Advanced Objects

Victor Eijkhout, Susan Lindsey

Fall 2025
last formatted: October 2, 2025

Headers

1. **Independent definition of function**

Functions can be defined outside the class declaration:

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

Implementation:

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

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:

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

Implementation:

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

'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:

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

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)

```
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)

Operator overloading

6. Operator overloading

Syntax:

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

For instance:

```
 Code:
1 // geom/pointscale.cpp
2 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`'.

7. Index overloading

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

```
Code:
1 // array/getindex4.cpp
2 class vector10 {
3 private:
4   int array[10];
5 public:
6   /* ... */
7   int operator[](int i) {
8     return array[i];
9   };
10   /* ... */
11 vector10 v;
12 v[2] = 34; // COMPILE ERROR!!!
13 cout << v[2] << '\n';
```

```
Output:
1 getindex4.cpp:38:8:
     ↪error:
     ↪expression is
     ↪not assignable
2  38 |   v[2] = 34;
     ↪// COMPILE
     ↪ERROR!!!
3    |   ~~~~ ^
4 1 error generated.
5 make[1]: ***
     ↪[../Make.inc:118:
     ↪getindex4.o]
     ↪Error 1
```

TACC
Getting works, setting doesn't.

8. Index overloading'

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

```
Code:
1 // array/getindex5.cpp
2 class vector10 {
3 private:
4   int array[10];
5 public:
6     /* ... */
7   int operator[](int i) const {
8     return array[i];
9   };
10  int& operator[](int i) {
11    return array[i];
12  };
13    /* ... */
14  vector10 v;
15  v[2] = 34;
16  cout << v[2] << '\n';
```

```
Output:
1 34
```

More constructors

9. 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 };
```

10. 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.

11. **Copy constructor**

- Default defined copy and 'copy assignment' constructors:

```
1 some_object x(data);
2 some_object y = x;
3 some_object z(x);
```

- They copy an object:
  - simple data, including pointers
  - included objects recursively.

- You can redefine them as needed.

```cpp
1 // object/copyscalar.cpp
2 class has_int {
3 private:
4   int mine{1};
5 public:
6   has_int(int v) {
7     cout << "set: " << v
8          << '\n';
9     mine = v; };
10  has_int( has_int &h ) {
11    auto v = h.mine;
12    cout << "copy: " << v
13         << '\n';
14    mine = v; };
15  void printme() {
16    cout << "I have: " << mine
17         << '\n'; };
18 };
```

12. **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
```

13. **Copying is recursive**

Class with a vector:

```cpp
// 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:

```
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.

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:
```
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

17. 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.

TACC

18. Reference to internals

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

```cpp
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 }
```

19. Access gone wrong

We make a class for points on the unit circle

```cpp
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.

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:

```cpp
// 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

```
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;
```