Objects and classes, advanced topics

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: November 7, 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:

```
1  // geom/pointscale.cpp
2  Point Point::operator*(double f) {
3      return Point(f*x,f*y);
4  };
5      /* ... */
6    cout << "p1 to origin "
7         << p1.dist_to_origin() <<
         '\n';
8    Point scale2r = p1*2.;
9    cout << "scaled right: "
10        << scale2r.dist_to_origin()
       << '\n';
11   // ILLEGAL Point scale2l = 2.*p1;
```

Output:

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

Can also:

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:

```
1 // object/functor.cpp
2 class IntPrintFunctor {
3 public:
4   void operator()(int x) {
5     cout << x << '\n';
6   }
7 };
8     /* ... */
9   IntPrintFunctor intprint;
10  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. What is the problem?

You want a `std::vector` but with some added functionality.

```
// proposed construct call:
namedvec<float> x("xvec",5);
// proposed usage:
x.size();
x.name();
x[4];
```

5. 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`.

6. Inherit from vector

Named vector inherits from standard vector:

```
// object/container0.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/container0.cpp
  namedvector
    fivevec("five",5);
  cout << fivevec.name()
       << ": "
       << fivevec.size()
       << '\n';
  cout << "at zero: "
       << fivevec.at(0)
       << '\n';
```

Exercise 4

Extend the code from 12 to make *namedvector* templated.

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

Exercise 5

Extend the code from 12 and 4 to make a namespaced class *geo::vector* that has the functionality of *namedvector*.

```
// object/container.cpp
  using namespace geo;
  geo::vector<float> float4("four",4);
  cout << float4.name() << '\n';
  float4[1] = 3.14;
  cout << float4.at(1) << '\n';
  geo::vector<std::string> string3("three",3);
  string3.at(2) = "abc";
  cout << string3[2] << '\n';
```

Internal access

7. Direct alteration of internals

Return a reference to a private member:

```
1 class Point {
2 private:
3   double x,y;
4 public:
5   double &x_component() { return x; };
6 };
7 int main() {
8   Point v;
9   v.x_component() = 3.1;
10 }
```

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

8. Reference to internals

Returning a reference saves you on copying.
Prevent unwanted changes by using a 'const reference'.

```
1  class Grid {
2  private:
3    vector<Point> thepoints;
4  public:
5    const vector<Point> &points() const {
6      return thepoints; };
7  };
8  int main() {
9    Grid grid;
10   cout << grid.points()[0];
11   // grid.points()[0] = whatever ILLEGAL
12 }
```

9. Access gone wrong

We make a class for points on the unit circle

```
1  // object/unit.cpp
2  class UnitCirclePoint {
3  private:
4    float x,y;
5  public:
6    UnitCirclePoint(float x) {
7      setx(x); };
8    void setx(float newx) {
9      x = newx; y = sqrt(1-x*x);
10   };
```

You don't want to be able to change just one of x, y!
In general: enforce invariants on the members.

10. Const functions

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

11. ‘this’ pointer to the current object

```
1 class Myclass {
2 private:
3   int myint;
4 public:
5   Myclass(int myint) {
6     this->myint = myint; // `this' redundant!
7   };
8 };
```

12. '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)

```
1 /* forward definition: */ class someclass;
2 void somefunction(const someclass &c) {
3   /* ... */ }
4 class someclass {
5 // method:
6 void somemethod() {
7   somefunction(*this);
8 };
```

(Rare use of dereference star)

Headers

13. C headers plusplus

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

Classes in C++ need some extra syntax.

14. Data members in proto

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

```
1 class something {
2 private:
3   int localvar;
4 public:
5   // declaration:
6   double somedo(vector);
7 };
```

Implementation file:

```
1 // definition
2 double something::somedo(vector v) {
3     .... something with v ....
4     .... something with localvar ....
5 };
```

15. Static class members

A static member acts as if it's shared between all objects.
(Note: C++17 syntax)

Code:

```
1 // link/static17.cpp
2 class myclass {
3 private:
4   static inline int count=0;
5 public:
6   myclass() { ++count; };
7   int create_count() {
8     return count; };
9 };
10    /* ... */
11  myclass obj1,obj2;
12  cout << "I have defined "
13       << obj1.create_count()
14       << " objects" << '\n';
```

Output:

```
I have defined 2 objects
```

16. Static class members, C++11 syntax

```
1 // link/static.cpp
2 class myclass {
3 private:
4   static int count;
5 public:
6   myclass() { ++count; };
7   int create_count() { return count; };
8 };
9     /* ... */
10 // in main program
11 int myclass::count=0;
```