:5: error: constructor for 'IOStream' must explicitly initialize the
base class 'Stream' which does not have a default constructor
    IOStream(const std::string &fileName): OutputStream(std::cout, fileName), In
putStream(std::cin, fileName){
```

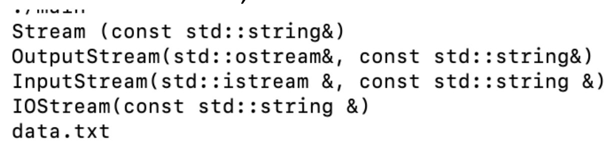
How does virtual inheritance ensure only 1 instance? When the child class instance is created, the constructor of the child class will directly create an instance of the base class stream and our Stream class does not have a default constructor it only has a parameterized constructor. **So we need to invoke that parameterized constructor from the iostream class.**

Do this by modifying the constructor call from the IOStream class:



```
    IOStream(const std::string &fileName): OutputStream(std::cout, fileName),
      InputStream(std::cin, fileName), Stream(fileName){
```

This works. Notice, that now the Stream constructor is displayed **only once.**



```
Stream (const std::string&)
OutputStream(std::ostream&, const std::string&)
InputStream(std::istream &, const std::string &)
IOStream(const std::string &)
data.txt
```

this shows that there is only 1 isnstance of the Stream object in the iostream object, which is why when we now call .GetfileName() this works.

Refer to the slides to see how this is internally implemented.