Objects and classes, advanced topics

Victor Eijkhout, Susan Lindsey

Fall 2023

`last formatted: November 3, 2023`

Operator overloading

1. Better syntax

Operations that 'feel like arithmetic'"

So far:

```
Point p3 = p1.add(p2);
Point p4 = p3.scale(2.5);
```

Improved:

```
Point p3 = p1+p2;
Point p4 = p3*2.5;
```

This is possible because you can *overload* the *operators*. For instance,

```
// geom/overload.cpp
  Point operator*( float f ) {
    return Point( x*f,y*f );
  }
```

2. Operator overloading

Syntax:

```
<returntype> operator<op>( <argument> ) { <definition> }
```

For instance:

Code:

```
// geom/pointscale.cpp
Point Point::operator*(double f) {
    return Point(f*x,f*y);
};
    /* ... */
  cout << "p1 to origin "
       << p1.dist_to_origin() <<
     '\n';
  Point scale2r = p1*2.;
  cout << "scaled right: "
       << scale2r.dist_to_origin()
     << '\n';
  // ILLEGAL Point scale2l = 2.*p1;
```

Output:

```
p1 to origin 2.23607
scaled right: 4.47214
```

Can also:

```
void Point::operator*=(double factor);
```

Exercise 1

Define the plus operator between `Point` objects. The declaration is:

```
Point operator+(Point q);
```

You can base this off the file `overload.cpp` in the repository

Exercise 2

Revisit exercise **??** and replace the *add* and *scale* functions by overloaded operators.

Hint: for the *add* function you may need '`this`'.

3. Functor example

Simple example of overloading parentheses:

Code:

```
// object/functor.cpp
class IntPrintFunctor {
public:
  void operator()(int x) {
    cout << x << '\n';
  }
};
    /* ... */
  IntPrintFunctor intprint;
  intprint(5);
```

Output:

```
5
```

Exercise 3

Evaluate a linear function:

Using method:

```
// geom/overload.cpp
  LinearFunction line(p1,p2);
  cout << "Value at 4.0: "
       << line.evaluate_at(4.0)
       << '\n';
```

using operator:

```
// geom/overload.cpp
  float y = line(4.0);
  cout << y << '\n';
```

Write the appropriate overloaded operator.

You can base this off the file `overload.cpp` *in the repository*

Inherit from containers

4. Has-a std container

You could write

```
class namedvec {
private:
    std::string name;
    std::vector<double> contents;
public:
    namedvec( std::string n,int s );
    // ...
};
```

The problem now is that for every vector method, `at`,`size`,`push_back`, you have to re-implement that for your `namedvec`.

5. Inherit from vector

Named vector inherits from standard vector:

```
// object/container.cpp
#include <vector>
#include <string>
class namedvector
  : public std::vector<int> {
private:
  std::string _name;
public:
  namedvector( std::string
     n,int s )
     :
     _name(n),std::vector<int>(s)
     {};
  auto name() { return _name;
     };
};
```

```
// object/container.cpp
  namedvector
    fivevec("five",5);
  cout << fivevec.name()
       << ": " <<
    fivevec.size() << '\n';
  cout << "at zero: "
       << fivevec.at(0) <<
    '\n';
```

Extend the code from 11 to make *namedvector* templated.

```
// object/container.cpp
  temptvector<int> fivetemp("five",5);
  cout << fivetemp.name()
       << ": " << fivetemp.size() << '\n';
  cout << "at zero: "
       << fivetemp.at(0) << '\n';
```

Extend the code from 11 and to make a namespaced class *geo::vector* that has the functionality of *namedvector*.

```
// object/container.cpp
  using namespace geo;
  geo::vector<int> fivegeo("five",5);
  cout << fivegeo.name()
       << ": " << fivegeo.size() << '\n';
  cout << "at zero: "
       << fivegeo.at(0) << '\n';
```

Internal access

6. Direct alteration of internals

Return a reference to a private member:

```
class Point {
private:
  double x,y;
public:
  double &x_component() { return x; };
};
int main() {
  Point v;
  v.x_component() = 3.1;
}
```

Only define this if you need to be able to alter the internal entity.

7. Reference to internals

Returning a reference saves you on copying.

Prevent unwanted changes by using a 'const reference'.

```
class Grid {
private:
  vector<Point> thepoints;
public:
  const vector<Point> &points() const {
    return thepoints; };
};
int main() {
  Grid grid;
  cout << grid.points()[0];
  // grid.points()[0] = whatever ILLEGAL
}
```

8. Access gone wrong

We make a class for points on the unit circle

```
// object/unit.cpp
class UnitCirclePoint {
private:
  float x,y;
public:
  UnitCirclePoint(float x) {
    setx(x); };
  void setx(float newx) {
    x = newx; y = sqrt(1-x*x);
  };
```

You don't want to be able to change just one of x,y!

In general: enforce invariants on the members.

9. Const functions

A function can be marked as const:
it does not alter class data,
only changes are through return and parameters

10. 'this' pointer to the current object

```
class Myclass {
private:
  int myint;
public:
  Myclass(int myint) {
    this->myint = myint; // `this' redundant!
  };
};
```

11. 'this' use

You don't often need the this pointer. Example: you need to call a function inside a method that needs the object as argument)

```
/* forward definition: */ class someclass;
void somefunction(const someclass &c) {
  /* ... */ }
class someclass {
// method:
void somemethod() {
  somefunction(*this);
};
```

(Rare use of dereference star)

Headers

12. C headers plusplus

You know how to use `.h` files in C.

Classes in C++ need some extra syntax.

13. Data members in proto

Data members, even private ones, need to be in the header file:

```
class something {
private:
  int localvar;
public:
  // declaration:
  double somedo(vector);
};
```

Implementation file:

```
// definition
double something::somedo(vector v) {
   .... something with v ....
   .... something with localvar ....
};
```

14. Static class members

A static member acts as if it's shared between all objects.

(Note: C++17 syntax)

Code:

```
// link/static17.cpp
class myclass {
private:
  static inline int count=0;
public:
  myclass() { ++count; };
  int create_count() {
    return count; };
};
    /* ... */
  myclass obj1,obj2;
  cout << "I have defined "
       << obj1.create_count()
       << " objects" << '\n';
```

Output:

```
I have defined 2 objects
```

15. Static class members, C++11 syntax

```
// link/static.cpp
class myclass {
private:
  static int count;
public:
  myclass() { ++count; };
  int create_count() { return count; };
};
    /* ... */
// in main program
int myclass::count=0;
```