Objects and classes

Victor Eijkhout, Susan Lindsey

Fall 2023

`last formatted: September 7, 2023`

1. 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.

2. 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
}
```

3. Access gone wrong

We make a class for points on the unit circle

```
// /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.

4. Const functions

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

5. 'this' pointer to the current object

Inside an object, a pointer to the object is available as `this`:

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

6. ‘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

7. Operator overloading

Syntax:

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

For instance:

Code:

```
// /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

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

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

8. 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:
};
```

9. 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.

10. 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.

```
// /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'; };
};
```

11. Copy constructor in action

Code:

```
// /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
```

12. Copying is recursive

Class with a vector:

```
// /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:

```
// /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
```

13. 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.

14. Destructor example

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

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

15. Destructor example

Destructor called implicitly:

Code:

```
// /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

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:

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

Implementation file:

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

18. Static class members

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

Code:

```
// /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
```

19. Static class members, C++11 syntax

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