Lecture 31 — Member Functions, Namespaces, Equality

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 30, 2016

Acknowledgments: W.D. Bishop

Part I

Member Functions

Scope Resolution Operator

Thus far we have made some extensive use of the Scope Resolution Operator (`::`) but not explained how it works.

In C++, unlike some other languages, the implementations of member functions is usually found outside the class definition.

It is similar to the dot operator (`.`) in that it is used to "go inside" the thing on the left side.

Scope Resolution Operator

So, to implement a function defined in a class `Coordinates` such as `void output()`:

```
void Coordinates::output( ) {
    cout << "(" << x << "," << y << ")" << endl;
}
```

Note that private variables can be used inside the implementation.

Part II

Namespaces

Namespaces

A namespace is a collection of associated types.

Namespaces often correspond with packages, libraries, or application programming interfaces (APIs).

Namespaces permit code reuse by ensuring that types defined in one library do not conflict with those defined in another library.

Thus we could define two distinct classes called `Coordinates` and differentiate between them by putting them in different namespaces:

`Real::Coordinates` may represent (x, y)-coordinates;
`Polar::Coordinates` may represent polar (r, θ)-coordinates.

The `using` Keyword

The `using` keyword can be used to eliminate the need for specifying the full name of a type or a function.

For example, the statement `using namespace std;` allows a C++ program to use `std::cout` as `cout`

The scope of a `using` directive is limited to the file in which it appears; each file needs to specify its own `using` namespaces.

Part III

Equality

Equality

When using equality operators, == and `!=`, we have thus far used them on simple types like `int`; `7 == 0` evaluates to `false`.

When it comes to classes, however, things don't always behave as we would expect.

What does it mean for two `Coordinates` objects to be equal?

We would expect it means the x and y values are the same.

Default Behaviour

The default behaviour of the == operator does not do what we want.

One approach would be to use a function for this: `equals`.

Use: `c1.equals(c2);`

This function and returns true if they are equal and false if they are not.

That works, but why can't we use the == operator?

Yes We Can!

Actually we can.

Just as a function can be overloaded, we may overload an operator to make use of programmer defined types.

The syntax is:

```
bool operator == ( const Coordinates& lhs, const
Coordinates& rhs );
```

The arguments are “left hand side” and “right hand side” respectively.
For equals it doesn’t matter, but for subtraction it would!

Implementation of Operator Overloading

```
bool operator ==(const Coordinates& lhs, const Coordinates& rhs)
{
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
```

It is really that simple!

Overloading Other Operators

Many other operators can be overloaded, particularly mathematical operators.

This allows us to use natural looking syntax for what might otherwise be a difficult to read operation.

For example, we could define all mathematical operators for complex numbers, thus allowing us to add them easily with the + operator.

As Uncle Ben said...

Operator overloading is a very complex topic and much of it is beyond the scope of the course.

It is something that provides a “nice” syntax when used correctly but can lead to incredibly difficult to solve errors when abused.

Potential pitfall: C++ has stack and heap allocation and an operator must be written in such a way that it does not cause memory leaks.

This is, fortunately, not applicable in the case of an equality test.