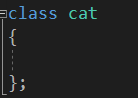
**Chapter 15: Object Oriented Programming**

Object oriented programming is a different approach to writing code. There are two concepts that need to be addressed: Classes and Objects. A class can be viewed as a blueprint from which many objects are created. Imagine the blueprint (class) for a house. Many houses (objects) can be constructed from these plans, each one containing its own attributes / locational data.

The class approach builds from the struct approach covered in a previous chapter. However, there are benefits to using classes that structs do not possess. These are techniques such as inheritance and polymorphism, which will reduce the amount of code you are required to write, whilst also making for a stronger code base.

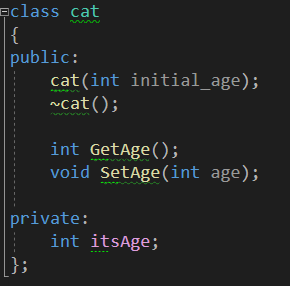
To declare a new class, simply use the keyword class followed by the name you wish to call your class. The following example creates a class called Cat. This can be added above the main function just as you would a struct. However, it is best practice to create separate files. Each class will have it’s own cpp file and a header file.



To control what can be accessed and changed in your class you will need to use accessors. These restrict access to internal functions / variables according to the table below:

|  |  |
| --- | --- |
| **Accessor** | **Description** |
| public | Anything falling under public can be accessed from anywhere. |
| protected | Access is only given to the current class and child classes. |
| private | Access is only available to the current class.  This is the default access level if one is not set. |

Next, we will add some functions and variables to our cat class. These will use differing accessor levels.



As you can see from the above code snippet there are two functions with the same name as the class, i.e. **cat**.

The following is a constructor. A constructor is the function that will be called when an object of this class type is created. A constructor does not have a return type like standard functions do but may take whatever values as parameters as are required for setup.



The second function with the same name as the class is the destructor. A destructor does not have a return type and cannot take any parameters. The function name is always preceded with a tilde ~ character to distinguish it from the constructor.

The destructor will always be called when an object goes out of memory when created on the stack or is deleted from memory when created on the heap.

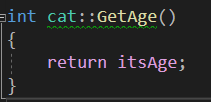


When writing the definitions for functions, you need to include the header class at the top of your source file. Each function prototype requires the name of the class the function is referring to. See the GetAge() function below.

**In the header:**



**In the source file (cpp file):**

****

If a function definition contains minimal code as in the above example it is suitable to be implemented as an inline function. That is to say the implementation is contained within the header. The above code snippets modified into an inline function would look like this:

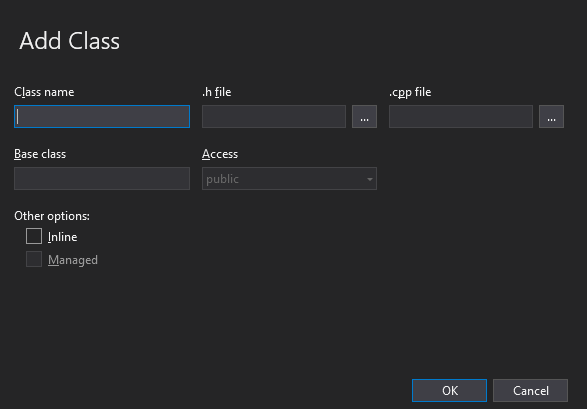


A small detail to notice here is the semicolon placement. This is contained in the curly braces.

**Creating a class**

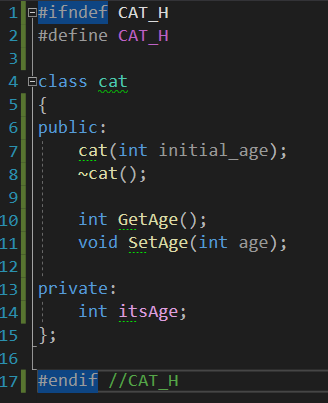
Follow along with the following instructions.

Create a new project and once loaded right click in the solution explorer or hit Shift+Alt+C. This will bring up the following dialog box:



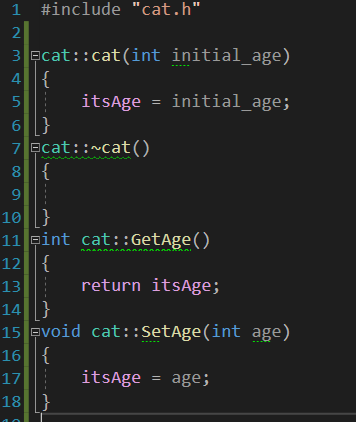
Add the class name cat, this will automatically complete the header and source data and click OK.

In the cat.h file (header file) add the following code:

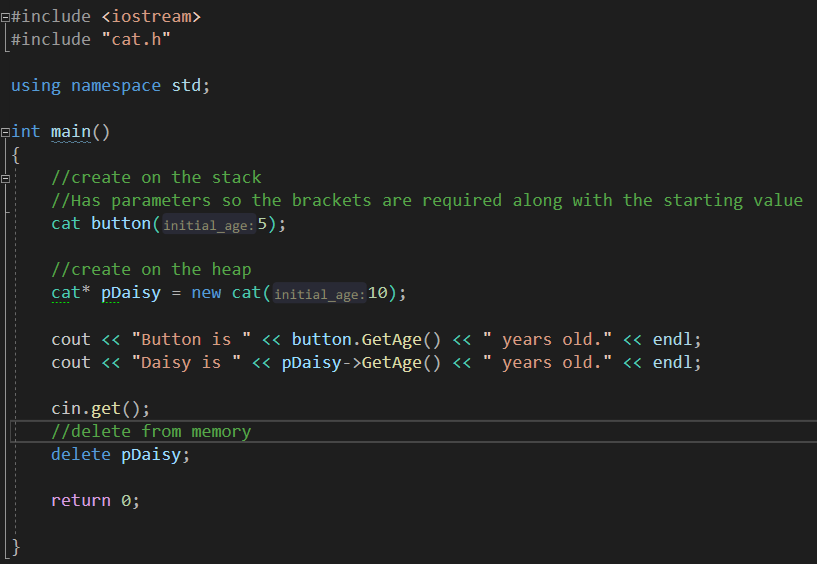


Two things may standout here: ifndef and define. These are used as include guards. A common error with classes and header files is double declaration. When you have conflicting includes in the wrong place can lead to very puzzling errors in your program.

Next, in the source file add the following:



And finally, in the main replicate the following code:



Run the program and see what results you get.

**Inheritance**

Inheritance is a key benefit to using an objected oriented approach. When you write a class, you may find that it contains all the information another class requires. Instead of duplicating this code you can instead inherit all this behaviour for free.

As an example, imagine a game character. This character has a texture for drawing its image on screen and a minimum of a positional data. We need to create a Mario class and a Luigi class. Instead of repeating this generic code we can write a base class for this data and then have both Mario and Luigi inherit this behaviour. Or as seen in the lecture, a mammal class and types of animals can inherit from the mammal class.

To inherit from another class, use the following syntax in the header file that will be doing the inheriting:





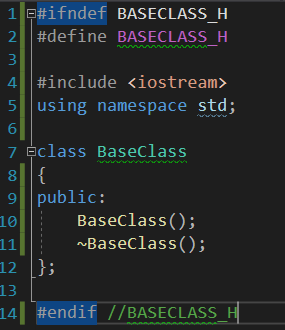
One important aspect of classes is the order that they are constructed and destructed as we will see in the next program.

**Creation Order**

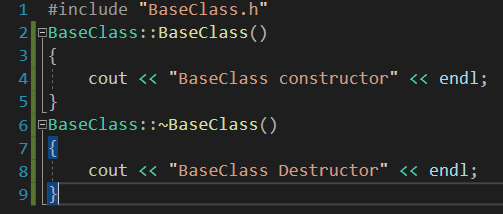
Create a new project and follow along with the instructions.

Create a new class called BaseClass using Shift+Alt+C or right clicking the solution explorer.

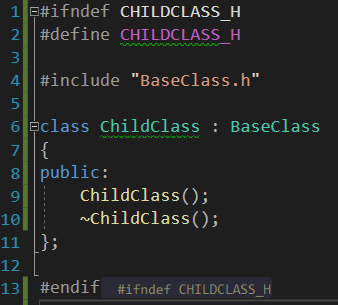
Add the following to the header file:



Next, replicate the following for the BaseClass source file:

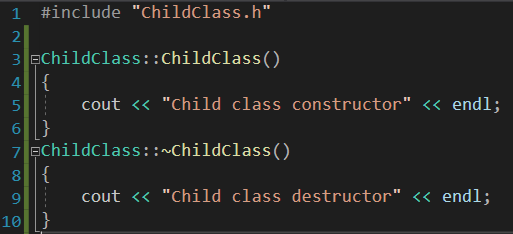


Now create a new class called ChildClass just as you did with the BaseClass and add the following to the header file:



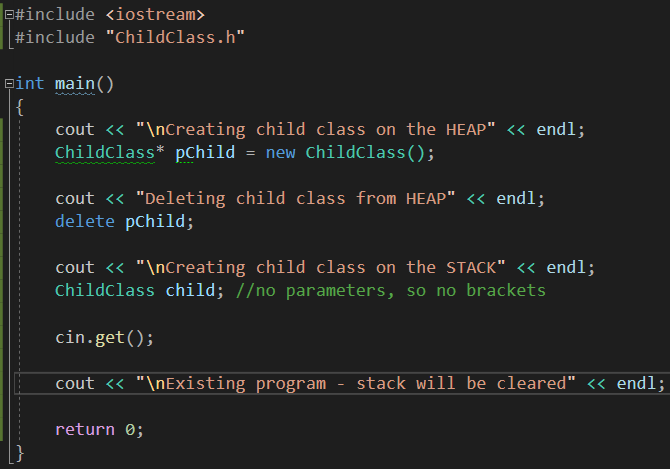
Ensure you use the correct syntax and inherit form the BaseClass as shown.

Next, complete the cpp file:



Notice due to the includes, using namespace was only added once in BaseClass.h and has been carried across.

Finally, in main, replicate the following code and run your program:



The point of this program was to examine the output and see the order of how things are called and work to give a better understanding of inheritance.

**Polymorphism**

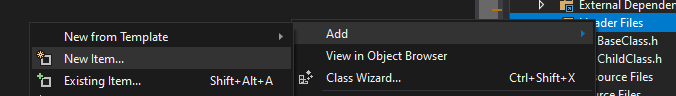
Polymorphism is another technique which will extend your programming skills. This technique allows you to call functions on a class even when it is contained within a list of its parent type. Going with the Mario example again, we could have an array of BaseCharacter type, which could hold all the characters in the level. Obviously, Mario is of Mario type and only inherits from BaseCharacter, but because of the lineage the array can hold Mario.

It will make sense when you have worked through the example program below. Before we get into that though there is one more keyword you need to be aware of. This is the **virtual** keyword. In a base class if you use this term before your functions it will enable the correct function to be called regardless of the type it was called from. This is only stated in a class you wish it to be overridden from.

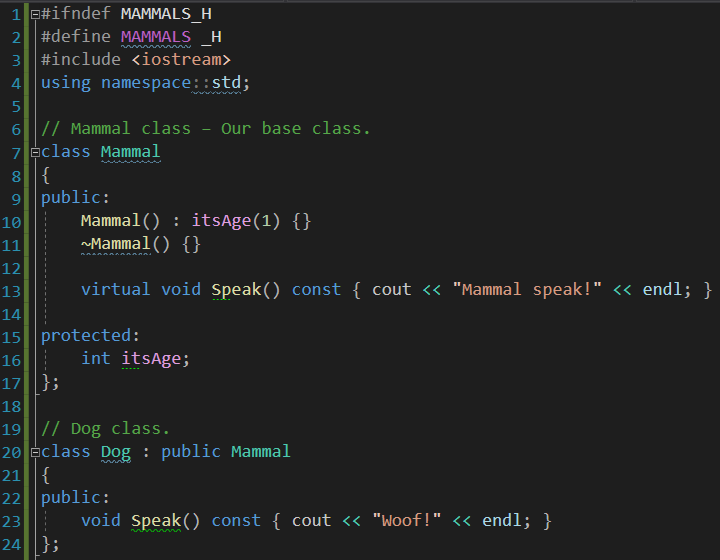
**Polymorphic Behaviour**

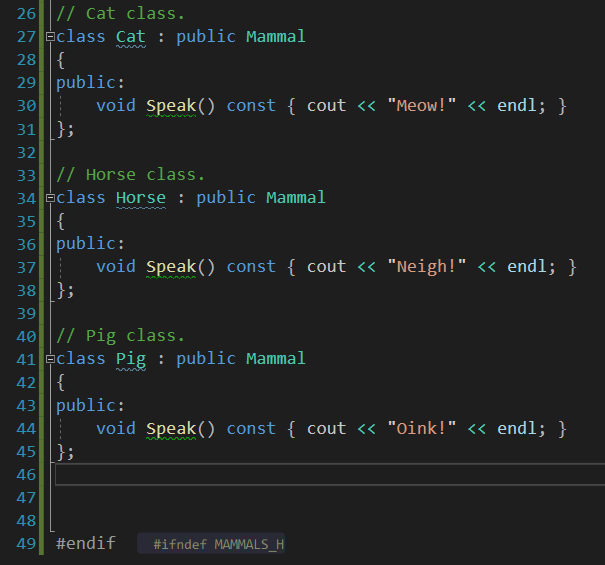
Create a new project and follow along. This will be slightly different from the other two projects, just to show an example of polymorphism working.

Right click the header folder in the solution explorer -> Add -> new item:



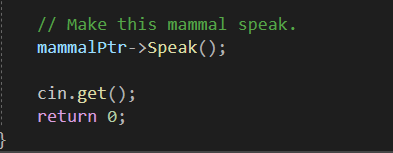
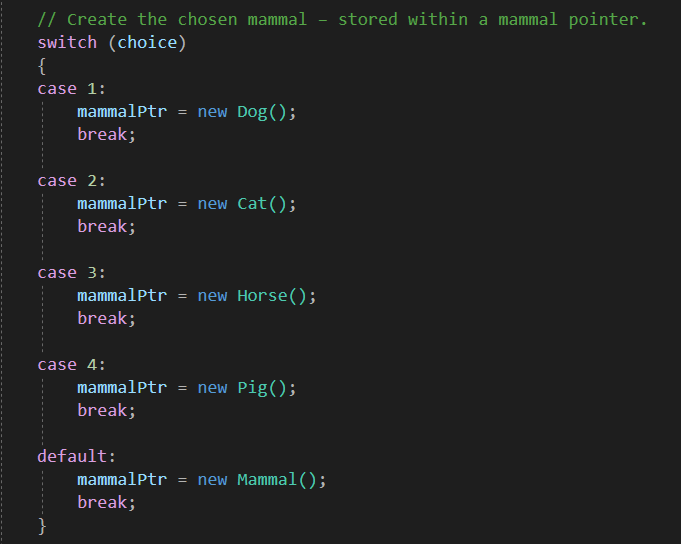
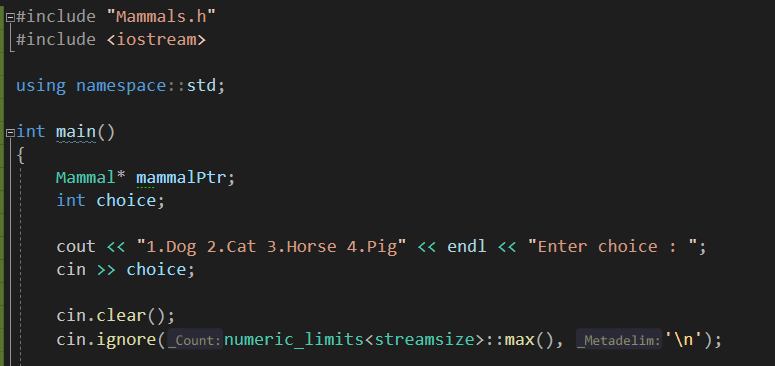
Ensure to click the file type as a header file, then give it the name Mammals.h





Classes can be contained in single header files or even contained in one source file above the main, it’s just easier to maintain and read if in separate files dedicated to the class.

Then, back in main, add the following:



Run and test your program. This program demonstrates overriding functions.

**Program 40: Pet Care**

This program is building on the last and incorporating separate files for the classes. New protected functions that will affect how our animals respond to the user.

You will need for class files, Pet must be the parent and then three child classes, Dog, Cat and Hamster. These child functions need just three methods, constructor and destructor and use of the virtual Talk method you will create in Pet.

In Pet you will need:

* A constructor which takes two pre-set integer parameters which are set to 0 in the braces (hunger and boredom)
* A destructor
* A virtual void method called Talk
* Two void methods
  + Feed and Play, each taking a related a single pre-set int food or fun, both set to 4
    - (int food = 4)
* Two private member variables
  + m\_hunger and m\_bored (set to 0)
* two protected methods
  + An inline constant function that takes type int and returns m\_hunger + m\_bored called GetMood
  + And a void PassingTime which takes a pre-set int called time that is equal to 1

The source file needs to contain the following:

* **The Constructor**
  + Cout that a new pet has arrived
  + Set m\_hunger to equal hunger (hunger being the passed variable) and the same with m\_bored equal to boredom
* **Feed**
  + Cout a message that the animal has eaten e.g. “Burp!”
  + m\_hunger needs to be set to -= food
  + create an if statement that controls if m\_hunger falls below 0 then it is set to 0, preventing negative numbers
  + call PassingTime
* **Play**
  + Same as above but with fun and m\_bored
* **Talk**
  + Cout “I am your pet and I feel “
  + Create an int mood and set to equal the GetMood method call
  + Add an if, else if, else statement
    - If mood above 15 cout “mad”
    - If above 10 – “frustrated”
    - If above 5 – “okay”
    - Else “happy”
  + Call PassingTime
* **PassingTime**
  + M\_hunger += time and the same for m\_bored.

The child override Talk method must state which animal it is but otherwise be pretty much the same as the Pet::Talk method. Feel free to change the mood values as you like depending on the animal.

In the main, create a pointer to Pet as you did with mammal, and ask the user which pet they want. Depending on the choice set the pointer to equal a new animal similar to the mammal program and call Talk(). Once this has been done create a loop with a switch statement. Give the user the following options:

* 0 – Quit
* 1 – Listen to your pet
* 2 – Feed your pet
* 3- Play with the pet
* These cases should call the appropriate method

Run the program and ensure it works correctly. Screenshot should show the pets mood changing and show when they Talk it states the correct animal too.

This program, with some changes to make it all automated, would be a simple state machine.

**Program 40 Source Code:**

**Program 40 Screenshot of output**