Advanced Objects

Victor Eijkhout, Susan Lindsey

Fall 2025

last formatted: September 25, 2025

Operator overloading

1. Operator overloading

Syntax:

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

For instance:

Code:

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

Output:

```
1 p1 to origin 2.2361
2 scaled right:
      ↪4.472136
```

Exercise 1

Rewrite the `halfway` method of exercise **??** and replace the `add` and `scale` functions by overloaded operators.

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

2. Constructors and contained classes

Finally, if a class contains objects of another class,

```
1 class Inner {
2 public:
3   Inner(int i) { /* ... */ }
4 };
5 class Outer {
6 private:
7   Inner contained;
8 public:
9 };
```

3. When are contained objects created?

```
1 Outer( int n ) {
2   contained = Inner(n);
3 };
```

1. This first calls the default constructor
2. then calls the `Inner(n)` constructor,
3. then copies the result over the `contained` member.

```
1 Outer( int n )
2   : contained(Inner(n)) {
3     /* ... */
4 };
```

1. This creates the `Inner(n)` object,
2. placed it in the `contained` member,
3. does the rest of the constructor, if any.

4. Copy constructor

- Default defined copy and 'copy assignment' constructors:

```
some_object x(data);
some_object y = x;
some_object z(x);
```

- They copy an object:
  - simple data, including pointers
  - included objects recursively.
- You can redefine them as needed.

```
// object/copyscalar.cpp
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v
         << '\n';
    mine = v; };
  has_int( has_int &h ) {
    auto v = h.mine;
    cout << "copy: " << v
         << '\n';
    mine = v; };
  void printme() {
    cout << "I have: " << mine
         << '\n'; };
};
```

5. Copy constructor in action

Code:

```
1 // object/copyscalar.cpp
2 has_int an_int(5);
3 has_int other_int(an_int);
4 an_int.printme();
5 other_int.printme();
6 has_int yet_other = other_int;
7 yet_other.printme();
```

Output:

```
1 set: 5
2 copy: 5
3 I have: 5
4 I have: 5
5 copy: 5
6 I have: 5
```

6. Copying is recursive

Class with a vector:

```
1 // object/copyvector.cpp
2 class has_vector {
3 private:
4   vector<int> myvector;
5 public:
6   has_vector(int v) { myvector.push_back(v); };
7   void set(int v) { myvector.at(0) = v; };
8   void printme() { cout
9       << "I have: " << myvector.at(0) << '\n'; };
10 };
```

Copying is recursive, so the copy has its own vector:

Code:

```
1 // object/copyvector.cpp
2 has_vector a_vector(5);
3 has_vector other_vector(a_vector);
4 a_vector.set(3);
5 a_vector.printme();
6 other_vector.printme();
```

Output:

```
1 I have: 3
2 I have: 5
```

7. Destructor

- Every class *myclass* has a *destructor* *~myclass* defined by default.
- The default destructor does nothing:

```
1 ~myclass() {};
```

- A destructor is called when the object goes out of scope. Great way to prevent memory leaks: dynamic data can be released in the destructor. Also: closing files.

8. Destructor example

Just for tracing, constructor and destructor do `cout`:

```
1 // object/destructor.cpp
2 class SomeObject {
3 public:
4   SomeObject() {
5     cout << "calling the constructor"
6          << '\n';
7   };
8   ~SomeObject() {
9     cout << "calling the destructor"
10         << '\n';
11  };
12 };
```

9. Destructor example

Destructor called implicitly:

Code:

```
1 // object/destructor.cpp
2 cout << "Before the nested scope"
3      << '\n';
4 {
5   SomeObject obj;
6   cout << "Inside the nested scope"
7        << '\n';
8 }
9 cout << "After the nested scope"
10     << '\n';
```

Output:

```
1 Before the nested
      ↪scope
2 calling the
      ↪constructor
3 Inside the nested
      ↪scope
4 calling the
      ↪destructor
5 After the nested
      ↪scope
```

References and such

10. 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.

11. 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 }
```

12. 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.

13. Const functions

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

14. 'this' pointer to the current object

A pointer to the object itself is available as `this`. Variables of the current object can be accessed this way:

```
1 class Myclass {
2 private:
3   int myint;
4 public:
5   Myclass(int myint) {
6     this->myint = myint;   // option 1
7     (*this).myint = myint; // option 2
8   };
9 };
```

15. '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

16. C headers plusplus

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

Classes in C++ need some extra syntax.

17. 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 };
```

18. 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:

```
1 I have defined 2
      ↪objects
```

19. 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;
```