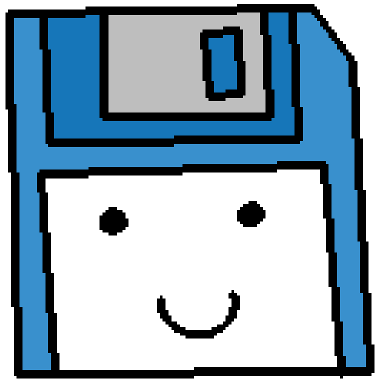
Classes and Design

Written by Rachel J. Morris, last updated 2016-02-08

Splitting up Declarations & Definitions



.hpp and .cpp files

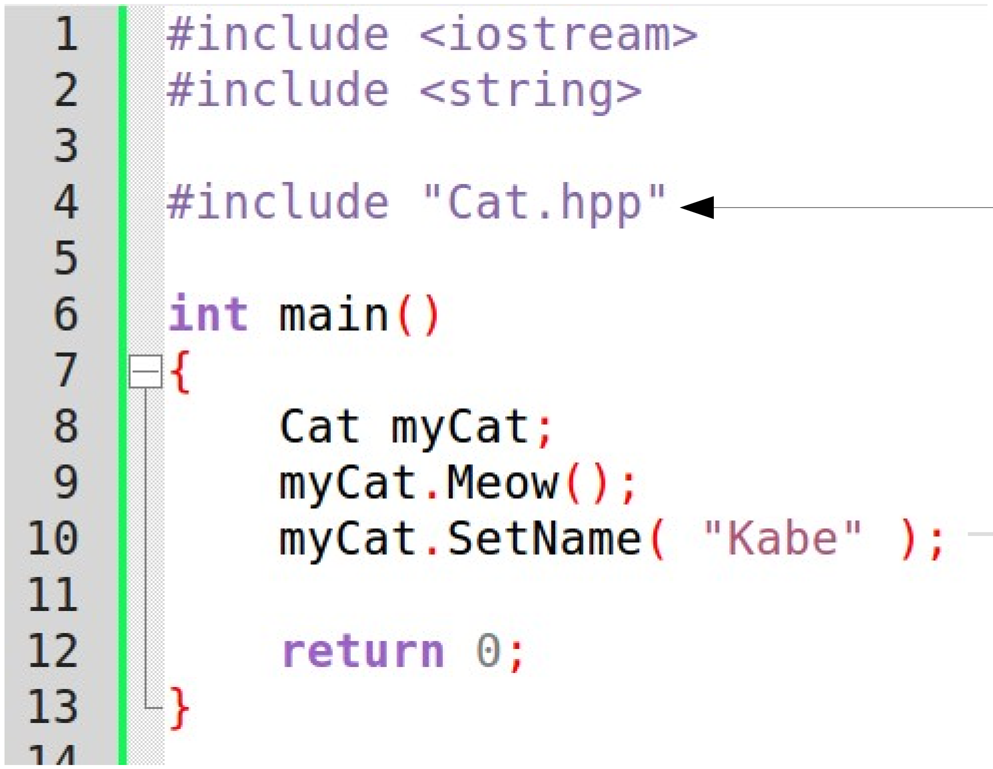
In C++, it is standard to split up your **class declarations** and **class definitions** into two separate files –
The header (.hpp) and the source (.cpp) files.

One of the good things about this is that, you can get a high-level “overview” of what a class contains and does (the **interface**) from peeking at the header file,

While the more in-depth **implementation** (the actual logic code) is hidden in the source files.

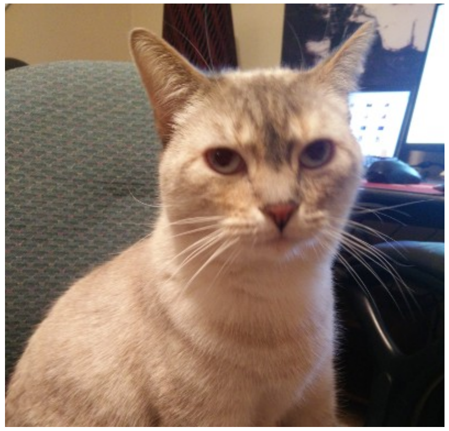
.hpp and .cpp files

When you want to create a new class, you need to get into the habit of creating a header (.hpp) file and a source (.cpp) file – usually matching the class' name.



```
#include <iostream>
#include <string>

#include "Cat.hpp"

int main()
{
    Cat myCat;
    myCat.Meow();
    myCat.SetName( "Kabe" );

    return 0;
}
```

“Cat” class:
Cat.hpp, Cat.cpp



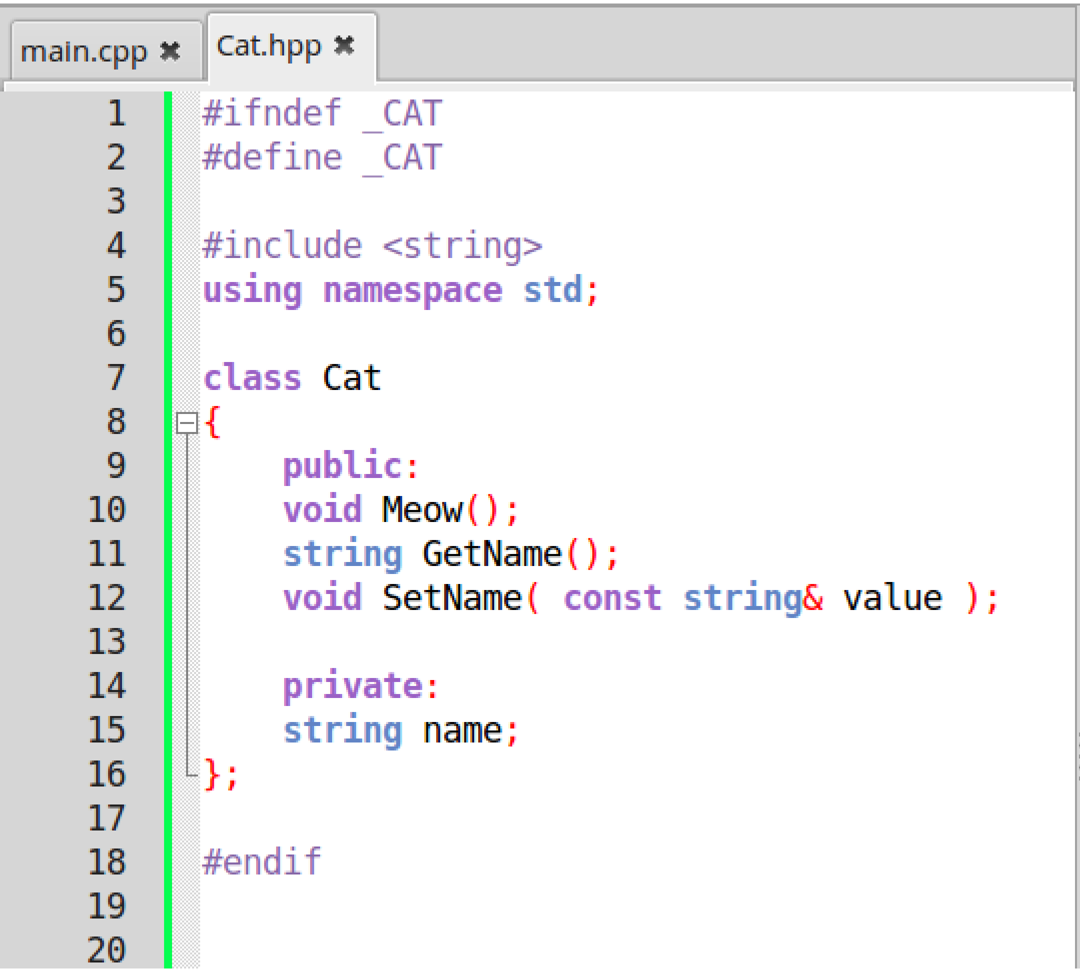
.hpp and .cpp files

Cat.hpp

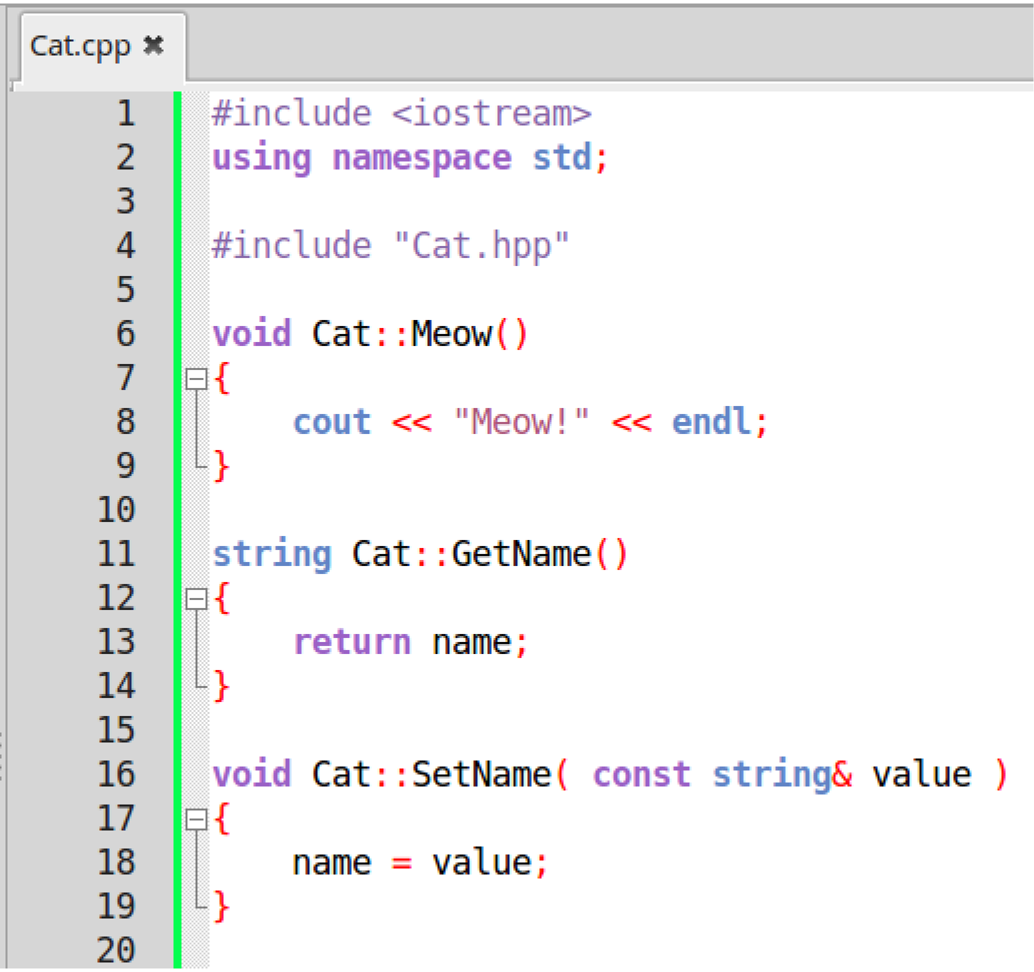
Class declaration, function declarations, variable declarations

Cat.cpp

Function definitions



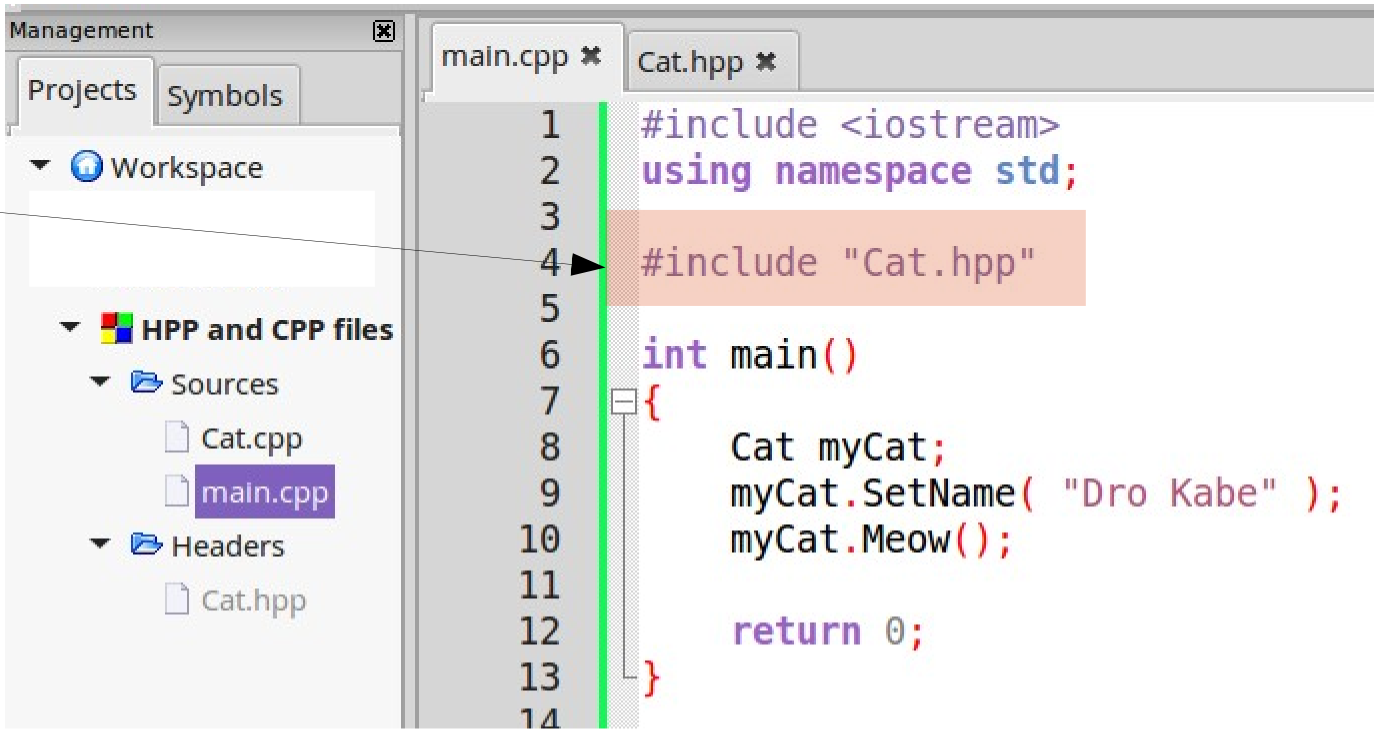
```cpp
#ifndef _CAT
#define _CAT

#include <string>
using namespace std;

class Cat
{
    public:
    void Meow();
    string GetName();
    void SetName( const string& value );

    private:
    string name;
};

#endif
```



```cpp
#include <iostream>
using namespace std;

#include "Cat.hpp"

void Cat::Meow()
{
    cout << "Meow!" << endl;
}

string Cat::GetName()
{
    return name;
}

void Cat::SetName( const string& value )
{
    name = value;
}
```

.hpp and .cpp files

We don't need to use `#include` for our .cpp Files, only any header (.hpp) files.

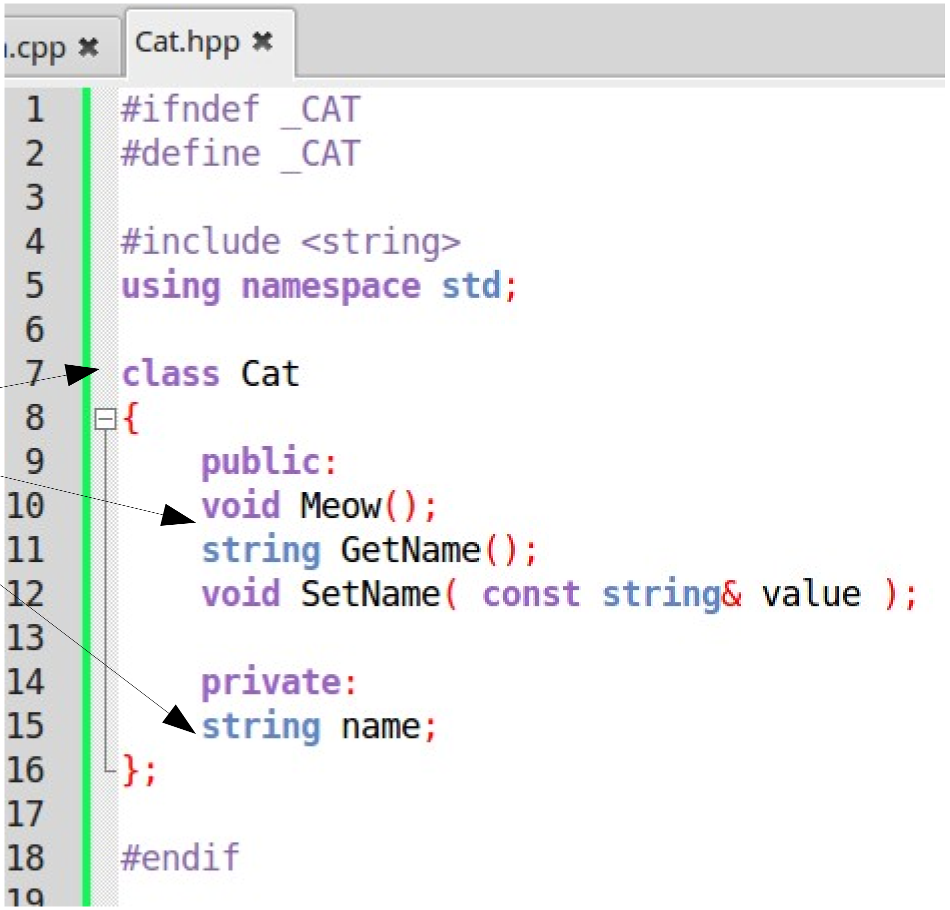


```
#include <iostream>
using namespace std;

#include "Cat.hpp"

int main()
{
    Cat myCat;
    myCat.SetName( "Dro Kabe" );
    myCat.Meow();

    return 0;
}
```

.hpp and .cpp files

Header Files (.hpp, or .h)

With a header file, you will declare your class, as well as its member variables and functions.



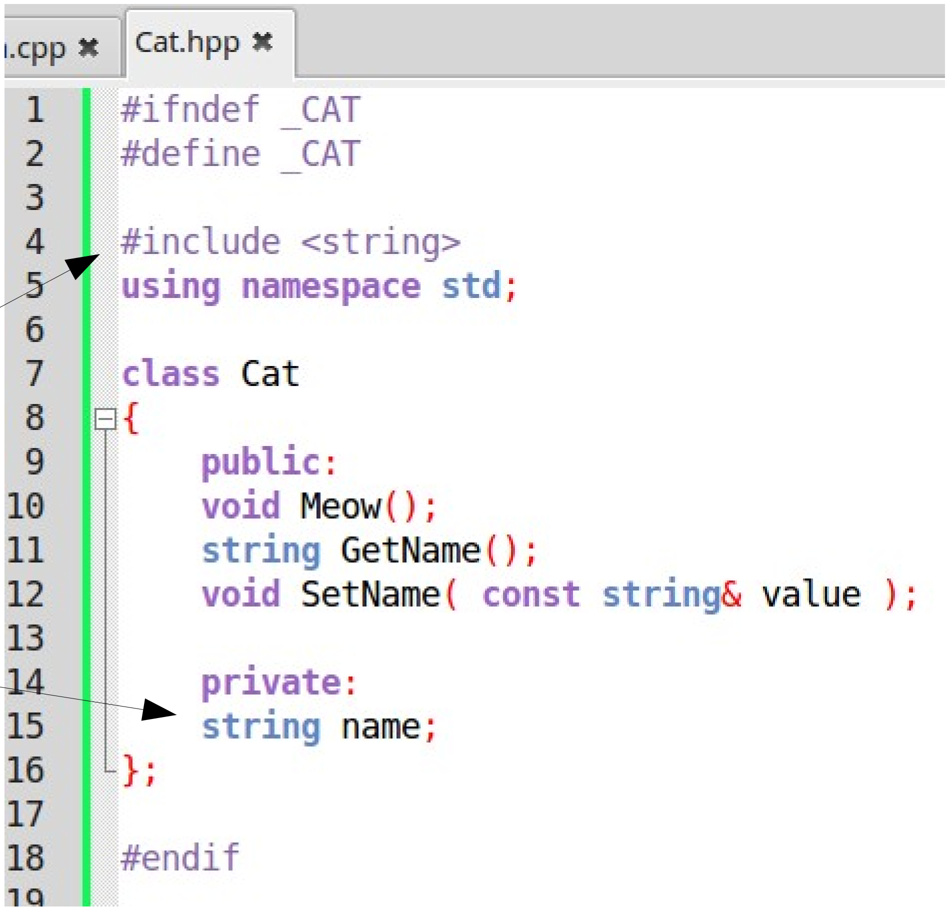
```cpp
#ifndef _CAT
#define _CAT

#include <string>
using namespace std;

class Cat
{
    public:
    void Meow();
    string GetName();
    void SetName( const string& value );

    private:
    string name;
};

#endif
```

.hpp and .cpp files

Header Files (.hpp, or .h)

If your class declaration uses any special libraries, you need to include them in this file as well.

Here, we have a **string** variable, so we need to include the string library.

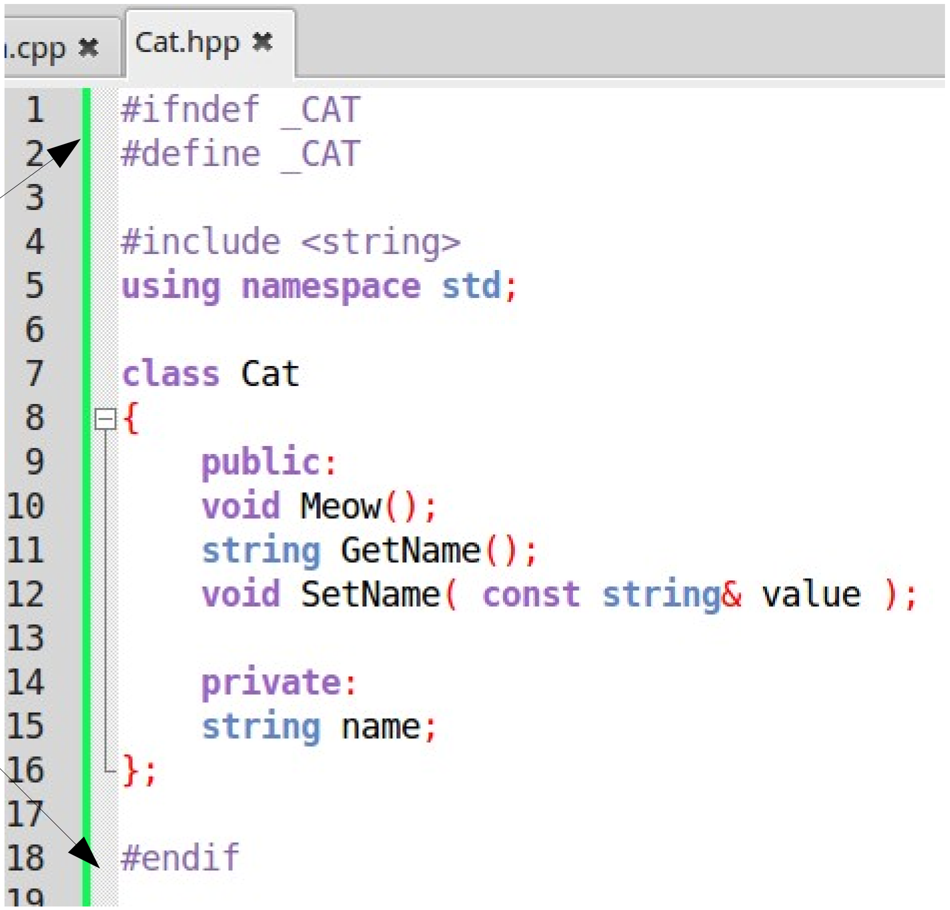


```
#ifndef _CAT
#define _CAT

#include <string>
using namespace std;

class Cat
{
    public:
    void Meow();
    string GetName();
    void SetName( const string& value );

    private:
    string name;
};

#endif
```

.hpp and .cpp files

Header Files (.hpp, or .h)

And, surrounding your header code, you will need to add some **~special stuff~** that might look confusing right now.



```cpp
#ifndef _CAT
#define _CAT

#include <string>
using namespace std;

class Cat
{
    public:
    void Meow();
    string GetName();
    void SetName( const string& value );

    private:
    string name;
};

#endif
```

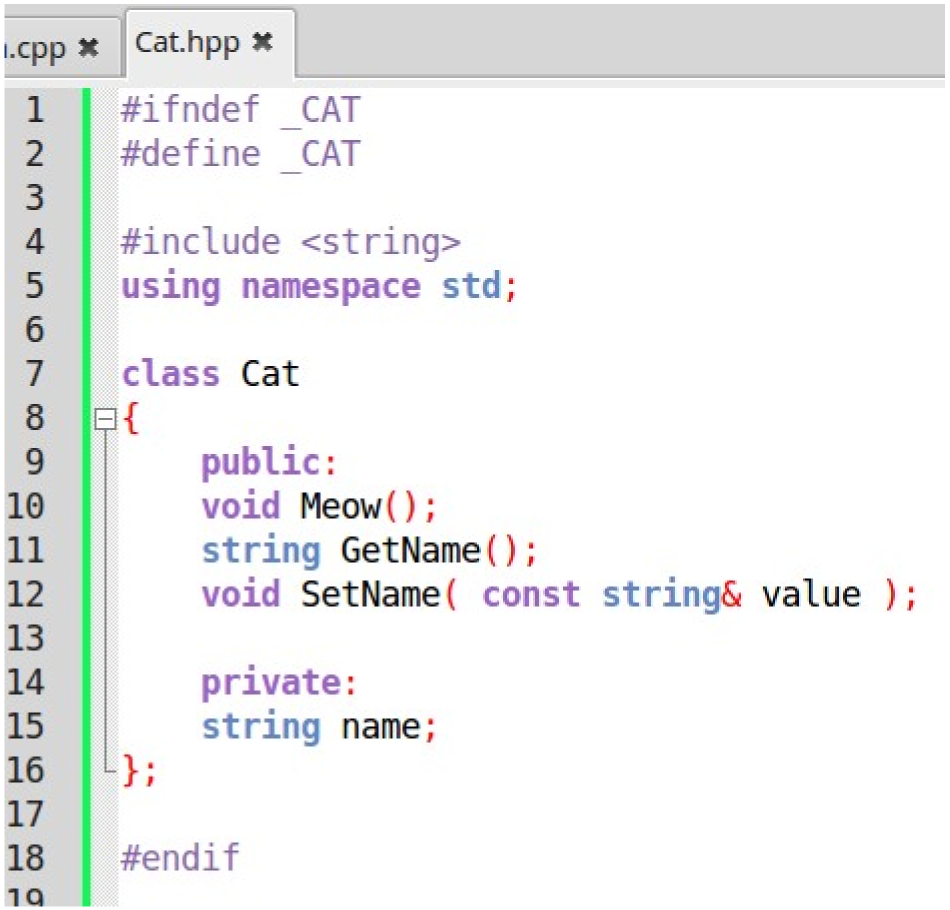
.hpp and .cpp files

Header Files (.hpp, or .h)

If we include this **header** file from multiple **source** files, we will get a compile error – it will say that we're *redeclaring* the class.

To make sure the compiler doesn't think we're declaring our class multiple times, we have to add **Preprocessor Statements**.

Preprocessor Statements begin with the # (pound sign, hash tag, whatever), Just like our `#include`.



```
#ifndef _CAT
#define _CAT

#include <string>
using namespace std;

class Cat
{
    public:
    void Meow();
    string GetName();
    void SetName( const string& value );

    private:
    string name;
};

#endif
```

.hpp and .cpp files

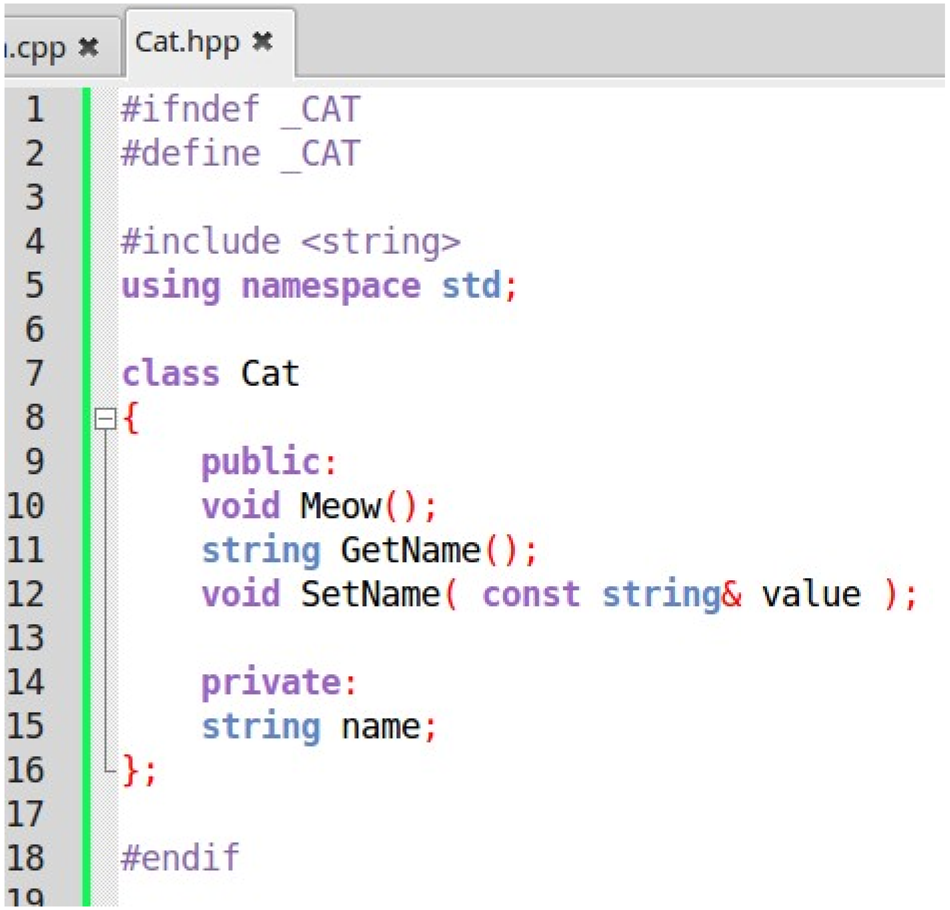
Header Files (.hpp, or .h)

For now, memorize this:

```
#ifndef _CLASSNAME
#define _CLASSNAME
#endif
```

It basically says,
"If _CLASSNAME is not defined, Define _CLASSNAME"
Your code goes after this, then end with
"End If statement"

Here, you can name _CLASSNAME anything. I tend to use an underscore, then the class name in all caps to make sure it's original.

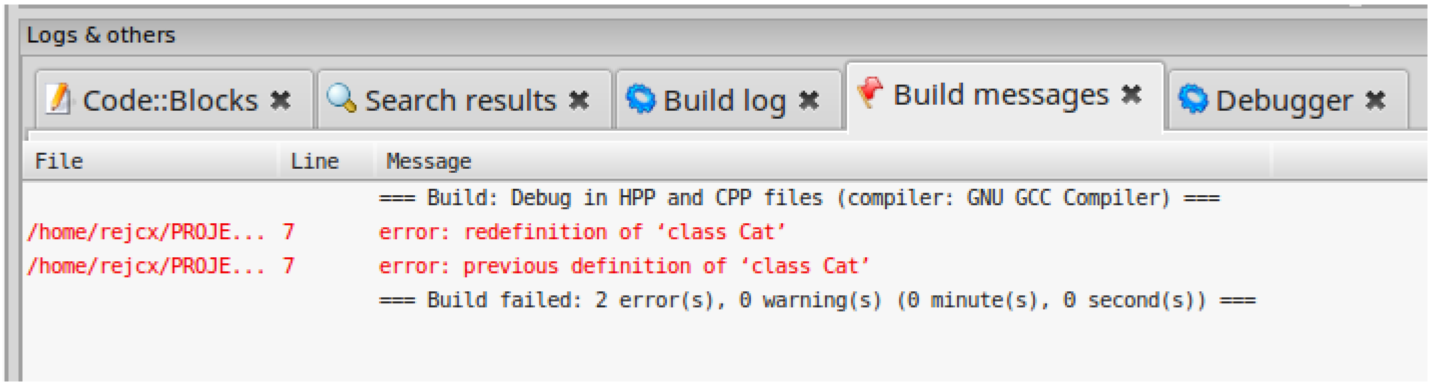


```
#ifndef _CAT
#define _CAT

#include <string>
using namespace std;

class Cat
{
    public:
    void Meow();
    string GetName();
    void SetName( const string& value );

    private:
    string name;
};

#endif
```

.hpp and .cpp files

Header Files (.hpp, or .h)

If you don't add the #ifndef, #define, and #endif statements, you might get a build error similar to:



```
File                Line   Message
                           === Build: Debug in HPP and CPP files (compiler: GNU GCC Compiler) ===
/home/rejcx/PROJE... 7     error: redefinition of ‘class Cat’
/home/rejcx/PROJE... 7     error: previous definition of ‘class Cat’
                           === Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Redefinition of your class.

This means that the compiler is reading the class declaration multiple times, and complaining that it “already exists”. This is why we need the defines.

.hpp and .cpp files

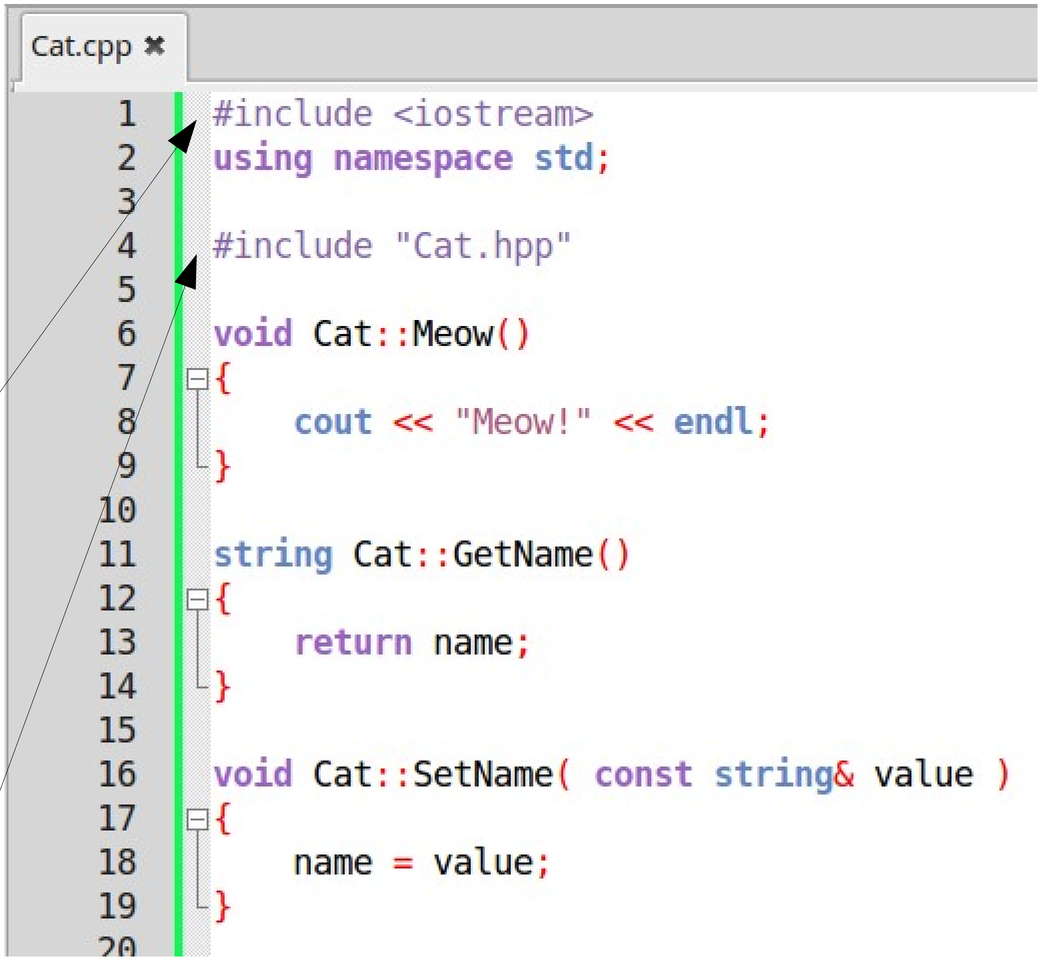
Source Files (.cpp)

The class' source file is where you actually **define** your functions – add the function bodies and internal code.

You just need to make sure to #include any libraries you're using, and any of your own .hpp files you're using in this file.

Usually, this is at least the class' header file.

Notice that with the files we've made, we use double-quotes around the file name, rather than the < > brackets like for C++ libraries.

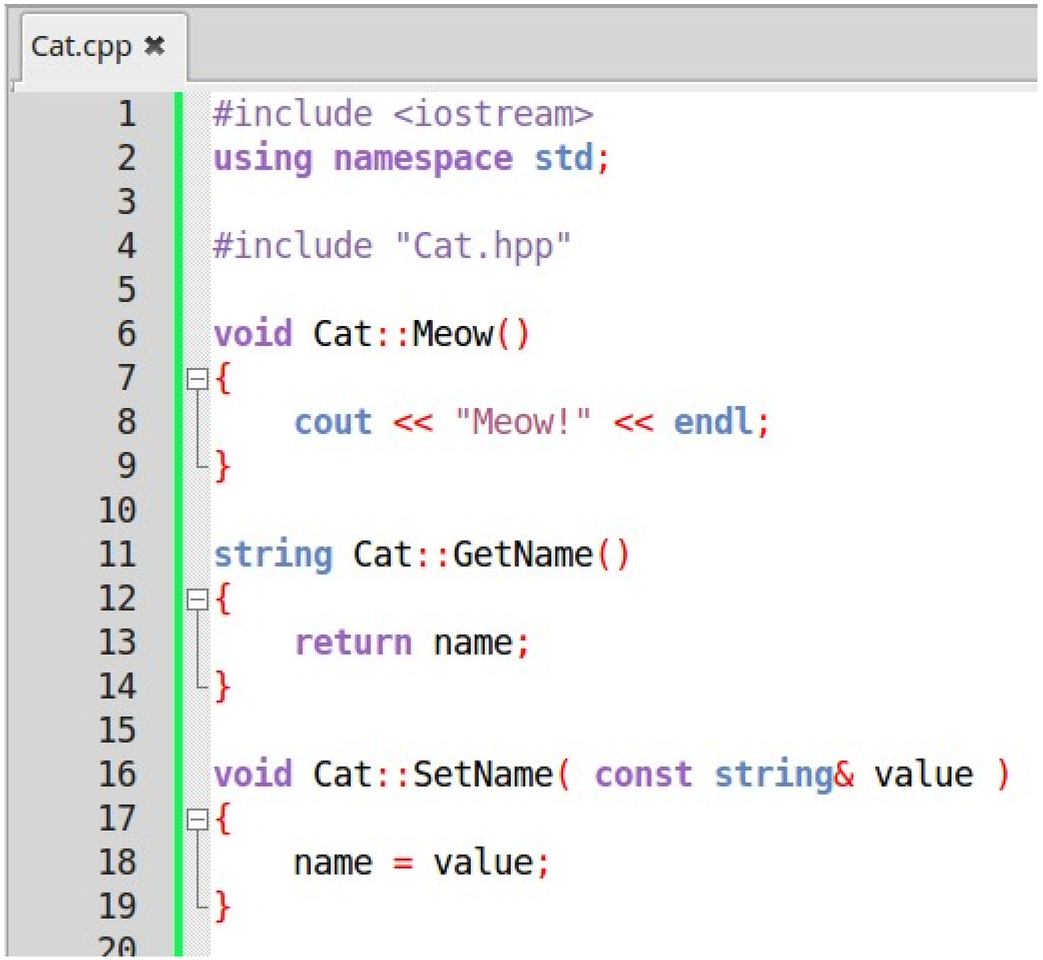


```
#include <iostream>
using namespace std;

#include "Cat.hpp"

void Cat::Meow()
{
    cout << "Meow!" << endl;
}

string Cat::GetName()
{
    return name;
}

void Cat::SetName( const string& value )
{
    name = value;
}
```

.hpp and .cpp files

Source Files (.cpp)

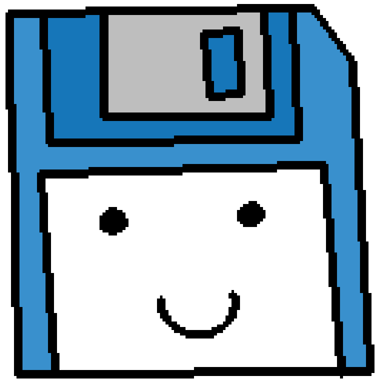
See the 2nd class lecture for more on how to define a function **outside** of the class declaration, using the scope resolution operator ::



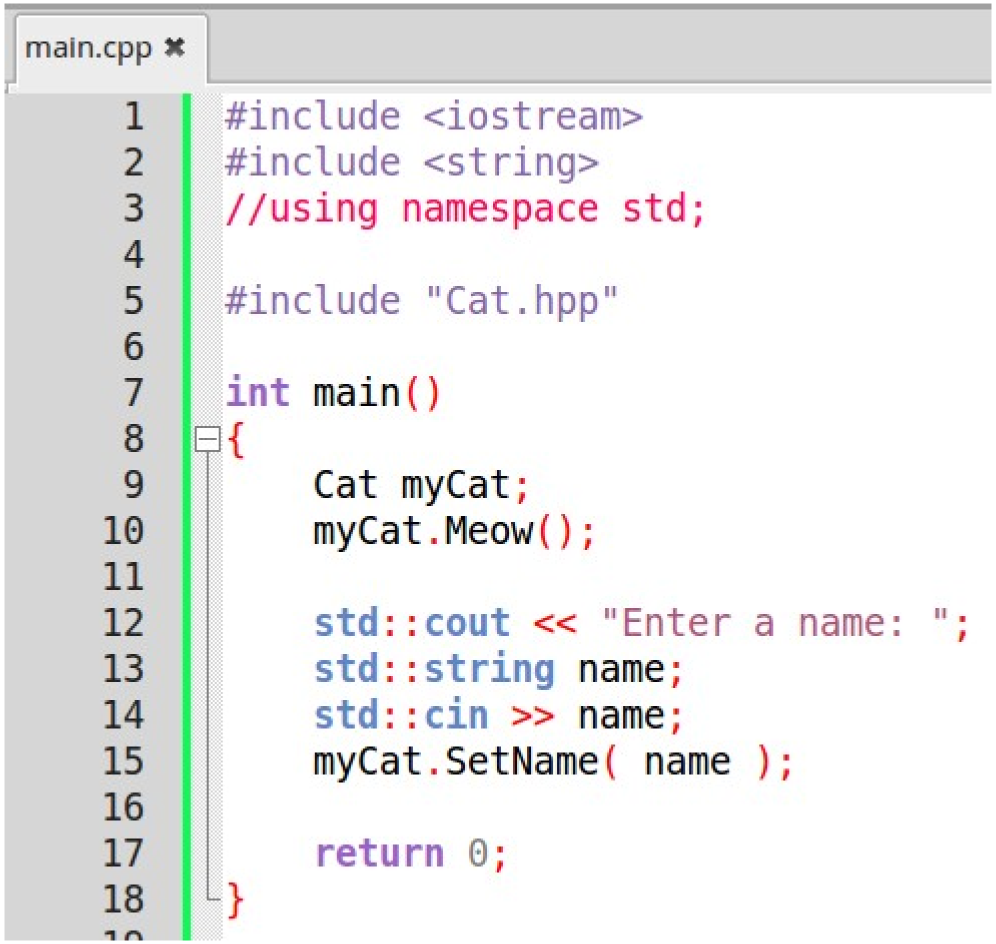
Cat.cpp

```cpp
#include <iostream>
using namespace std;

#include "Cat.hpp"

void Cat::Meow()
{
    cout << "Meow!" << endl;
}

string Cat::GetName()
{
    return name;
}

void Cat::SetName( const string& value )
{
    name = value;
}
```

Namespaces



Namespaces



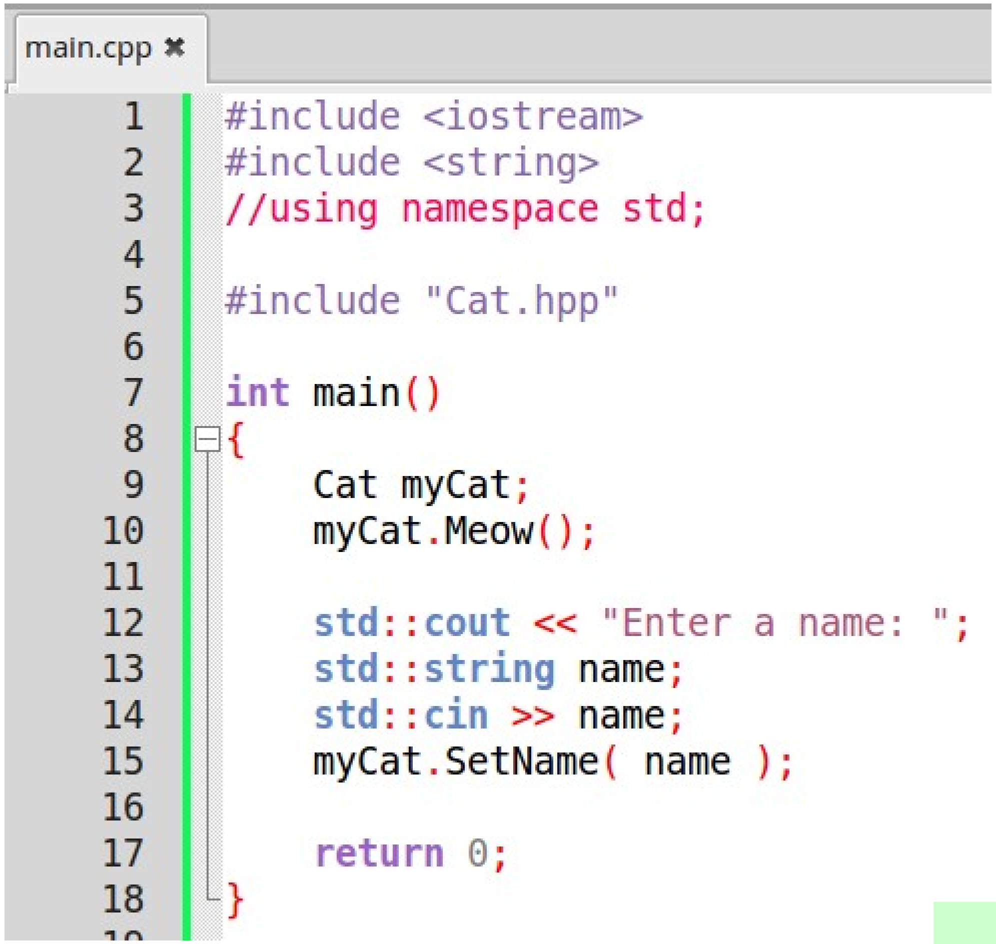
main.cpp

```cpp
#include <iostream>
#include <string>
//using namespace std;

#include "Cat.hpp"

int main()
{
    Cat myCat;
    myCat.Meow();

    std::cout << "Enter a name: ";
    std::string name;
    std::cin >> name;
    myCat.SetName( name );

    return 0;
}
```

Know how we're writing
`using namespace std;`
At the beginning of all our programs?

Without that line, we would have to prefix any commands or objects we're using from the standard library with
std::

Namespaces



main.cpp

```cpp
#include <iostream>
#include <string>
//using namespace std;

#include "Cat.hpp"

int main()
{
    Cat myCat;
    myCat.Meow();

    std::cout << "Enter a name: ";
    std::string name;
    std::cin >> name;
    myCat.SetName( name );

    return 0;
}
```

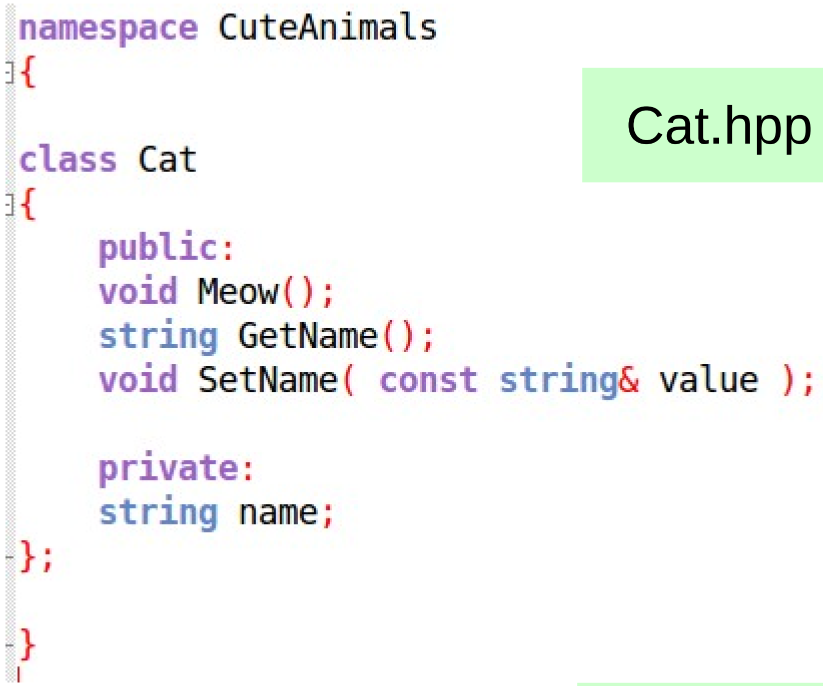
cout, cin, **string**, and others are all part of the **std** namespace of the C++ standard library.

A **namespace** is a way to store a set of code under a special label, a special *name*.

If we're using libraries from multiple locations, there's a chance that there could be a **naming conflict** – that is, two libraries could potentially use the same name for their own functions or classes.

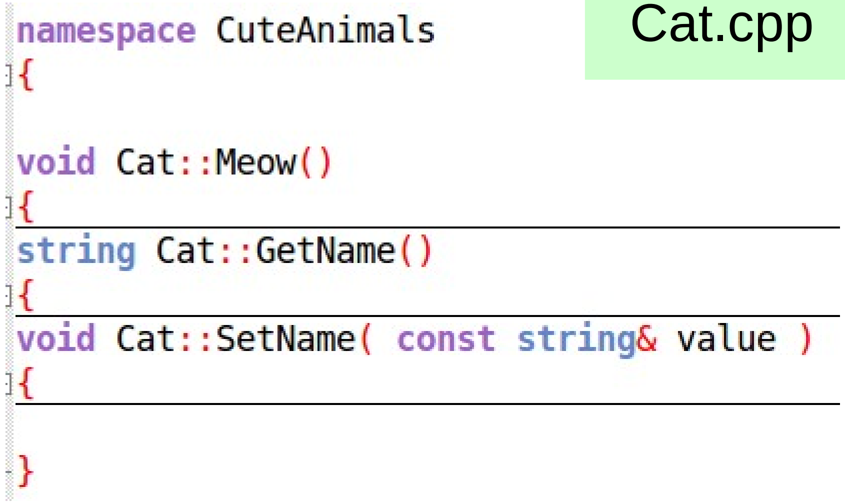
So we can use **namespace** to mitigate this, by specifying exactly where our functions and classes come from.

Namespaces



```
namespace CuteAnimals
{

class Cat
{
    public:
    void Meow();
    string GetName();
    void SetName( const string& value );

    private:
    string name;
};

}
```

Cat.hpp



```
namespace CuteAnimals
{

void Cat::Meow()
{
string Cat::GetName()
{
void Cat::SetName( const string& value )
{

}
```

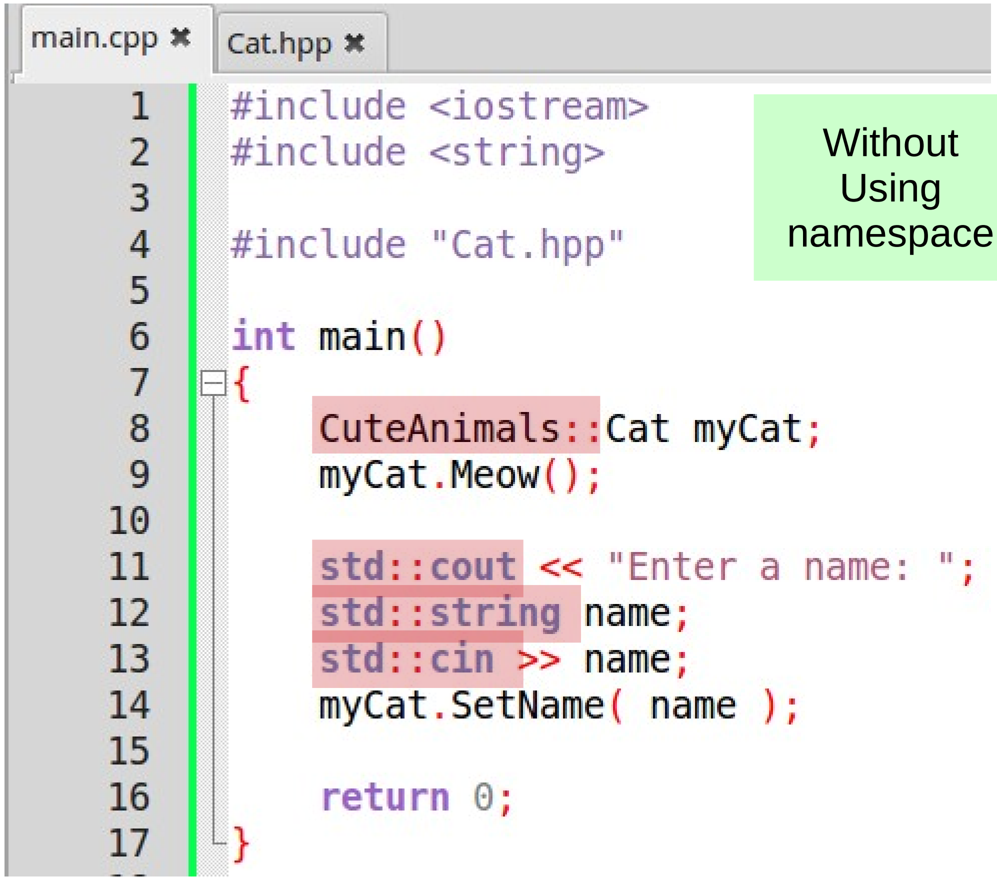
Cat.cpp

We can write our own namespaces by simply using the **namespace** keyword, followed by our namespace name, then surrounding the code between curly braces { }

The namespace specification should go around both the class declaration in the .hpp file, and the function definitions in the .cpp file.

Namespaces

Then, we would either need to prefix any usage of this class with the **namespace**, or write **`using namespace YOUR_NAMESPACE;`** At the beginning of the program.

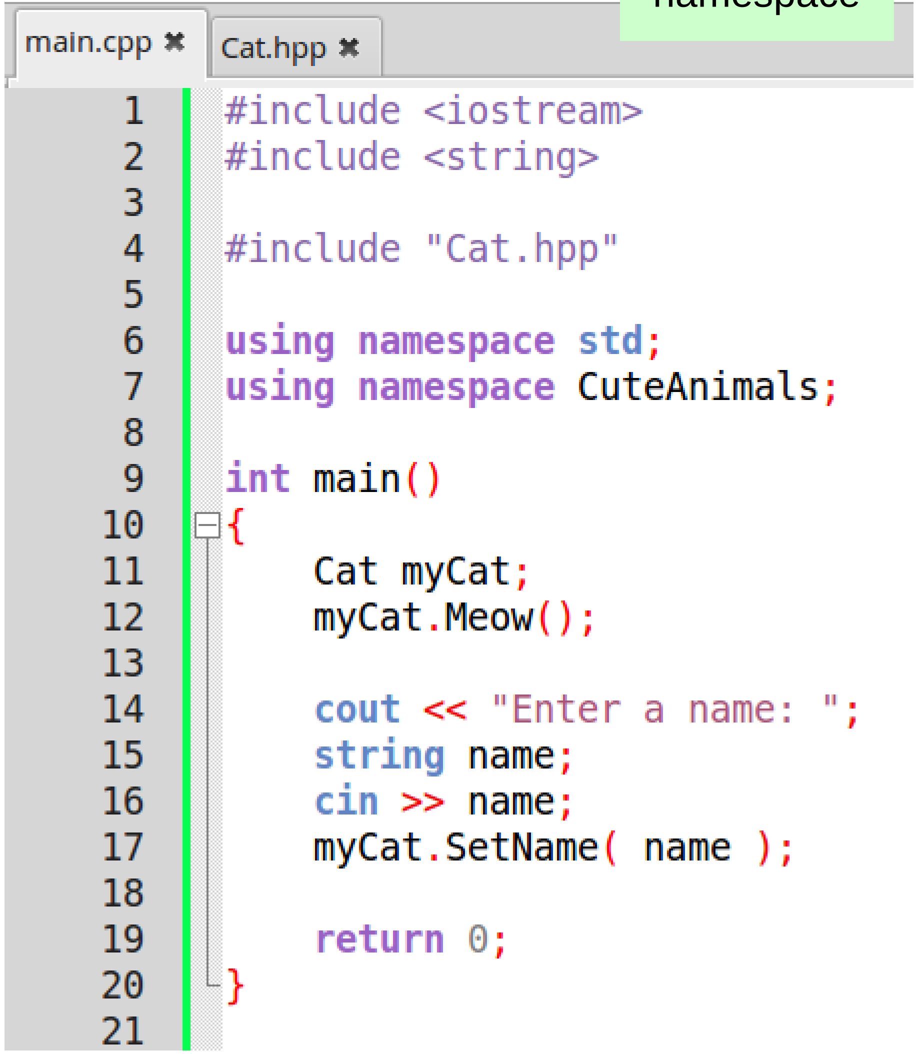


main.cpp ✖ Cat.hpp ✖

```cpp
#include <iostream>
#include <string>

#include "Cat.hpp"

int main()
{
    CuteAnimals::Cat myCat;
    myCat.Meow();

    std::cout << "Enter a name: ";
    std::string name;
    std::cin >> name;
    myCat.SetName( name );

    return 0;
}
```

Without Using namespace

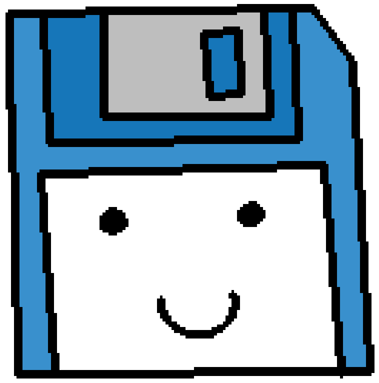
With Using namespace



main.cpp ✖ Cat.hpp ✖

```cpp
#include <iostream>
#include <string>

#include "Cat.hpp"

using namespace std;
using namespace CuteAnimals;

int main()
{
    Cat myCat;
    myCat.Meow();

    cout << "Enter a name: ";
    string name;
    cin >> name;
    myCat.SetName( name );

    return 0;
}
```

Why use classes?

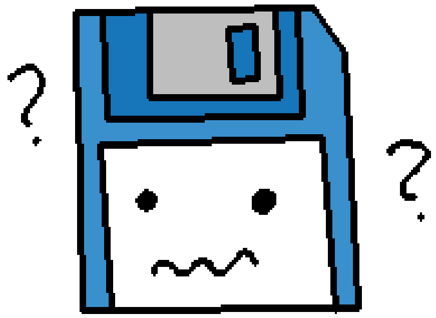


Why use classes?

So, what's with all these rules and features?

Why even bother using classes?

Why not keep it simple?



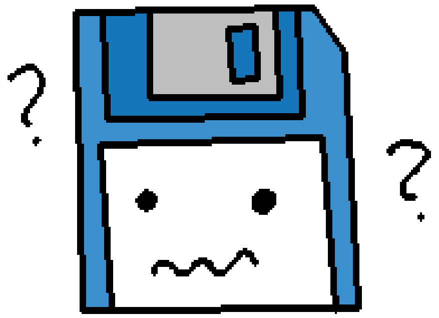
Why use classes?

Classes are a very powerful aspect of Object Oriented Programming.

Programs are very infrequently “simple”; they can get quite large, and may need to be maintained for years or even decades.

Being able to structure our code in a logical manner allows us to better maintain the code, and hopefully keep it cleaner.

There are a few things to keep in mind when designing classes...

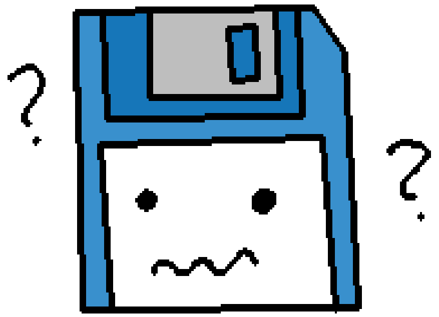


Encapsulation

Encapsulation refers to restricting access to the internals of a class (or object),

as well as designing the class so that there is an “interface” that an outside user/programmer has access to for that class (usually the public member functions)

We utilize **public**, **protected**, and **private** characterizations for our internal members (variables and functions) in order to define how our class is to be used, and how it works behind-the-scenes.

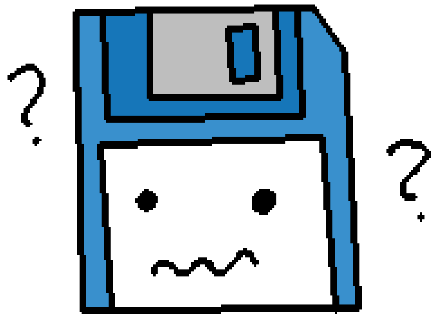


Information Hiding

Information Hiding is related to encapsulation, and also is about hiding the inner-workings of an object from the user/other programmers.

This is because the inner workings may change later on, but the user only needs to know *what the function or object does*, not how it does it.

By having public functions available to the user/programmer, we can change the code behind-the-scenes, but still offer the same functionality.



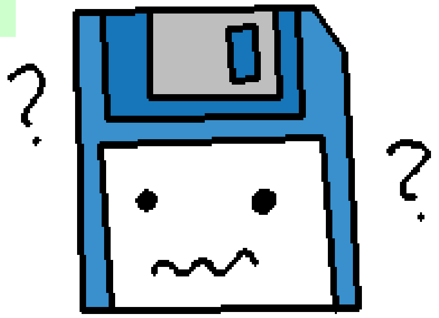
Abstraction

Abstraction is about reducing duplicate code in the long run by creating generic objects that can be reused and repurposed for different items

(for example, a generic **List** that can store any data-type within).

It is about creating a structure around our data – how do we design a piece of software that handles our data meaningfully?

We abstract a problem by writing functions and objects, and by utilizing features of our language to make our software generic, such as by using **templates** or **polymorphism** (later lectures).



Abstraction

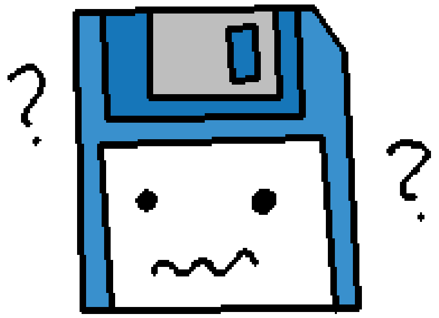
For example, let's say we have an image file...

Each pixel of the image has a Red, Green, and Blue value, and each R,G,B value is in order, in a 2D grid.

We could write a class or set of functions that know how to load and save these image files.

Later, somebody could use your class to make a program that will load an animation, as a series of your image files for each frame.

You've abstracted image loading to make it easier, and someone else has abstracted the problem of animating, by extending your work for an animation tool.



There is a lot to read – in books, blogs, articles, and more – about good software design and principles.

It is good to become somewhat acquainted with these concepts in order to become a good software developer.

Software design can be a very logical and creative process, and if that's your cup of tea, it can be a lot of fun.