Object-Oriented Programming in C++

Pre-Lecture 5: More about Classes

Prof Niels Walet (Niels.Walet@manchester.ac.uk)

Room 7.07, Schuster Building

Version: February 15, 2020

Prelecture 5

Outline

In today's Prelecture we will improve our use of classes. We shall introduce more of their features, and thus be able to make fuller use of what they can do. We shall look at:

- Using `const` in function parameters
- Passing objects to functions
- Returning objects from functions
- Overloading operators
- Friends of classes

Refresher and Introduction

Class example

A graduated example

We shall first work our way through a detailed example, to illustrate a few of the ideas we have already seen, at the same time increasing our understanding....

Class example

Give the class a name

```
1  // PL5/aclass.cpp
2  // A populated class for 3 vectors
3  // Niels Walet, last updated 09/02/2020
4  #include<iostream>
5  class vector3
6  {
7  private:
8
9
10
11 public:
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27 };
```

Listing 1 : PL5/aclass1.cpp

Class example

Add private data

```
1  // PL5/aclass.cpp
2  // A populated class for 3 vectors
3  // Niels Walet, last updated 09/02/2020
4  #include<iostream>
5  class vector3
6  {
7  private:
8    double x{};
9    double y{};
10   double z{};
11 public:
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27 };
```

Listing 2 : PL5/aclass2.cpp

Class example

Add constructor(s) and destructor

```
1  // PL5/aclass.cpp
2  // A populated class for 3 vectors
3  // Niels Walet, last updated 09/02/2020
4  #include<iostream>
5  class vector3
6  {
7  private:
8    double x{};
9    double y{};
10   double z{};
11 public:
12   // Constructors and destructor
13   vector3() = default ;
14   vector3(double x_in, double y_in, double z_in) : x{x_in}, y{y_in}, z{z_in} {}
15   ~vector3(){std::cout<<"Destroying vector"<<std::endl;}
16
17
18
19
20
21
22
23
24
25
26
27 };
```

Listing 3 : PL5/aclass3.cpp

Class example

Add accessor and mutator functions

```
// PL5/aclass.cpp
// A populated class for 3 vectors
// Niels Walet, last updated 09/02/2020
#include<iostream>
class vector3
{
private:
  double x{};
  double y{};
  double z{};
public:
  // Constructors and destructor
  vector3() = default ;
  vector3(double x_in, double y_in, double z_in) : x{x_in}, y{y_in}, z{z_in} {}
  ~vector3(){std::cout<<"Destroying vector"<<std::endl;}
  // Access functions to set and get vector components
  void set_x(const double x_in) {x=x_in;}
  void set_y(const double y_in) {y=y_in;}
  void set_z(const double z_in) {z=z_in;}
  double get_x() const {return x;}
  double get_y() const {return y;}
  double get_z() const {return z;}
};
```

Listing 4 : PL5/aclass4.cpp

Class example

Add some useful functions

```cpp
// PL5/aclass.cpp
// A populated class for 3 vectors
// Niels Walet, last updated 09/02/2020
#include<iostream>
class vector3
{
private:
  double x{};
  double y{};
  double z{};
public:
  // Constructors and destructor
  vector3() = default ;
  vector3(double x_in, double y_in, double z_in) : x{x_in}, y{y_in}, z{z_in} {}
  ~vector3(){std::cout<<"Destroying vector"<<std::endl;}
  // Access functions to set and get vector components
  void set_x(const double x_in) {x=x_in;}
  void set_y(const double y_in) {y=y_in;}
  void set_z(const double z_in) {z=z_in;}
  double get_x() const {return x;}
  double get_y() const {return y;}
  double get_z() const {return z;}
  // Function to print out vector
  void show() const {std::cout<<"("<<x<<","<<y<<","<<z<<")"<<std::endl;}
  // Function to add a scalar to each vector component
  void add_scalar(const double s) {x+=s; y+=s; z+=s;}
};
```

Listing 5 : PL5/aclass.cpp

Class example

Apply:

```
28 int main()
29 {
30   // Define 3 vectors
31   vector3 a;
32   vector3 b{1,2,3};
33   vector3 c{-1,-2,-3};
34   // Print vectors
35   a.show();
36   b.show();
37   c.show();
38   // Add a scalar to each vector component
39   double s{-1.5};
40   b.add_scalar(s);
41   b.show();
42   return 0;
43 }
```

Listing 6 : selection of PL5/aclassfull.cpp

results in

```
(0,0,0)
(1,2,3)
(-1,-2,-3)
(-0.5,0.5,1.5)
Destroying vector
Destroying vector
Destroying vector
```

constantness

Class extensions

Using const in function parameters

- We first introduced the `const` modifier when defining constant data types [e.g. `const double msun_in_kg{1.989e30}`]
- The `const` modifier can also be used in function parameter lists
- Example 1: passing by reference
  `void my_member_function(const &my_object)`
  - Guarantees argument cannot be modified inside function
  - Useful when passing by reference to speed things up
- Example 2: applicable to member functions of a class
  `void my_member_function(double my_double) const`
  - This use is not quite so clear! It denotes **immutability**
  - It guarantees member data are not modified
- Use `const` wherever appropriate: gives even more data protection

passing objects

Class extensions

Passing objects to functions

- We can pass our newly-defined objects to other functions
- Example: a function to calculate the dot product of two vectors

```
double dot_product(const vector3 &v1, const vector3 &v2)
{
  double result =
    v1.get_x()*v2.get_x() +
    v1.get_y()*v2.get_y() +
    v1.get_z()*v2.get_z();
  return result;
}
```

- Our `vector3` objects are passed in the same manner as other data types
- Note these have been passed by reference (more efficient)
- We have used the public access functions of `v1` and `v2` to get their components

Class extensions

Passing objects to functions

- We can add the following lines to `main` in our previous code

```
50  // Dot product using normal function (passing two vectors)
51   double dp=dot_product(b,c);
52   std::cout<<"Dot product b.c = "<<dp<<std::endl;
```

Listing 7 : selection of PL5/aclassfull2.cpp

- This does the job, but the function is defined outside of the `vector3` class
- Hardly need classes for this– this is just old-fashioned procedural programming!
- Since we are acting on the object's member data, much better to include the function as part of the class!
- Once again highlights the key feature of OOP: encapsulation

Class extensions

Passing objects to functions

- Inside the class, we add the following (public) function

```
double dot_product(const vector3 &v) const
{
  return (x*v.x+y*v.y+z*v.z);
}
```

- This looks a little bit different: only one argument is needed
- Since the function is a member of the `vector3` class, data for one of the vectors already belongs to the object calling the function (member data)
- Easier to see by example: we now calculate the dot product as

```
47  double dp=b.dot_product(c);
```

Listing 8 : selection of PL5/aclassfull3.cpp

- Here `b` is the object calling the function and `c` is argument of the function

Class extensions

Returning objects from functions

- Our `vector3` class name is now also a valid return type
- Example: add a member function to add two vectors together

```
vector3 plus(const vector3 &v) const
{
  vector3 temp;
  temp.set_x(x+v.x);
  temp.set_y(y+v.y);
  temp.set_z(z+v.z);
  return temp;
}
```

- We can then add the following lines in `main` to demonstrate use of this function

```
49  vector3 d=b.plus(c);
50  std::cout<<"sum of b and c is ";
51  d.show();
```

Listing 9 : selection of PL5/aclassfull4.cpp

Class extensions

Returning objects from functions

- Again, here `b` is the object calling the function and `c` is argument of the function
- Rather than modify the data of `b` we declared a new vector (`temp`) inside the function and set its components using the access functions
- Note we have used the (default) constructor of `vector3` from within one of its own member functions!
- Alternatively, we can use the parameterized constructor

```
33   vector3 plus(const vector3 &v) const
34   {
35     vector3 temp{x+v.x,y+v.y,z+v.z};
36     return temp;
37   }
```

Listing 10 : selection of PL5/aclassfull4a.cpp

operator overloading

Class extensions

operator overloading

- This is a useful (but controversial) feature of C++, really extending its functionality
- Particularly useful when our objects are mathematical constructs (e.g. vectors, complex numbers)
- An example: we can overload the function of the + operator to use our `vector3` class
- In the class itself, we add the following function

```
29  vector3 operator+(const vector3 &v) const
30  {
31    vector3 temp{x+v.x,y+v.y,z+v.z};
32    return temp;
33  }
```

Listing 11 : selection of PL5/aclassfinal.cpp

Overloading operators:

adding vectors

- You may then be tempted to write the following
  `vector3 e{b.operator+(c)}; e.show();`
- And you would be right! This would indeed work and produce the same result as our `plus` function
- But instead, how about we write `vector3 e{b+c}; e.show();`
- This gives the same result!
- We have overloaded the addition (+) operator to include our vectors

Overloading operators:

adding a vector and a scalar

- We can also overload the + operator with other arguments
- Example: right addition of a scalar to a vector

```
35   vector3 operator+(const double scalar) const
36   {
37     vector3 temp{x+scalar,y+scalar,z+scalar};
38     return temp;
39   }
```

Listing 12 : selection of PL5/aclassfinal.cpp

- This can then called in as follows `vector3 f(e+1.5); f.show();`
- Can overload most operators, e.g +,-,*,/,+=,[]...
- But be careful to use a sensible *meaning* for these operators!

Overloading operators:

a limitation

- Overloading using a member function always requires an object (`vector3`) to be on the left of the (e.g., `+`) operator.
- What if we want to left-add a scalar? I.e. `g=1.5+e`.
- We can not use a member function of the object `e` as it appears on the right of the expression!
- There is a way round: can overload a *non-member* function instead

```
// Non-member function to left-add scalar to vector
   vector3 operator+(double scalar, const vector3 &v) {
      vector3 temp;
      temp.set_x(scalar+v.get_x());
      temp.set_y(scalar+v.get_y());
      temp.set_z(scalar+v.get_z());
      return temp;
   }
```

- Relevant whenever class of object on the left (e.g., in this case the built-in `double`) cannot be modified.

friends

Class extensions

Friends of classes

- Another controversial one–I like it, but some coding styles think this is bad:
- The previous non-member function requires the use of access functions to get the vector's co-ordinates (which were private data members).
- No way to make the function a member function.
- Problem: function is separate from class (one of the main features of OOP is encapsulation: including all data and functions acting on that data in one class).
- Solution: we can make a non-member function a friend of the class.
- Functions that are friends are ordinary functions, but can access member data of "friendly" object(s) in parameter list instead.

Class extensions

Friends of classes

- For our left-add function we can put following line in class
  `friend vector3 operator+(double scalar, const vector3 &v);`
- This declares the function as a friend of the class - private access permission is granted for member data of object parameter
- Our function can now look like

```
// Friend function to left-add scalar to vector
vector3 operator+(double scalar, const vector3 &v) {
   vector3 temp(scalar+v.x,scalar+v.y,scalar+v.z);
   return temp;
}
```

- Since the function is now a friend of `vector3`, we can directly access the private data belonging to the object, `v`
- Procedure is similar to putting our member functions outside the class (always need to include declaration in the class)
- Good practice to group both types of functions together

summary

Class extensions

Summary

We have discussed the following ideas:

- Using `const` in function parameters
- Passing objects to functions
- Returning objects from functions
- Overloading operators
- Friends of classes

Look at the aclassfinal.cpp code for an example using all of these features.