

# Objects and classes

Victor Eijkhout, Susan Lindsey

Fall 2019

# Classes

# Definition of object

An object is an entity that you can request to do certain things. These actions are the *methods* and to make these possible the object probably stores data, the *members*.

When designing an object, first ask yourself: 'what functionality should this support'.

# Object functionality

Small illustration: vector objects.

**Code:**

```
Vector v(1.,2.); // make vector (1,2)
cout << "vector has length "
      << v.length() << endl;
v.scaleby(2.);
cout << "vector has length "
      << v.length() << endl
      << "and angle " << v.angle()
      << endl;
```

**Output**

**[object] functionality:**

```
vector has length 2.23607
vector has length 4.47214
and angle 1.10715
```

Note the 'dot' notation; in a `struct` we use it for the data members; in an object we (also) use it for methods.

# Exercise 1

Thought exercise:

What data does the object need to store to do this?

Is there more than one possibility?

# The object workflow

Similar to `struct`:

- You have to declare what an object looks like by giving a

```
class myobject { /* ... */ };
```

definition, typically before the *main*;

- You create specific objects with a declaration

```
myobject  
  object1( /* .. */ ),  
  object2( /* .. */ );
```

- You let the objects do things:

```
object1.do_this();  
object2.do_that( /* ... */ );
```

# Constructor

Use a constructor: function with same name as the class.  
Typically used to initialize data members.

```
class Vector {  
private: // members  
    double x,y;  
public: // methods  
    Vector( double x,double y )  
        : x(x),y(y) {};
```

The syntax `x(x)` copies the argument to the data member.

# Member default values

Class members can have default values, just like ordinary variables:

```
class Point {  
private:  
    float x=3., y=.14;  
private:  
    // et cetera  
}
```

Each object will have its members initialized to these values.



# Member initialization in the constructor

```
class Vector {  
private: // members  
    double r,theta;  
public: // methods  
    Vector( double x,double y ) {  
        r = sqrt(x*x+y*y);  
        theta = atan(y/x);  
    }  
}
```

# Methods

# Class methods

We used methods *length* and *scaleby*. These are defined inside the class:

```
double length() {  
    return sqrt(x*x + y*y); };  
double angle() {  
    return atan(y/x);  
};  
};
```

- They look like ordinary functions,
- except that they can use the data members of the class, for instance *x*;
- Methods can only be used on an object:

```
Vector vec(5,12);  
double s = vec.length();
```

## Exercise 2

Add methods *print* and *angle* to the *Vector* class.

How many parameters do they need?

# Methods that alter the object

Code:

```
class Vector {  
    /* ... */  
    void scaleby( double a ) {  
        vx *= a; vy *= a; };  
    /* ... */  
};  
  
/* ... */  
Vector p1(1.,2.);  
cout << "p1 has length "  
      << p1.length() << endl;  
p1.scaleby(2.);  
cout << "p1 has length "  
      << p1.length() << endl;
```

Output

[geom] pointscaleby:

```
p1 has length 2.23607  
p1 has length 4.47214
```

# Methods that create a new object

Code:

```
class Vector {  
    /* ... */  
    Vector scale( double a ) {  
        return Vector( vx*a, vy*a ); };  
    /* ... */  
    cout << "p1 has length "  
        << p1.length() << endl;  
    Vector p2 = p1.scale(2.);  
    cout << "p2 has length "  
        << p2.length() << endl;  
}
```

Output

[geom] pointscale:

```
p1 has length 2.23607  
p2 has length 4.47214
```

# Default constructor

```
Vector v1(1.,2.), v2;  
cout << "v1 has length " << v1.length() << endl;  
v2 = v1.scale(2.);  
cout << "v2 has length " << v2.length() << endl;
```

gives (g++; different for intel):

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

# Default constructor

The problem is with `v2`:

```
Vector v1(1.,2.), v2;
```

- `v1` is created with the constructor;
- `v2` uses the default constructor;
- as soon as you define a constructor, the default constructor goes away;
- you need to redefine the default constructor:

```
Vector() {};  
Vector( double x,double y )  
    : x(x),y(y) {};
```



## Exercise 3

Extend the `Point` class of the previous exercise with a method: `distance` that computes the distance between this point and another: if `p,q` are `Point` objects,

`p.distance(q)`

computes the distance between them.

Hint: remember the 'dot' notation for members.

## Optional exercise 4

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

You can write this function directly, or you could write functions `Add` and `Scale` and combine these.

(Later you will learn about operator overloading.)

# Constructors

# Constructor

Use a constructor: function with same name as the class.  
Typically used to initialize data members.

```
class Vector {  
private: // members  
    double x,y;  
public: // methods  
    Vector( double x,double y )  
        : x(x),y(y) {};
```

The syntax `x(x)` copies the argument to the data member.

# Public versus private

- Interface: `public` functions that determine the functionality of the object; effect on data members is secondary.
- Implementation: data members, keep `private`: they only support the functionality.

This separation is a Good Thing:

- Protect yourself against inadvertant changes of object data.
- Possible to change implementation without rewriting calling code.

## Exercise 5

Make a class `LinearFunction` with a constructor:

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

and a 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;
```

## Exercise 6

Make a class `LinearFunction` with two constructors:

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

where the first stands for a line through the origin.  
Implement again the `evaluate` function so that

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

# Classes for abstract objects

Objects can model fairly abstract things:

**Code:**

```
class stream {
private:
    int last_result{0};
public:
    int next() {
        return last_result++; };
};

int main() {
    stream ints;
    cout << "Next: "
         << ints.next() << endl;
    cout << "Next: "
         << ints.next() << endl;
    cout << "Next: "
         << ints.next() << endl;
```

**Output**

**[object] stream:**

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



# Project Exercise 7

Write a class `primegenerator` that contains

- members `how_many_primes_found` and `last_number_tested`,
- a method `nextprime`;
- Also write a function `isprime` that does not need to be in the class.

Your main program should look as follows:

```
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found() < nprimes) {
    int number = sequence.nextprime();
    cout << "Number " << number << " is prime" << endl;
}
```

## Project Exercise 8

The Goldbach conjecture says that every even number, from 4 on, is the sum of two primes  $p + q$ . Write a program to test this for the even numbers up to a bound that you read in.

This is a great exercise for a top-down approach! Make an outer loop over the even numbers  $e$ . In each iteration, make a `primegenerator` object to generate  $p$  values. For each  $p$  test whether  $e - p$  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.

# Turn it in!

- If you have compiled your program, do:  
`sdstestgold yourprogram.cc`  
where 'yourprogram.cc' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:  
`sdstestgold -s yourprogram.cc`  
where the `-s` flag stands for 'submit'.
- If you don't manage to get your code working correctly, you can submit as incomplete with  
`sdstestgold -i yourprogram.cc`

## Other object stuff

# Class prototypes

Header file:

```
class something {  
private:  
    int i;  
public:  
    double dosomething( std::vector<double> v );  
};
```

Implementation file:

```
double something::dosomething( std::vector<double> v ) {  
    // do something with v  
};
```

## **Advanced stuff about constructors**

# Copy constructor

- Several default copy constructors are defined
- They copy an object:
  - simple data, including pointers
  - included objects recursively.
- You can redefine them as needed, for instance for deep copy.

```
class has_int {  
private:  
    int mine{1};  
public:  
    has_int(int v) {  
        cout << "set: " << v <<  
        endl;  
        mine = v; };  
    has_int( has_int &h ) {  
        auto v = h.mine;  
        cout << "copy: " << v <<  
        endl;  
        mine = v; };  
    void printme() { cout  
        << "I have: " << mine <<  
        endl; };  
};
```

# Copy constructor in action

## Code:

```
has_int an_int(5);  
has_int other_int(an_int);  
an_int.printme();  
other_int.printme();
```

## Output

[object] copyscalar:

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



# Copying is recursive

Class with a vector:

```
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) << endl; };  
};
```

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

**Code:**

```
has_vector a_vector(5);  
has_vector other_vector(a_vector);  
a_vector.set(3);  
a_vector.printme();  
other_vector.printme();
```

**Output**

**[object] copyvector:**

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

# 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.

# Destructor example

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

```
class SomeObject {  
public:  
    SomeObject() {  
        cout << "calling the constructor"  
              << endl;  
    };  
    ~SomeObject() {  
        cout << "calling the destructor"  
              << endl;  
    };  
};
```

# Destructor example

Destructor called implicitly:

**Code:**

```
cout << "Before the nested scope"
      << endl;
{
    SomeObject obj;
    cout << "Inside the nested scope"
          << endl;
}
cout << "After the nested scope"
      << endl;
```

**Output**

**[object] destructor:**

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