Advanced Objects

Victor Eijkhout, Susan Lindsey

Fall 2025

`last formatted: October 3, 2025`

Headers

1. Independent definition of function

Functions can be defined outside the class declaration:

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

Implementation:

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

2. C headers plusplus

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

Header files for classes in C++ need some extra syntax.

3. Independent definition of function

Functions can be defined outside the class declaration:

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

Implementation:

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

'This'

4. '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 i) {
    this->myint = i;   // option 1
    (*this).myint = i; // option 2
  };
};
```

5. ‘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

6. Operator overloading

Syntax:

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

For instance:

Code:

```
// geom/pointscale.cpp
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 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`'.

7. Index overloading

If you have objects that contain an array, you may want to overload the index operator.

Code:

```
// array/getindex4.cpp
class vector10 {
private:
  int array[10];
public:
    /* ... */
  int operator[](int i) {
    return array[i];
  };
    /* ... */
  vector10 v;
  v[2] = 34; // COMPILE ERROR!!!
  cout << v[2] << '\n';
```

Output:

```
getindex4.cpp:38:8: error: expression is not assignable
   38 |    v[2] = 34; // COMPILE ERROR!!!
      |    ~~~~ ^
1 error generated.
make[1]: *** [../Make.inc:118: getindex4.o] Error 1
```

Getting works, setting doesn't.

8. Index overloading'

Two overloads of the index operator, note the `const`:

Code:

```
// array/getindex5.cpp
class vector10 {
private:
  int array[10];
public:
    /* ... */
  vector10 v;
  v[2] = 34;
  cout << v[2] << '\n';
```

Output:

```
34
```

More constructors

9. 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:
};
```

10. 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.

11. 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'; };
};
```

12. 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
```

13. 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
```

14. 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.

15. Destructor example

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

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

16. 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
```

References and such

17. 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.

18. 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
}
```

19. 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.

20. Const functions

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

Static variables

21. 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
```

22. 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;
```