# 2. Overloading, operators and friends

Max Cattafi
m.cattafi@imperial.ac.uk

EE2-12 – Software Engineering 2
Object Oriented Software Engineering

Imperial College
London

# In this lecture

- Some remarks on overloading.
- Defining operators for our classes.

# `Point` again

File `point.hpp` containing the class declaration.

```
#ifndef POINT_HPP
#define POINT_HPP
class Point{
    public:
        Point();
        Point(double x_in, double y_in);
        double get_x() const;
        double get_y() const;
        Point origin_symmetric(const Point& p) const;
        void origin_symmetric();
    private:
        double x;
        double y;
};
#endif
```

# Point again

File point.cpp containing the implementation of member functions.

```
#include "point.hpp"
Point::Point(){
    x = 0;
    y = 0;
}
Point::Point(double x_in, double y_in){
    x = x_in;
    y = y_in;
}
double Point::get_x() const{
    return x;
}
double Point::get_y() const{
    return y;
}
// etc.
```

# Compiling before the `main`

- Compile the class source with `g++ -c point.cpp`
- An object file ('object' not in the sense of OOP, it's called like this also in C) `point.o` is generated.
- (Header files are already (materially) included into the relevant source files.)
- The `.o` file cannot be executed, but it can be linked with other `.o` files (assuming there is also a `main.o` obtained as above):
  `g++ point.o main.o -o pm`
- This is useful in order to:
  - Find syntax errors in a specific source file.
  - (Re)compile only the source files which have been modified (less time required to (re)compile large applications).

# Another `main`

```
// file mainpv.cpp containing an example main function
#include <iostream>
#include <vector>
#include "point.hpp"
using namespace std;
int main(){
    vector<Point> vp(3); // constructor for vector
    cout << "initial points: " << endl;
    for(int i = 0; i<vp.size(); ++i){
        cout << vp[i].get_x() << " " << vp[i].get_y() << endl;
    }
    cout << "overwrite initial points: " << endl;
    double x, y;
    for(int i = 0; i<vp.size(); ++i){
        cin >> x >> y;
        Point p1(x, y); // local Point object
        vp[i] = p1; // we can assign objects with =
    }
    cout << "add new points: " << endl;
    do{
        cin >> x >> y;
        Point p1(x, y); // local Point object
        vp.push_back(p1); // *copied* in vector
    } while(!(x == 0 && y == 0));
    cout << "one more point: " << endl;
    cin >> x >> y;
    vp.push_back(Point(x,y)); // temporary object
    cout << "all " << vp.size() << " points: " << endl;
    for(int i = 0; i<vp.size(); ++i){
        cout << vp[i].get_x() << " " << vp[i].get_y() << endl;
    }
    return 0;
}
```

# Outcome

```
initial points:
0 0
0 0
0 0
overwrite initial points:
1.2 4.3
-2 3.4
1 1
add new points:
0 3.3
0 0
one more point:
2 1
all 6 points:
1.2 4.3
-2 3.4
1 1
0 3.3
0 0
2 1
```

# Why overloading constructors?

- Imagine we remove the default constructor.

```
// in point.hpp
//     Point(double init_x, double init_y);


// in point.cpp
//     Point::Point(){
//         x = 0;
//         y = 0;
//     }
```

- The idea being that the constructor with parameters is more general.
- We can e.g. just call `Point p(0,0);` when needed.

# Other changes are needed...

- We need to change something else in our class (but this is a good idea in any case):

```
Point Point::origin_symmetric(const Point& p) const {
//      Point tmp;
//      tmp.x = - p.x;
//      tmp.y = - p.y;
    Point tmp(-p.x, -p.y);
    return tmp;
}
```

- Would the rest e.g. `vector<Point> vp(3);` still work?

# Would it?

```
g++ -c mainpv.cpp

 mainpv.cpp:
 In constructor std::vector<_Tp, _Alloc>::vector
 [etc.]
 error: no matching function for call to Point::Point()
 note: candidates are: Point::Point(double, double)
 note:                 Point::Point(const Point&)
```

# The compiler message explained

- `error: no matching function for call to Point::Point()`

→ The default (i.e. without arguments) constructor is missing.

- `candidates are: Point::Point(double, double)`

→ There is a constructor with two arguments defined (by us) for the class (but it's unsuitable for `vector<Point> vp(3)`).

- `note: Point::Point(const Point&)`

→ There is also another constructor, not defined by us, which takes as argument another `Point` (as const reference).
This is the *copy constructor*.

- We can still make everything work using another constructor for `vector`:

```
vector<Point> vp(3, Point(0,0));
```

# Sawing the branch under your ladder

- Sometimes (often?) it is necessary to introduce changes in one's classes.
- (It happens also with functions.)
- Subtle mistakes easily introduced.
- Changes should always *extend* the class functionality (more on this in future lectures).
- Keeping back-compatibility with the rest of the code base.

# Overloading and implicit conversion

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  string my_type_check(int something){
5       return "int";
6  }
7  // string my_type_check(double something){
8  //    return "double";
9  // }
10 int main(){
11     int n1 = 1;
12     double n2 = 1.5;
13     cout << "n1 = " << n1 << " is " << my_type_check(n1) << endl;
14     cout << "n2 = " << n2 << " is " << my_type_check(n2) << endl;
15     return 0;
16 }
```

- Implicit conversion.
- No compiler errors and (usually) no warnings either.

```
n1 = 1 is int
n2 = 1.5 is int
```

# If you really want the type...

```
1  #include <iostream>
2  #include <typeinfo>
3  #include "point.hpp"
4
5  using namespace std;
6
7  int main(){
8      int n1;
9      double n2;
10     Point p;
11     cout << typeid(n1).name() << endl;
12     cout << typeid(n2).name() << endl;
13     cout << typeid(p).name() << endl;
14     return 0;
15 }
```

- Prints something resembling `int, double, Point`.
- In my case, respectively `i, d, 5Point`.
- Might need compiler enabling RunTime Type Information (RTTI).
- We'll see in the future why RunTime.

# Not overloadable

- Can't overload on the return type.
- Design choice.
- "Keep resolution for an individual operator or function call context-independent".

```
1   float sqrt(float);
2   double sqrt(double);
3
4   void f(double da, float fla)
5   {
6       float fl = sqrt(da);          // call sqrt(double)
7       double d = sqrt(da);          // call sqrt(double)
8       fl = sqrt(fla);               // call sqrt(float)
9       d = sqrt(fla);                // call sqrt(float)
10  }
```

"[otherwise] it would no longer be possible to look at a call of sqrt() in isolation [e.g. when the return value is not used] and determine which function was called." [Bjarne Stroustrup, "The C++ Programming Language"]

# Default arguments

```
// in point.hpp
Point(double in_x = 0, double in_y = 0);
```

```
// in point.cpp
Point::Point(double in_x /* = 0 */, double in_y /* = 0 */){
// add a comment (not to forget
// there are defaults in the declaration...)
    x = in_x;
    y = in_y;
}
```

(This works for any function, also non-member ones.)

# Using default arguments

```
Point p1, p2(1), p3(1,2);
// p1 is (0,0)
// p2 is (1,0)
// p3 is (1,2)
```

- Only trailing parameters can have default arguments.
  E.g.: `void f(int a=0, int b)` is not allowed.
- Once a default is used, all the following arguments get default values too.
  E.g.: We can't set the y coordinate to a specific value, unless we set the x one too.

# Overloading or default arguments?

- If there are no meaningful defaults which can be used, use overloading.
- If the default values can be used in the function in the exact same way as any other values (i.e. they are not a special case) use default arguments.
- If the behaviour of the function differs significantly in the default and non-default cases, use overloading.

# Operator overloading

```
int n = 10;
cout << (n == n) << endl;
// compares two int
// prints a bool (1) (on the std out)
cout << (n + n) << endl;
// sums two int
// prints an int (20)
string s = "10";
cout << (s == s) << endl;
// compares two strings
// prints a bool (1)
ofstream outf("out.txt");
outf << (s + s) << endl;
// appends two strings
// prints a string (1010) (on a file)
```

# Operators

- The `==` operator on `string`s is declared as:

```
bool operator==(
        const string& st1, const string& st2
)
```

- Writing:

```
cout « operator==(s, s);
```

  is equivalent to:

```
cout « (s == s);
```

- The latter is (arguably) more readable.
- We can overload operators like we would overload functions.

# Operators as (member?) functions

- Notice that

```
bool operator==(
        const string& st1, const string& st2
)
```

  *is not* a member function of `string`.
- (Otherwise it would be written as: `bool string::operator==()` .)
- Some operators, if overloaded, *must* be member functions (notably: assignment = and subscript `[ ]`).
- Some operators are conventionally defined as member functions (e.g. +=).
- Some operators, including == for `string`, are conventionally non-member (global) functions.

# Requirements specification

Write a program which:

- reads from the user a sequence of `Point` objects terminated by the origin (0, 0)
- counts how many times a duplicate element is entered and prints the result.

# Implementation

```
#include <iostream>
#include <vector>
#include "point.hpp"
using namespace std;
int main(){
    int duplicates = 0;
    double x, y;
    vector<Point> vp;
    do{
        cin >> x >> y;
        Point p1(x, y);
        int i = 0;
        bool found = false;
        while(!found && i<vp.size()){
            if(p1 == vp[i]){
                found = true;
                cout << "duplicate found!" << endl;
                duplicates++;
            }
            ++i;
        }
        vp.push_back(p1);
    } while(!(x == 0 && y == 0));
    cout << "you have entered " << duplicates << " duplicates" << endl;
    return 0;
}
```

```
error: no match for operator== in p1 == vp.std::vector<_Tp, _Alloc>::operator[] [with _Tp = Point, ↩
      _Alloc = std::allocator<Point>](((unsigned int)i))
```

# Why?

- We have assignment $=$ which seems to work fine (copying fields).
- Why don't we have also comparison $==$ (comparing fields)?
- *I personally consider it unfortunate that copy operations are defined by default and I prohibit copying of objects of many of my classes. However, C++ inherited its default assignment and copy constructors from C, and they are frequently used.*
  *– Bjarne Stroustrup, "The Design and Evolution of C++"*
- (We'll see also why BS considers it unfortunate that assignment and copy constructors are defined by default.)

# Point with ==

- We would still like to test equality between points with ==.
- `if(p1 == vp[i]){`
  seems more readable than

```
if(
    (p1.get_x() == vp[i].get_x())
    &&
    (p1.get_y() == vp[i].get_y())
){
```

- We define two points as equal if they have the same coordinates.
- 
```
bool operator==(
    const Point& p1, const Point& p2
)
```

- We need to access the coordinates.

# Accessing `private` fields

- We can use getters.
- However the purpose of overloading operators is also to be able to do without getters.
- Remember we introduced getters in order to print the state of `Point` objects, but we'd like to eventually just print it as `cout « p1 « endl`
- Is there another way?
- How does `string` do it?

# Friend functions

```
// in point.hpp
class Point{
    public:
    ...
    friend bool operator==(
        const Point& p1, const Point& p2
    );
    ...
};


// somewhere else, e.g. point.cpp
bool operator==(const Point& p1, const Point& p2){
    return (p1.x == p2.x) && (p1.y == p2.y);
}
```

# Friends

- Functions declared as `friend` in a class declaration can access its private data (as if they were member functions).
- The `friend` declaration can be applied to global functions, member functions, and entire classes (it then holds for all the member functions in that class).
- Friendship among classes is not reciprocal and not transitive either.

# Is friendship a good idea?

- Arguably doesn't break encapsulation (you still know what is accessing what).
- Still arguably not very elegant.
- "C++ is a hybrid object-oriented language, not a pure one, and `friend` was added to get around practical problems that crop up."
- "It's fine to point out that this makes the language less 'pure', because C++ is designed to be pragmatic, not to aspire to an abstract ideal."
  [Bruce Eckel, "Thinking in C++"]

# Friends overcome diversity

```
string operator+ (const string& lhs,const string& rhs);
string operator+ (const char* lhs, const string& rhs);
string operator+ (char lhs, const string& rhs);
string operator+ (const string& lhs, const char* rhs);
string operator+ (const string& lhs, char rhs);
```

- "Because of *the diversity of left-hand parameter types*, this function is implemented as an overload of the global `operator+` function."
  [cplusplus.com, string std::operator+ reference]

# `Point` with `<`

- We would like to define an order relation between `Point`s.
- Let `Point p1` be 'less than' (`<`) `Point p2` if and only if `p1` is closer to the origin (0,0) than `p2`.

# Compare distances

```
// in point.hpp
class Point{
    public:
    ...
    double distance(const Point& p1);
    ...
    friend bool operator<(
        const Point& p1, const Point&p2
    );
    ...
};

// in point.cpp

double Point::distance(const Point& p1){
    ...
}

// somewhere else or in point.cpp
bool operator<(const Point& p1, const Point& p2){
    Point o(0,0);
    return (p1.distance(o) < p2.distance(o));
}
```

# Something went wrong

```
In function
"bool operator<(const Point&, const Point&)":
error: passing "const Point" as "this" argument of
"double Point::distance(const Point&)"
discards qualifiers
error: passing "const Point" as "this" argument of
"double Point::distance(const Point&)"
discards qualifiers
```

# Information asymmetry

```
bool operator<(const Point& p1, const Point& p2){
    Point o(0,0);
    return (p1.distance(o) < p2.distance(o));
}
```

- `p1` and `p2` passed as `const` references.
- We know that member function `distance` doesn't change the object it is called on.
- But the compiler doesn't!

# Let the compiler know (keep const correctness)

```
// in point.h
class Point{
    public:
    ...
    double distance(const Point& p2) const;
    ...
};

// in point.cpp
double Point::distance(const Point& p2) const {
    ...
}
```

# Wrap-up

- Overloading operators for more abstraction and encapsulation.
- Choose your friends carefully.
- Be `cons[t]`istent.