Classes and the Pathway to OO Design

Warwick RSE

28/09/2023

Classes

Classes

- The idea of a class is that it bundles together information that is logically connected together as well as (sometimes at least) functions that act on that data
- We have already seen classes in use - **std::vector** and **std::string** are classes
- Here we'll look into writing them and understanding the bits of them

Example Class

```
#include <iostream>
class demo{
public:
int int_data;
float float_data;
};

int main(){
  demo mydemo;
  mydemo.int_data = 14;
  mydemo.float_data = 1234.56;

  std::cout << "Int data is " << mydemo.int_data << "\n";
  std::cout << "Float data is " << mydemo.float_data << "\n";
}
```

Example Class

```cpp
#include <iostream>
class demo{
public:
int int_data;
float float_data;
};

int main(){
  demo mydemo;
  mydemo.int_data = 14;
  mydemo.float_data = 1234.56;

  std::cout << "Int data is " << mydemo.int_data << "\n";
  std::cout << "Float data is " << mydemo.float_data << "\n";
}
```

- To create your own classes you first have to have a **definition** of the data (and functions) that are bundled together
- This is a class without functions, often called a **plain old data** or **POD** class
- Technically POD is now replaced with **Trivial** and **Standard_Layout** from C++20, but you'll probably hear POD used more still

Example Class

```
#include <iostream>
class demo{
public:
int int_data;
float float_data;
};

int main(){
  demo mydemo;
  mydemo.int_data = 14;
  mydemo.float_data = 1234.56;

  std::cout << "Int data is " << mydemo.int_data << "\n";
  std::cout << "Float data is " << mydemo.float_data << "\n";
}
```

- This **public:** line states that every variable after that line can be accessed from outside the class
- The default in classes is **private:** and we'll encounter that in a bit
- There is also **protected:** that we won't really cover

Example Class

```
#include <iostream>
class demo{
public:
int int_data;
float float_data;
};

int main(){
  demo mydemo;
  mydemo.int_data = 14;
  mydemo.float_data = 1234.56;

  std::cout << "Int data is " << mydemo.int_data << "\n";
  std::cout << "Float data is " << mydemo.float_data << "\n";
}
```

- Once you have a definition of a class you need to create **instances** of the class
- The **definition** defines what **can** be stored in a class
- An instance **actually** stores data
- You can have as many instances as you want

Example Class

```cpp
#include <iostream>
class demo{
public:
int int_data;
float float_data;
};

int main(){
  demo mydemo;
  mydemo.int_data = 14;
  mydemo.float_data = 1234.56;

  std::cout << "Int data is " << mydemo.int_data << "\n";
  std::cout << "Float data is " << mydemo.float_data << "\n";
}
```

- You access **member variables** by using a **.** between the name of the **instance** and the name of the **member variable** specified in the **definition**
- While every instance will have whatever name you want, the members will always have the names specified in the definition

Methods

Methods

- A function attached to a class is generally called a **method** after the term used in an early programming language allowing this, Simula
- We've already seen methods called on classes with things like **vector.size()**
  - And **vector.push_back(value)** etc.
- The idea is to add functions that apply to the data stored in a class to the class itself
- This is where **private** variables come in - Methods can access private variables

Methods

```
#include <iostream>
class demo{
private:
int data;
public:
  void set_data(int newdata){data = newdata;}
  int get_data(){return data;}
};

int main(){
  demo mydemo;
  //Can't set mydemo.data since it is private
  mydemo.set_data(123);

  //Can't read mydemo.data either
  std::cout << "Data is " << mydemo.get_data() << "\n";
}
```

- Here we use a private variable **data** and create **set_data** and **get_data** methods to set and retrieve its value
- This would, for example, allow you to validate that data was being set to an allowed value
- Note that in the method you can just access the member variable by name

Methods

- Methods work the same as any other function but are always aware of which **instance** of the class they were called on
- This is done by a hidden parameter to every method called **this** which is a pointer to the instance that the method was called on
- You can use **this** manually to access member variables and methods, but generally you don't need to - just use the name directly, although function parameters and local variables in a method **shadow** the member variables and methods if they have the same name, so be careful!
- If you use **this**, remember that it is a pointer, so you have to access the member variables and methods using **->**, just like we saw with the iterators (although here for a different reason)

Struct

- If you've come from C then you might be familiar with a concept in C called a **struct** which does much the same thing, but doesn't have methods
- In **C++** POD classes are almost-guaranteed to be the same layout in memory as a C struct
- In fact a class with everything public and no methods is **exactly** like a C struct
- The keyword **struct** is still in C++ and simply means **a class where the default access of all members is public**
  - Use it like class if this is the behaviour that you want, commonly used for POD classes

Default Values

Default Values

```
#include <iostream>
class demo{
public:
int int_data {14};
float float_data{1234.56};
};

int main(){
  demo mydemo;

  std::cout << "Int data is "
   << mydemo.int_data << "\n";
  std::cout << "Float data is "
   << mydemo.float_data << "\n";
}
```

- You can assign default values to members of classes from C++11 onwards
- The best way to do it is to put the initial value in **{ }** after the name of the value and before the **;**
  - Technically this is called **uniform initialisation**.
- There are other methods of initialising variables such as assigning with **=** or constructing with **()** rather than **{}** but this is the one we would recommend since it is the least ambiguous for the compiler

Call function with class

Default Values

```
#include <iostream>
class demo{
private:
int data;
public:
  void set_data(int newdata)
    {data = newdata;}
  int get_data(){return data;}
};

int get_and_double(demo &d){
  return d.get_data()*2;
}

int main(){
  demo mydemo;
  //Can't set mydemo.data since it is private
  mydemo.set_data(123);

  //Can't read mydemo.data either
  std::cout << "Data doubled is "
    << get_and_double(mydemo) << "\n";
}
```

- You can pass an instance of a class to a function just like any other type in C++
- Here I am passing it as a reference
  - Generally want to do that
- Classes are generally larger than single data items
- Copying classes can be surprisingly involved and computationally expensive

Special Methods

Constructors

```
#include <iostream>
class demo{
public:
int int_data {14};
float float_data{1234.56};
demo(int i, float f)
  {int_data=i;float_data=f;}
demo()=default;
};

int main(){
  demo mydemo{6,5.6};

  std::cout << "Int data is "
    << mydemo.int_data << "\n";
  std::cout << "Float data is "
    << mydemo.float_data << "\n";
}
```

- There are special methods that you can create for a class that are used by the language in places where you don't explicitly call a function
- The most common is a **constructor** method
- This is called when the object is created and can be used to set up any parameters of the function

Constructors

```
#include <iostream>
class demo{
public:
int int_data {14};
float float_data{1234.56};
demo(int i, float f)
  {int_data=i;float_data=f;}
demo()=default;
};

int main(){
  demo mydemo{6,5.6};

  std::cout << "Int data is "
    << mydemo.int_data << "\n";
  std::cout << "Float data is "
    << mydemo.float_data << "\n";
}
```

- This is a constructor. Note that it doesn't have a return type and has the name of the class as the method name
- You can have any parameters that you want to a constructor, although some kinds of parameters have special meanings
- The constructor to be used is chosen by following the normal rules for overloaded functions

Constructors

```
#include <iostream>
class demo{
public:
int int_data {14};
float float_data{1234.56};
demo(int i, float f)
  {int_data=i;float_data=f;}
demo()=default;
};

int main(){
  demo mydemo{6,5.6};

  std::cout << "Int data is "
    << mydemo.int_data << "\n";
  std::cout << "Float data is "
    << mydemo.float_data << "\n";
}
```

- This is how the constructor is used
- When you declare the instance of the class put **{ }** after it and the values of the parameters to the constructor
- You can also put **( )** rather than **{ }**, but once again uniform initialisation is more unambiguous
- There are a lot of strange ambiguities in C++ and uniform initialisation was designed to try and fix them - we'd advise using it

Constructors

```
#include <iostream>
class demo{
public:
int int_data {14};
float float_data{1234.56};
demo(int i, float f)
  {int_data=i;float_data=f;}
demo()=default;
};

int main(){
  demo mydemo{6,5.6};

  std::cout << "Int data is "
    << mydemo.int_data << "\n";
  std::cout << "Float data is "
    << mydemo.float_data << "\n";
}
```

- This is the default constructor and is used when you don't pass any parameters when creating and instance
- When you create a non-default constructor (like our one taking an int and a float) then you **delete** the automatic default constructor so you have to put it back manually if you still want the default behaviour
- If you just want the default behaviour then you can put **=default** here, but if you want parameterless construction to do something then you can implement it like a normal function or constructor

Destructors

- The opposite of a **constructor** is a **destructor**
- Destructors are called when an object is destroyed and should release any resources that the object owns that need to be manually released
- Destructors never take parameters and are defined as
  - **~classname(){//Put destruction code here}**
- In modern C++ there aren't any simple reasons for wanting destructors so we're not really giving any proper examples

Philosophy

- Construction is your opportunity to gather resources etc. that your class needs
- Destruction is then your opportunity to release resources
- Destructors are called separately for everything, so every member variable of your class will have it's destructor called automatically if they have one
- Note that most C++ built in "things" have their own destructors
- So if your class uses a **std::vector** to store objects or a **std::ifstream** to read from a file then you don't need to do anything to clean them up
- They will be automatically be cleaned up when your class is destroyed

Other constructors

- There are two special constructors that you should know about
- **copy constructors** are used when you initialise one instance of a class from another instance of a class
- **move constructors** are used when you initialise an instance of a class from a temporary instance of a class, such as a literal or the return from a function
- Default versions of move and copy constructors are created for you, but writing certain other things can cause them to be deleted just like the default constuctor
  - When this happens you'll have to write them manually
- Be careful about assuming that copy or move constructors will definitely be called - it is permissible in C++ for things that look like moves or copies to be optimised away!

Other special methods

- There are one more common class of special methods - **operators**
- Operators are methods that are called when the class has an operator called on it
- For example, you can implement **operator+** to allow you to add things to your class by using the **+** operator, just like you would do for numbers
- You can also implement more esoteric operators like **operator()** which allows you to call your class instance like a function (these are often called **functors**)
- You can also implement **operator[]** which allows you to access elements like an array - this is how **std::vector** works when you access it using **[ ]**

Other special methods

- You can do a lot with operators
- For example, you can implement **operator+** for any type that you want, so that you can add integers or floats to your class
  - Generally be careful though! If you provide any mathematical operators then a developer will expect you to provide all of them for all sensible types.
- Similarly, you don't **have** to require that **operator[]** takes a single integer like array subscription does
- Be aware that if you break conventions of the language like **[ ]** taking a single integer you might confuse people!

Templated Classes

Template Classes

- You can template classes much as you can template functions
- You now **have** to specify the types of the template parameter in **< >** because automatic inference is not possible
- The template applies to the whole class and you can use the type parameters anywhere
  - In member variables, in method descriptions, etc.
- It is also possible to individually template methods separately to the entire class but if you are reaching that point you are into fairly advanced things

Template Classes

```
#include <iostream>
#include <vector>

template <typename T>
class datastore{
  public:
  T item;
  std::vector<T> subitems;
};

int main(){
  datastore<int> d;
  d.item = 14;
  d.subitems.push_back(17);
  d.subitems.push_back(18);

  std::cout << "Item ID is " << d.item << "\n";
  std::cout << "Sub items are " << "\n";
  for (auto &element:d.subitems){
    std::cout << element << "\n";
  }
}
```

- **template <typename T>** as before
- You can now use T when defining any members of your class
- Create an instance of your struct as before, but now with **< >** to explicitly specify the template parameter
- Looks exactly like **std::vector**