More Objects

Victor Eijkhout, Susan Lindsey

Fall 2025
last formatted: September 24, 2025

Interaction between objects

1. Methods that create a new object

Code:

```
// geom/pointscale.cpp
class Point {
    /* ... */
  Point scale( float a ) {
    Point scaledpoint( x*a, y*a );
    return scaledpoint;
  };
    /* ... */
  println("p1 to origin {:.5}",
          p1.dist_to_origin());
  Point p2 = p1.scale(2.);
  println("p2 to origin {:.5}",
          p2.dist_to_origin());
```

Output:

```
p1 to origin 2.2361
p2 to origin 4.4721
```

2. Anonymous objects

Create a point by scaling another point:

```
new_point = old_point.scale(2.81);
```

Two ways of handling the `return` statement of the `scale` method:

Naive:

```
// geom/pointscale.cpp
Point Point::scale( float a ) {
  Point scaledpoint =
    Point( x*a, y*a );
  return scaledpoint;
};
```

Concise:

```
// geom/pointscale.cpp
Point Point::scale( float a ) {
  return Point( x*a, y*a );
};
```

'move semantics' and 'copy elision':
compiler is pretty good at avoiding copies

Exercise 1

Write a method `halfway` that, given two `Point` objects `p,q`, construct the `Point` halfway, that is, $(p+q)/2$:

```
Point p(1,2.2), q(3.4,5.6);
Point h = p.halfway(q);
```

You can write this function directly, or you could write functions `Add` and `Scale` and combine these.
(Later you will learn about operator overloading.)

How would you print out a `Point` to make sure you compute the halfway point correctly?

3. Using the default constructor

No constructor explicitly defined;

You recognize the default constructor in the main by the fact that an object is defined without any parameters.

Code:

```
// object/default.cpp
class IamOne {
private:
  int i=1;
public:
  void print() {
    cout << i << '\n';
  };
};
    /* ... */
  IamOne one;
  one.print();
```

Output:

```
1
```

4. Default constructor

Refer to `Point` definition above.
Consider this code that looks like variable declaration, but for objects:

```
Point p1(1.5, 2.3);
Point p2;
p2 = p1.scaleby(3.1);
```

Compiling gives an error (g++; different for intel):

```
pointdefault.cpp: In function 'int main()':
pointdefault.cpp:32:21: error: no matching function for call to
                'Point::Point()'
```

5. Default constructor

The problem is with *p2*:

```
Point p1(1.5, 2.3);
Point p2;
```

- *p1* is created with your explicitly given constructor;
- *p2* uses the default constructor:

```
Point() {};
```

- default constructor is there by default, unless you define another constructor.
- you can re-introduce the default constructor:

```
// geom/pointdefault.cpp
Point() = default;
Point( float x,float y )
  : x(x),y(y) {};
```

(but often you can avoid needing it)

6. Other way

State that the default constructor exists with the `default` keyword:

```
// object/default.cpp
Point() = default;
Point( double x,double y )
  : x(x),y(y) {};
```

State that there should be no default constructor with the `delete` keyword:

```
Point() = delete;
```

Exercise 2

Make a class *LinearFunction* with a constructor:

```
LinearFunction( Point input_p1,Point input_p2 );
```

and a member function

```
float evaluate_at( float x );
```

which you can use as:

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

7. Classes for abstract objects

Objects can model fairly abstract things:

Code:

```
// object/stream.cpp
class Stream {
private:
  int last_result{0};
public:
  int next() {
    return last_result++; };
};

int main() {
  Stream ints;
  println( "Next: {}",
    ints.next() );
  println( "Next: {}",
    ints.next() );
  println( "Next: {}",
    ints.next() );
```

Output:

```
Next: 0
Next: 1
Next: 2
```

8. Preliminary to the following exercise

A prime number generator has:
an API of just one function: `nextprime`

To support this it needs to store:
an integer `last_prime_found`

Programming Project Exercise 3

Write a class *primegenerator* that contains:

- Methods *number_of_primes_found* and *nextprime*;
- Also write a function *isprime* that does not need to be in the class.

Your main program should look as follows:

```
// primes/6primesbyclass.cpp
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << '\n';
}
```

Programming Project Exercise 4

Write a program to test the Goldbach conjecture for the even numbers up to a bound that you read in.

First formulate the quantor structure of this statement, then translate that top-down to code, using the generator you developed above.

1. Make an outer loop over the even numbers e.
2. For each e, generate all primes p.
3. From $p + q = e$, it follows that $q = e - p$ is prime: test if that q is prime.

For each even number `e` then print `e,p,q`, for instance:

```
The number 10 is 3+7
```

If multiple possibilities exist, only print the first one you find.

9. A Goldbach corollary

The Goldbach conjecture says that every even number $2n$ (starting at 4), is the sum of two primes $p+q$:

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes:

$$n = \frac{p+q}{2} \qquad \text{or} \qquad q - n = n - p.$$

In particular this holds for each prime number:

$$\forall_{r\,\mathrm{prime}} \exists_{p,q\,\mathrm{prime}} \colon r = (p+q)/2 \text{ is prime.}$$

We now have the statement that each prime number is the average of two other prime numbers.

Programming Project Exercise 5

Write a program that tests this. You need at least one loop that tests all primes r; for each r you then need to find the primes p, q that are equidistant to it.

Use your prime generator. Do you use two generators for this, or is one enough? Do you need three, for p, q, r?

For each r value, when the program finds the p, q values, print the p, q, r triple and move on to the next r.

Advanced stuff

10. 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.

11. 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
}
```

12. 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.

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:

```
class Myclass {
private:
  int myint;
public:
  Myclass(int myint) {
    this->myint = myint;   // option 1
    (*this).myint = myint; // option 2
  };
};
```

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)

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

(Rare use of dereference star)

Operator overloading

16. Operator overloading

Syntax:

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

For instance:

Code:

```
// geom/pointscale.cpp
Point Point::operator*(float f) {
    return Point(f*x,f*y);
};
    /* ... */
  println("p1 to origin {:.5}",
          p1.dist_to_origin());
  Point scale2r = p1*2.;
  println("scaled right: {}",
          scale2r.dist_to_origin());
  // ILLEGAL Point scale2l = 2.*p1;
```

Output:

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

Exercise 6

Rewrite the `halfway` method of exercise 1 and replace the `add` and `scale` functions by overloaded operators.

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

17. Constructors and contained classes

Finally, if a class contains objects of another class,

```
class Inner {
public:
  Inner(int i) { /* ... */ }
};
class Outer {
private:
  Inner contained;
public:
};
```

18. When are contained objects created?

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

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

```
Outer( int n )
  : contained(Inner(n)) {
    /* ... */
};
```

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

19. 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'; };
};
```

20. Copy constructor in action

Code:

```
// object/copyscalar.cpp
has_int an_int(5);
has_int other_int(an_int);
an_int.printme();
other_int.printme();
has_int yet_other = other_int;
yet_other.printme();
```

Output:

```
set: 5
copy: 5
I have: 5
I have: 5
copy: 5
I have: 5
```

21. Copying is recursive

Class with a vector:

```
// object/copyvector.cpp
class has_vector {
private:
  vector<int> myvector;
public:
  has_vector(int v) { myvector.push_back(v); };
  void set(int v) { myvector.at(0) = v; };
  void printme() { cout
      << "I have: " << myvector.at(0) << '\n'; };
};
```

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

Code:

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

Output:

```
I have: 3
I have: 5
```

22. Destructor

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

```
~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.

23. Destructor example

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

```
// object/destructor.cpp
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << '\n';
  };
  ~SomeObject() {
    cout << "calling the destructor"
         << '\n';
  };
};
```

24. Destructor example

Destructor called implicitly:

Code:

```
// object/destructor.cpp
cout << "Before the nested scope"
     << '\n';
{
  SomeObject obj;
  cout << "Inside the nested scope"
       << '\n';
}
cout << "After the nested scope"
     << '\n';
```

Output:

```
Before the nested scope
calling the constructor
Inside the nested scope
calling the destructor
After the nested scope
```

Headers

25. C headers plusplus

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

Classes in C++ need some extra syntax.

26. 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 ....
};
```

27. 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
```

28. 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;
```