# EE2-12 Software Engineering 2: Object-Oriented Programming

## Week 5 - Classes Relationships: Association, Aggregation/Composition and Generalisation (Inheritance)

Sahbi Ben Ismail

Imperial College London, Department of Electrical and Electronic Engineering

Imperial College London

Autumn Term 2018-19

# Module Syllabus

- Week 1 Classes and Objects I: Introduction
- Week 2 Classes and Objects II: Constructors, and Operator Overloading
- Week 3 More on Classes, Objects, and Operator Overloading
- Week 4 Objects and Dynamic Memory
- Week 5 **Classes Relationships: Association, Aggregation/Composition and Generalisation (Inheritance)**
- Week 6 Polymorphism and Virtual Functions
- Week 7 Generic Programming: Templates, and the Standard Template Library (STL)
- Week 8 Exceptions Handling
- Week 9 C++ to Java
- Week 10 Revision

# Week 5 Intended Learning Outcomes (ILOs)

By the end of this week you should be better able to:

## Lecture

1. Understand the "***Big Three***" of a class with dynamically allocated members: the **destructor**, the **copy constructor**, and the **assignment operator**

1. Understand "***has-a***" relatashionship between classes, represent it in UML and implement it in C++
2. Understand "***is-a***" relatashionship between classes (**inheritance**), represent it in UML and implement it in C++

## Lab

1. Implement the "***Big Three***" for class `polynomial`
2. Revisit **composition** between classes `point` and `triangle`
3. Apply **inheritance** between classes `point` and `labeled_point`

# Outline

1. Introduction
   - Week 4 Dynamic Memory - Summary
2. Classes relashionships
   - Composition
   - Inheritance
3. Conclusion

# Outline

# Bjarne Stroustrup - GoingNative 2013

Youtube video https://www.youtube.com/watch?v=D5MEsboj9Fc
(start at 12'00", until 15'20" https://youtu.be/D5MEsboj9Fc?t=719)

## Bjarne Stroustrup - The Essence of C++

- C++ in four slides
- "*If you understand int and vector, you understand C++.*
    - *the rest is "details,*
    - *... but don't get lost in them."*
- value vs handle
- classes: construction/destruction

# Reminder about Labs

**Labs do not ONLY consist in applying the lectures topics, but also**

- test, observe, and understand
- make mistakes (at compilation and runtime), fix the bugs
- discover, self-learn
- ask questions
- get instant feedback from peers/teaching team

**next week lecture:**

- global feedback
- clarification
- pitfalls

**Week 4 - Lab**

- not well detailed
- too many topics for a Lab
  ↪re-adjustment needed (2019-20)

# Week 4 - Objects and Dynamic Memory

- pointers vs objects
- pointers vs dynamic arrays
- dynamic memory allocation and de-allocation
- *The Big Three*: the **destructor**, the **copy constructor**, and the **assignment operator**
  1. default vs user-defined ones
  2. a class with vs without dynamically allocated members: class `point` vs class `polynomial`
- closer eye on the class `std::vector`

# Pointer variables

## Pointer definition

Memory address of a variable.

```
int n = 1; //int
int* ptr = &n; //pointer to int
//ptr points to n

//address of operator (&)
//dereference operator (*)
```

## Pitfall: multiple pointers declaration

`int *p1, *p2, v1, v2;`

- p1, p2 hold pointers to int variables
- v1, v2 are ordinary int variables

# Pointers assignments I

## Pitfall: confusion between assignments

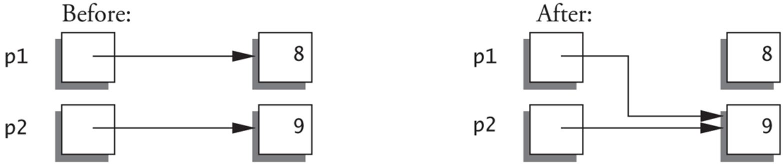
```
int *p1, *p2;
```

1. `p2 = p1;`
    - assigns one pointer to another
    - "make p2 point to where p1 points"
2. `*p1 = *p2;`
    - assigns "value pointed to" by p1, to "value pointed to" by p2

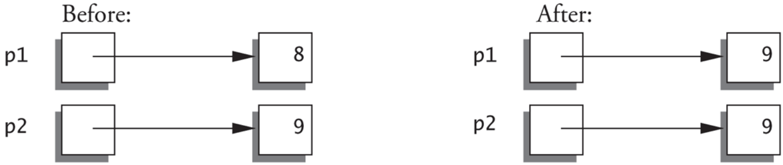
# Pointers assignments II

Uses of the assignment operator with pointer variables

```
p1 = p2;
```

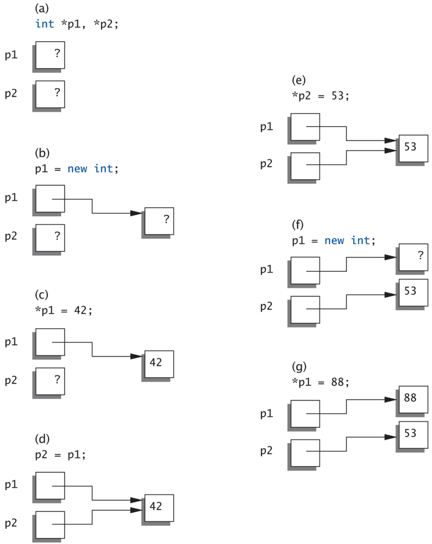



```
*p1 = *p2;
```



# Basic pointer manipulations

```
#include <iostream>
using namespace std;

int main() {
  int *p1, *p2;

  p1 = new int();
  *p1 = 42;
  p2 = p1;
  cout << "*p1 == " << *p1 << endl;
  cout << "*p2 == " << *p2 << endl;

  *p2 = 53;
  cout << "*p1 == " << *p1 << endl;
  cout << "*p2 == " << *p2 << endl;

  p1 = new int();
  *p1 = 88;
  cout << "*p1 == " << *p1 << endl;
  cout << "*p2 == " << *p2 << endl;

  return 0;
}
```



# Pointers vs Objects

- arrow operator `pointer->` vs dot operator `object.`
- `pointer->member` equivalent to `(*pointer).member`

```
int main() {
  point p(1, 2); //object of class point          /* Output:
                                                     1 2
  point* ptr = &p; //pointer to point                1 2
                                                     1 2
  cout << p.get_x() << " " << p.get_y() << endl;  */

  cout << ptr->get_x() << " " << ptr->get_y() << endl;
  cout << (*ptr).get_x() << " " << (*ptr).get_y() << endl;

  return 0;
}
```

# The predefined this pointer

- points to the current object (holds its memory address)
- this->member equivalent to (*this).member

```
double point::get_x() {
  return(this->x); //optional
                   //x <=> this->x <=> (*this).x
}

double point::distance_to(double x, double y) const {
  double dx = this->x - x;
  double dy = this->y - y;

  //"this" resolves ambiguity between the current object's members
  //and the formal parameters

  return(sqrt(dx*dx + dy*dy));
}
```

# Pointers vs Arrays vs Dynamic arrays

```
int main() {
  int* ptr;

  int a1[10];

  ptr = a1; //legal assignment, a1 and ptr are both pointer variab
  //a = ptr; //ILLEGAL! a1 is a CONSTANT pointer!
  //error: incompatible types in assignment of 'int*' to 'int [10]

  //dynamic array
  int* a2 = new int[5];
  ptr = a2; //legal assignment
  a2 = ptr; //legal assignment
}
```

# Pointers vs Arrays - Examples

## argv in the full prototype of the main function

```
int main(int argc, char* argv[]);
//an array of strings (pointers to char)

int main(int argc, char** argv);
//a pointer to string i.e an array of strings

//...
  std::cout << argv[2] << std::endl; //array notation
  std::cout << *(argv+2) << std::endl; //pointer notation
```

## Pointers vs Dynamic arrays

```
int main() {
  double* foo = new double[20];
  foo[2] = 2.5; // equivalent to *(foo+2) = 2.5;
  //(array notation is clearer than the pointer notation)
  //...
  delete[] foo;
```

# Dynamic memory allocation & de-allocation

## dynamic memory allocation: **new** keyword

```
int n = new int(5); //with a built-in primitive type (non class)
point p = new point(2, 3); //with a class type
double* d_array = new double[5]; //dynamic array
```

## Heap

- also called *"freestore"*
- reserved for dynamically-allocated variables (vs Stack)
- Memory IS **finite** (regardless of how much there is!)

## dynamic memory de-allocation: **delete** keyword

```
delete p;
delete[] d_array;
```

# Automatically vs dynamically allocated variables

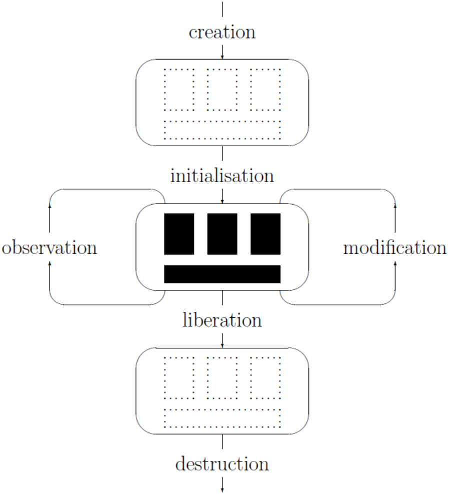
```
int main() {
  point p(1, 2);
  point* ptr = new point(3, 4); //dynamically allocated pointer to point

  //...

  //delete p; //compilation ERROR
              //type 'class point' argument given to 'delete', expected pointer

  delete ptr; //Once, and ONLY once


  //delete ptr; //runtime ERROR
                //double free or corruption (fasttop): ... Memory map ...
  return 0;
}
```

## Lab pitfalls

- automatically allocated: use of `delete` $\hookrightarrow$ compilation error
- dynamically allocated:
  - no `delete`: memory leak
  - more than one `delete` $\hookrightarrow$ runtime error

# Objects life cylce

```
class A {
  public:
    A(params);   //constructor
    ~A();        //destructor
    // ...

  private:
    // ...
};
```



# Lab Problematic: copying, assigning, destroying

## Test Battery

1. class `point`
   1. default Big Three
   2. user-defined Big Three
2. class `polynomial`
   1. default Big Three [with member function `at(int)`]
   2. user-defined Big Three [with member function `at(int)` and the subscript operator]

helpful feature: insertion operator overloading for easy printing (`operator<<`)

# Lab Problematic: class point

```
class point {
  //...
  private:
    double x;
    double y;
};

int main() {
  point p(1, 2);

  point p1(p); // What happens here? (copy constructor)
  point p2 = p; // And here? (assignment operator)

  //... some modifications on p automatic members x and y
  // What happens to p1 and p2?

  //nearly finished! Ultimate two tests
  p2 = p2; // Does it work? (assignment to self)
  point p3 = p2 = p1; // And here? (chain assignment)

  //end of scope. What happens here? (destructor)
}
```

# Lab Problematic: class polynomial

```cpp
class polynomial {
  //...
  private:
    double* coefficients; //dynamically allocated
    int degree;
};

int main() {
  polynomial p(2);
  cout << p.at(0) << endl;
  cout << p[0] << endl; // Does it work? (subscript operator, case 1)

  p.at(0) = 2; // Does it work? (at() member function)
  p.at(1) = 3; p.at(2) = 1;

  p[0] = 5; // Does it work? (subscript operator, case 2)
  p[1] = -4; p[2] = 3;

  polynomial p1(p); // What happens here? (copy constructor)
  polynomial p2 = p; // And here? (assignment operator)

  //... some modifications on p dynamic member coefficients
  // What happens to p1 and p2?

  //nearly finished! Ultimate two tests
  p2 = p2; // Does it work? (assignment to self)
  point p3 = p2 = p1; // And here? (chain assignment)

  //end of scope. What happens here? (destructor)
}
```

# Class point - default big three I

```
#ifndef POINT_HPP
#define POINT_HPP

#include <iostream> //for ostream

//no using namespace std; use std::ostream instead

class point {
  public:
    point();
    point(double x_in, double y_in); //parameterised constructor

    double get_x();
    double get_y();
    double get_distance_orig() const;

    void set_x(double x_in);
    void set_y(double y_in);

    double distance_to(const point& other) const;
    void translate(const point& other);
    void to_symmetric();

    friend std::ostream& operator<<(std::ostream& os, const point& p);

  private:
    double x;
    double y;
    double distance_orig; //distance to the origin
    void update_distance();
};

#endif
```

# Class point - default big three II

```cpp
int main() {
  point p(1, 2);
  cout << "p: " << p << endl;

  point p1(p); // What happens here? (copy constructor)
  cout << "p1: " << p1 << endl;

  point p2 = p; // And here? (assignment operator)
  cout << "p2: " << p2 << endl;

  //... some modifications on p automatic members x and y
  // What happens to p1 and p2?
  p.set_x(5);
  cout << "\np.set_x(5)" << endl;

  cout << "p:" << p << endl;
  cout << "p1: " << p1 << endl;
  cout << "p2: " << p2 << endl;

  //nearly finished! Ultimate two tests
  p2 = p2; // Does it work? (assignment to self)
  point p3 = p2 = p1; // And here? (chain assignment)

  cout << "\np3: " << p3 << endl;

  //end of scope. What happens here? (destructor)

  return 0;
}
```

```
/* Output:
 ./prog
 p: [1, 2]
 p1: [1, 2]
 p2: [1, 2]

 p.set_x(5)
 p:[5, 2]
 p1: [1, 2]
 p2: [1, 2]

 p3: [1, 2]
*/
```

# Class point - default big three III

## Conclusion

Without dynamically allocated member variables:
$\hookrightarrow$ member data can be “trivially” copied and assigned
$\hookrightarrow$ The default Big Three works

# Class point - user-defined destructor I

```
//point.hpp
~point();

//point.cpp
point::~point() {
  std::cout << "point " << *this << " is leaving" << std::endl;
}
```

# Class point - user-defined destructor II

```
int main() {
  point p(1, 2);
  cout << "p: " << p << endl;

  point p1(p); // What happens here? (copy constructor)
  cout << "p1: " << p1 << endl;

  point p2 = p; // And here? (assignment operator)
  cout << "p2: " << p2 << endl;

  //... some modifications on p automatic members x and y
  // What happens to p1 and p2?
  p.set_x(5);
  cout << "\np.set_x(5)" << endl;

  cout << "p:" << p << endl;
  cout << "p1: " << p1 << endl;
  cout << "p2: " << p2 << endl;

  //nearly finished! Ultimate two tests
  p2 = p2; // Does it work? (assignment to self)
  point p3 = p2 = p1; // And here? (chain assignment)

  cout << "\np3: " << p3 << endl;

  //end of scope. What happens here? (destructor)

  return 0;
}
```

```
/* Output:
  ./prog
  p: [1, 2]
  p1: [1, 2]
  p2: [1, 2]

  p.set_x(5)
  p:[5, 2]
  p1: [1, 2]
  p2: [1, 2]

  p3: [1, 2]
  point [1, 2] is leaving
  point [1, 2] is leaving
  point [1, 2] is leaving
  point [5, 2] is leaving
*/
```

# Class polynomial - default big three I polynomial.hpp

```
#ifndef POLYNOMIAL_HPP
#define POLYNOMIAL_HPP

#include <iostream> //for std::ostream

class polynomial {
  public:
    polynomial();
    polynomial(int d);

    double at(int i) const; //works, but does not allow assignments like p.at(0) = 5;
    //double& at(int i);
    double get_degree() const;

    friend std::ostream& operator<<(std::ostream& os, const polynomial& p);

  private:
    double* coefficients;
    int degree; //coefficients has degree+1 elements, indexed from 0 to degree.
};

#endif
```

# Class polynomial - default big three II polynomial.cpp

```
#include "polynomial.hpp"

polynomial::polynomial() {
  degree = 0;
  coefficients = new double[degree+1];
}


polynomial::polynomial(int d) {
  degree = d;
  coefficients = new double[degree+1];

  for(int i=0; i<=degree; i++) {//not i<degree (coefficients size = degree+1)
    coefficients[i] = 0.0;
  }
}


//works, but does not allow assignments like p.at(0) = 5;
double polynomial::at(int i) const {
  return coefficients[i];
}
```

# Class polynomial - default big three III main.cpp

```cpp
#include <iostream>
//for std::cout and std::endl

#include "polynomial.hpp"

using namespace std;

int main() {
  ///////////////////////////
  //Test 1) at(int) member function
  ///////////////////////////
  polynomial p(2);
  cout << p << endl;
  p.at(0) = 2;
  p.at(1) = 3;
  p.at(2) = 1;
  cout << p << endl;

  ///////////////////////////
  //Test 2) copy constructor
  ///////////////////////////
  polynomial p1(p);
  cout << p1 << endl;
```

```cpp

  ///////////////////////////
  //Test 3) subscript operator[]
  ///////////////////////////
  p[0] = 5;
  p[1] = -4;
  p[2] = 3;
  cout << p << endl;

  ///////////////////////////
  //Test 4)
assignment operator
  ///////////////////////////
  p = p;

  polynomial p2 = p;
  cout << p2 << endl;

  polynomial p3 = p2 = p1;

  cout << p1 << endl;
  cout << p2 << endl;
  cout << p3 << endl;
}
```

## Errors: at(int) and operator[]

- at(int)

```
main.cpp:12:10: error: lvalue required as left operand
of assignment p.at(0) = 2;
```

- operator[]

```
main.cpp:28:3: error: no match for 'operator[]'
(operand types are 'polynomial'and 'int')
p[0] = 5;
```

# at(int)

- at(int)

  main.cpp:12:10: error: lvalue required as left operand

  of assignment p.at(0) = 2;

↪ return a reference

```
//polynomial.hpp
//double at(int i) const;
double& at(int i);

//polynomial.cpp
//works, but does not allow assignments like p.at(0) = 5;
/*double polynomial::at(int i) const {
  return coefficients[i];
}*/

double& polynomial::at(int i) {
  return coefficients[i];
}
```

# Errors: at(int) and operator[]

- operator[]

  main.cpp:28:3: error: no match for 'operator[]'

  (operand types are 'polynomial'and 'int')

  p[0] = 5;

↪ don'y use it (for the moment)

```
////////////////////////////
//Test 3) subscript operator[]
////////////////////////////
/*p[0] = 5;
p[1] = -4;
p[2] = 3;
cout << p << endl;*/
```

# Yep, First run!

```
/* Output:
  ./prog
  coefficients[0] = 0
  coefficients[1] = 0
  coefficients[2] = 0
  P(x) =

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2
  */
```

```
  /*
  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2
*/
```

# Observation: Shallow copy vs deep copy

## Shallow copy

- assignment copies only member variable contents over
- default assignment and copy constructors

## Deep copy

- pointers, dynamic memory involved
- must dereference pointer variables to "get to" data for copying
- write your own assignment overload and copy constructor in this case!

# Class polynomial - user defined operator[] v1

```
//polynomial.hpp
double& operator[](int i);

//polynomial.cpp
double& polynomial::operator[](int i) {
  return coefficients[i];
}
```

# Class polynomial - user defined copy constructor

```
//polynomial.hpp
polynomial(const polynomial& other);


//polynomial.cpp
polynomial::polynomial(const polynomial& other) {
  degree = other.degree; //or other.get_degree();, both work
  coefficients = new double[degree+1];

  for(int i=0; i<=degree; i++) {//not i<degree (coefficients size = degree+1)
    coefficients[i] = other[i];
  }
}
```

# Error in copy constructor (caused by operator[])

```
polynomial.cpp: In copy constructor 'polynomial::polynomial(const polynomial&)':
polynomial.cpp:24:28: error: passing 'const polynomial' as 'this'
argument discards qualifiers [-fpermissive]

   coefficients[i] = other[i];

In file included from polynomial.cpp:1:0:
polynomial.hpp:16:11: note:   in call to 'double& polynomial::operator[](int)'
   double& operator[](int i);
```

# Class polynomial - user defined operator[] v2 (const)

```
//polynomial.hpp
double& operator[](int i);
const double& operator[](int i) const;

//polynomial.cpp
double& polynomial::operator[](int i) {
  return coefficients[i];
}

const double& polynomial::operator[](int i) const {
  return coefficients[i];
}
```

# Yep, Second run!

```
/* Output:
  ./prog
  coefficients[0] = 0
  coefficients[1] = 0
  coefficients[2] = 0
  P(x) =

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 5
  coefficients[1] = -4
  coefficients[2] = 3
  P(x) =  + 5.x^0 -4.x^1 + 3.x^2
  */
```

```
  /*
  coefficients[0] = 5
  coefficients[1] = -4
  coefficients[2] = 3
  P(x) =  + 5.x^0 -4.x^1 + 3.x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2
*/
```

# Class polynomial - user defined destructor

```
//polynomial.hpp
~polynomial();
```

```
//polynomial.cpp
polynomial::~polynomial() {
  std::cout << "polynomial destructor" << std::endl;
  delete[] coefficients;
}
```

# Yep, Third run!

```
/* Output:
  ./prog
  coefficients[0] = 0
  coefficients[1] = 0
  coefficients[2] = 0
  P(x) =

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 5
  coefficients[1] = -4
  coefficients[2] = 3
  P(x) =  + 5.x^0 -4.x^1 + 3.x^2
  */
```

```
  /*
  coefficients[0] = 5
  coefficients[1] = -4
  coefficients[2] = 3
  P(x) =  + 5.x^0 -4.x^1 + 3.x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  coefficients[0] = 2
  coefficients[1] = 3
  coefficients[2] = 1
  P(x) =  + 2.x^0 + 3.x^1 + x^2

  polynomial destructor
  polynomial destructor
  polynomial destructor
  polynomial destructor

  Error (Memory) ...
*/
```

# Errors in delete (originated by assignment operator)

```
*** Error in './prog': double free or corruption (fasttop): 0x0000000001535050 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7fc0ddd927e5]
/lib/x86_64-linux-gnu/libc.so.6(+0x8037a)[0x7fc0ddd9b37a]
/lib/x86_64-linux-gnu/libc.so.6(cfree+0x4c)[0x7fc0ddd9f53c]
./prog[0x400cb1]
./prog[0x40124a]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7fc0ddd3b830]
./prog[0x400a19]
======= Memory map: ========
00400000-00402000 r-xp 00000000 08:11 6040830                          /home/sbenisma/
00601000-00602000 r--p 00001000 08:11 6040830                          /home/sbenisma/
00602000-00603000 rw-p 00002000 08:11 6040830                          /home/sbenisma/
01523000-01555000 rw-p 00000000 00:00 0                                [heap]
7fc0d8000000-7fc0d8021000 rw-p 00000000 00:00 0
7fc0d8021000-7fc0dc000000 ---p 00000000 00:00 0
7fc0dda12000-7fc0ddb1a000 r-xp 00000000 08:11 6029477                  /lib/x86_64-lin
7fc0ddb1a000-7fc0ddd19000 ---p 00108000 08:11 6029477                  /lib/x86_64-lin
7fc0ddd19000-7fc0ddd1a000 r--p 00107000 08:11 6029477                  /lib/x86_64-lin
7fc0ddd1a000-7fc0ddd1b000 rw-p 00108000 08:11 6029477                  /lib/x86_64-lin
7fc0ddd1b000-7fc0ddedb000 r-xp 00000000 08:11 6029486                  /lib/x86_64-lin
7fc0ddedb000-7fc0de0db000 ---p 001c0000 08:11 6029486                  /lib/x86_64-lin
7fc0de0db000-7fc0de0df000 r--p 001c0000 08:11 6029486                  /lib/x86_64-lin
7fc0de0df000-7fc0de0e1000 rw-p 001c4000 08:11 6029486                  /lib/x86_64-lin
7fc0de0e1000-7fc0de0e5000 rw-p 00000000 00:00 0
7fc0de0e5000-7fc0de0fb000 r-xp 00000000 08:11 6029377                  /lib/x86_64-lin
7fc0de0fb000-7fc0de2fa000 ---p 00016000 08:11 6029377                  /lib/x86_64-lin
7fc0de2fa000-7fc0de2fb000 rw-p 00015000 08:11 6029377                  /lib/x86_64-lin
7fc0de2fb000-7fc0de46d000 r-xp 00000000 08:11 7603223                  /usr/lib/x86_64
7fc0de46d000-7fc0de66d000 ---p 00172000 08:11 7603223                  /usr/lib/x86_64
7fc0de66d000-7fc0de677000 r--p 00172000 08:11 7603223                  /usr/lib/x86_64
7fc0de677000-7fc0de679000 rw-p 0017c000 08:11 7603223                  /usr/lib/x86_64
7fc0de679000-7fc0de67d000 rw-p 00000000 00:00 0
7fc0de67d000-7fc0de6a3000 r-xp 00000000 08:11 6029482                  /lib/x86_64-lin
```

# Class polynomial - user defined assignment operator v1

```
//polynomial.hpp
void operator=(const polynomial& other); //works, but does not allow chain assignment
                                         //like p1 = p2 = p3;


//polynomial.cpp
//works, but does not allow chain assignment like p1 = p2 = p3;
void polynomial::operator=(const polynomial& other) {
  if(&other != this) {
    delete[] coefficients;

    degree = other.degree;
    coefficients = new double[degree+1];

    for(int i=0; i<=degree; i++) {//not i<degree (coefficients size = degree+1)
      coefficients[i] = other[i];
    }
  }
}
```

# Class polynomial - user defined assignment operator v2

```
//polynomial.hpp
//void operator=(const polynomial& other); //works, but does not allow chain assignment
                                           //like p1 = p2 = p3;
polynomial& operator=(const polynomial& other);


//polynomial.cpp
//works, but does not allow chain assignment like p1 = p2 = p3;
/*
void polynomial::operator=(const polynomial& other) {
...
}
*/
polynomial& polynomial::operator=(const polynomial& other) {
  if(&other != this) {
    delete[] coefficients;

    degree = other.degree;
    coefficients = new double[degree+1];

    for(int i=0; i<=degree; i++) {//not i<degree (coefficients size = degree+1)
      coefficients[i] = other[i];
    }
  }

  return *this;
}
```

# Yep, Fourth (and last) run!

```
/* Output:
./prog
coefficients[0] = 0
coefficients[1] = 0
coefficients[2] = 0
P(x) =

coefficients[0] = 2
coefficients[1] = 3
coefficients[2] = 1
P(x) =  + 2.x^0 + 3.x^1 + x^2

coefficients[0] = 2
coefficients[1] = 3
coefficients[2] = 1
P(x) =  + 2.x^0 + 3.x^1 + x^2

coefficients[0] = 5
coefficients[1] = -4
coefficients[2] = 3
P(x) =  + 5.x^0 -4.x^1 + 3.x^2

coefficients[0] = 5
coefficients[1] = -4
coefficients[2] = 3
P(x) =  + 5.x^0 -4.x^1 + 3.x^2
*/
```

```
/*
coefficients[0] = 2
coefficients[1] = 3
coefficients[2] = 1
P(x) =  + 2.x^0 + 3.x^1 + x^2

coefficients[0] = 2
coefficients[1] = 3
coefficients[2] = 1
P(x) =  + 2.x^0 + 3.x^1 + x^2

coefficients[0] = 2
coefficients[1] = 3
coefficients[2] = 1
P(x) =  + 2.x^0 + 3.x^1 + x^2

polynomial destructor
polynomial destructor
polynomial destructor
polynomial destructor
*/
```

# Objects and dynamic memory - Conclusion

**User-defined *Big Three***

1. Copy constructor
2. Assignment operator
3. Destructor

# Objects and dynamic memory - Class polynomial I

## 1) Copy constructor

```
//polynomial.hpp
polynomial(const polynomial& other);


//polynomial.cpp
polynomial::polynomial(const polynomial& other) {
  degree = other.degree; //or other.get_degree();, both work
  coefficients = new double[degree+1];

  for(int i=0; i<=degree; i++) {//not i<degree (coefficients size = degree+1)
    coefficients[i] = other[i];
  }
}
```

# Objects and dynamic memory - Class polynomial II

## 2) Assignment operator

```
//polynomial.hpp
polynomial& operator=(const polynomial& other);


//polynomial.cpp
polynomial& polynomial::operator=(const polynomial& other) {
  if(&other != this) {
    delete[] coefficients;

    degree = other.degree;
    coefficients = new double[degree+1];

    for(int i=0; i<=degree; i++) {//not i<degree (coefficients size = degree+1)
      coefficients[i] = other[i];
    }
  }

  return *this;
}
```

# Objects and dynamic memory - Class polynomial III

## 3) Destructor

```
//polynomial.hpp
~polynomial();


//polynomial.cpp
polynomial::~polynomial() {
  std::cout << "polynomial destructor" << std::endl;
  delete[] coefficients;
}
```

# Operator overloading: as member or not member (friend)?

```
//polynomial.hpp

//Operator overloading: as member functions (MUST BE)
polynomial& operator=(const polynomial& other);
double& operator[](int i);
const double& operator[](int i) const;

//Operator overloading: as friend function (non-member) (MUST BE)
friend std::ostream& operator<<(std::ostream& os, const polynomial& p);

//Operator overloading: as member functions or friend function (non-member)
//(BOTH ARE POSSIBLE)
//v1: as member function
/*
bool operator==(const polynomial& other);
bool operator!=(const polynomial& other);
bool operator<(const polynomial& other);
bool operator<=(const polynomial& other);
*/

//v2: as friend functions (Good Practise). WHY?
friend bool operator==(const polynomial& p1, const polynomial& p2);
friend bool operator!=(const polynomial& p1, const polynomial& p2);
friend bool operator<(const polynomial& p1, const polynomial& p2);
friend bool operator<=(const polynomial& p1, const polynomial& p2);
```

# Example: addition of point and int - problematic

```
int main() {
  point p(1, 2);

  std::cout << p + 5 << std::endl; //does not work
  std::cout << 5 + p << std::endl; //does not work
}
```

```
error: no match for 'operator+'(operand types are 'point'and 'int')
std::cout << p + 5 << std::endl;

error: no match for 'operator+'(operand types are 'int'and 'point')
std::cout << 5 + p << std::endl;
```

# Example: addition of point and int - solution 1 I

## point.hpp

```
//overloading addition operator as member function
point operator+(const point& other) const;
point operator+(int n) const;
```

# Example: addition of point and int - solution 1 II

point.cpp

```
//overloading addition operator as member function
point point::operator+(const point& other) const {
  return point(x + other.x, y + other.y);
}

point point::operator+(int n) const {
  return point(x + n, y + n);
}
```

# Example: addition of point and int - solution 1 III

main.cpp

```
int main() {
  point p(1, 2);

  std::cout << p + 5 << std::endl; //works, [6, 7]
  //std::cout << 5 + p << std::endl; //does not work
}
```

```
error: no match for 'operator+'(operand types are 'int'and 'point')
std::cout << 5 + p << std::endl;
```

# Example: addition of point and int - solution 2 I

## point.hpp

```
//overloading addition operator as friend function
//note: no 'const' at the end of declaration
//      (does it make sense to have one?)
//otherwise error: non-member function 'point operator+(...)'
//                  cannot have cv-qualifier [...] const
friend point operator+(const point& p1, const point& p2);
friend point operator+(const point& p, int n);
friend point operator+(int n, const point& p);
```

# Example: addition of point and int - solution 2 II

## point.cpp

```
//overloading addition operator as friend function
point operator+(const point& p1, const point& p2) {
  return point(p1.x + p2.x, p1.y + p2.y);
}

point operator+(const point& p, int n) {
  return point(p.x + n, p.y + n);
}

point operator+(int n, const point& p) {
  return point(p.x + n, p.y + n);
}
```

# Example: addition of point and int - solution 2 III

main.cpp

```
int main() {
  point p(1, 2);

  std::cout << p + 5 << std::endl; //works, [6, 7]
  std::cout << 5 + p << std::endl; //works, [6, 7]
}

/* Output:
  [6, 7]
  [6, 7]
*/
```

# Other example: comparison

```cpp
#include <iostream>

class A {};

class B {};

int main() {
    A a;
    B b;

    std::cout << "a == b?" << (a == b) << std::endl;
    std::cout << "b == a?" << (b == a) << std::endl;

    std::cout << "a < b?" << (a < b) << std::endl;
    std::cout << "b < a?" << (b < a) << std::endl;
}

/* Output:
    error: no match for 'operator==' (operand types are 'A' and 'B')
    error: no match for 'operator==' (operand types are 'B' and 'A')

    error: no match for 'operator<' (operand types are 'A' and 'B')
    error: no match for 'operator<' (operand types are 'B' and 'A')
*/
```

# Other example : comparison between point and labeled_point (or complex)

```
#include "point.hpp"
#include "labeled_point.hpp"

int main {
  point p(1, 2);
  labeled_point lp(2, 3, "A");

  cout << "p == lp? " << (p == lp) << endl;
  cout << "lp == p? " << (lp == p) << endl;

  cout << "p < lp? " << (p < lp) << endl;
  cout << "lp < p? " << (lp < p) << endl;
}

/* Output:
  error: no match for 'operator+' (operand types are 'point' and 'point')
  error: no match for 'operator+' (operand types are 'labeled_point' and 'labeled_point'

  error: no match for 'operator+' (operand types are 'point' and 'labeled_point')
  error: no match for 'operator+' (operand types are 'labeled_point' and 'point')
*/
```

# Operator overloading: `as member or not member (friend)?`

## Good practise

- Whenever it is possible, overload operators as **friend functions** to customise them to your need
- for a binary `operatorXX`:
  `operatorXX(lhs, rhs)` is more flexible than
  `operatorXX(rhs)`

# A closer eye at std::vector

```
#include <iostream>
#include <fstream>
#include <vector>
#include "point.hpp"
using namespace std;

int main() {
  vector<point> vp;
  cout << "\nvp.size()=" << vp.size() << "\nvp.capacity()=" << vp.capacity() << "" << en

  ifstream infile;
  infile.open("points.txt");

  double x, y;
  int i=0;

  while(infile >> x >> y) {
    i++;
    cout << "\ni=" << i << endl;
    cout << "push_back of x=" << x << ", y=" << y << "" << endl;
    /*point tmp(x, y);
    vp.push_back(tmp);*/
    vp.push_back(point(x, y));
    cout << "vp.size()=" << vp.size() << "\nvp.capacity()=" << vp.capacity() << "" << en
  }

  infile.close();

  cout << "\nvp.size()=" << vp.size() << "\nvp.capacity()=" << vp.capacity() << "" << en

  cout << "\nall " << vp.size() << " points: " << endl;
  for(int i=0; i<vp.size(); i++) {
    cout << "vp[" << i << "]= " << vp[i] << endl;
  }
```

# std::vector of points: Output

points.txt

```
1 2
3 4
5 6
7 8
9 10
```

```
$ make run
./prog

vp.size()=0
vp.capacity()=0

i=1
push_back of x=1, y=2
point [1, 2] is leaving
vp.size()=1
vp.capacity()=1

i=2
push_back of x=3, y=4
point [1, 2] is leaving
point [3, 4] is leaving
vp.size()=2
vp.capacity()=2

i=3
push_back of x=5, y=6
point [1, 2] is leaving
point [3, 4] is leaving
point [5, 6] is leaving
vp.size()=3
vp.capacity()=4
```

```
i=4
push_back of x=7, y=8
point [7, 8] is leaving
vp.size()=4
vp.capacity()=4

i=5
push_back of x=9, y=10
point [1, 2] is leaving
point [3, 4] is leaving
point [5, 6] is leaving
point [7, 8] is leaving
point [9, 10] is leaving
vp.size()=5
vp.capacity()=8

vp.size()=5
vp.capacity()=8

all 5 points:
vp[0]= [1, 2]
vp[1]= [3, 4]
vp[2]= [5, 6]
vp[3]= [7, 8]
vp[4]= [9, 10]
```

```
point [1, 2] is leaving
point [3, 4] is leaving
point [5, 6] is leaving
point [7, 8] is leaving
point [9, 10] is leaving
```

# Exercice (Constructor/Destructor)

```
#include <iostream>

class A {
  public:
    A(): n(0) { std::cout << "constructor A(). n=" << n << std::endl; }

    A(int n_in): n(n_in) { std::cout << "constructor A(int). n=" << n << std::endl; }

    ~A() { std::cout << "destructor ~A(). n=" << n << " is leaving" << std::endl; }

  private:
    int n;
};


int main() {
  {
    std::cout << "Hi" << std::endl;
    A a1;
    A a2(5);
  }

  std::cout << "\nHi again" << std::endl;
  A a1;

  A* a_ptr1 = &a1;
  a_ptr1 = new A(7);
  delete a_ptr1;

  A a2(5);

  A* a_ptr2 = new A(10);
}
```

# Exercice (Constructor/Destructor) - Answer

```
int main() {
  {
    std::cout << "Hi" << std::endl;
    A a1;
    A a2(5);
  }

  std::cout << "\nHi again" << std::endl;
  A a1;

  A* a_ptr1 = &a1;
  a_ptr1 = new A(7);
  delete a_ptr1;

  A a2(5);

  A* a_ptr2 = new A(10);
}
```

```
/* Output:
  Hi
  constructor A(). n=0
  constructor A(int). n=5
  destructor ~A(). n=5 is leaving
  destructor ~A(). n=0 is leaving

  Hi again
  constructor A(). n=0
  constructor A(int). n=7
  destructor ~A(). n=7 is leaving
  constructor A(int). n=5
  constructor A(int). n=10
  destructor ~A(). n=5 is leaving
  destructor ~A(). n=0 is leaving
*/
```

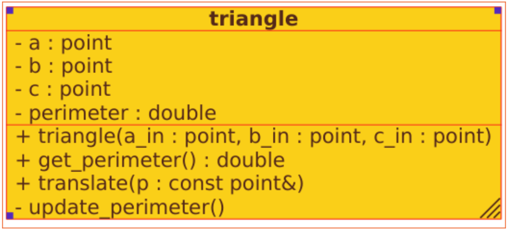
# Outline

# triangles and points (Week 2 - Lab): triangle.hpp

```
#ifndef TRIANGLE_HPP
#define TRIANGLE_HPP

#include <iostream> //ostream, used in operator<<

#include "point.hpp"

class triangle {
  public:
    triangle(const point& a_in, const point& b_in, const point& c_in);
    double get_perimeter() const;

    void translate(const point& p);

    friend std::ostream& operator<<(std::ostream& os, const triangle& t);

  private:
    point a;
    point b;
    point c;
    double perimeter;

    void update_perimeter();
};

#endif
```

# UML diagram of class triangle

- A class:
  - in UML: {name, attributes, operations}
  - in C++: {name, member variables, member functions}




- can also add another constructor

```
triangle(double, double double, double, double, double);
```

# triangles and points (Week 2 - Lab): triangle.cpp

```cpp
#include <iostream>
#include "triangle.hpp"

triangle::triangle(const point& a_in, const point& b_in, const point& c_in): a(
    a_in), b(b_in), c(c_in) {
  update_perimeter();
}

double triangle::get_perimeter() const {
  return perimeter;
}

void triangle::update_perimeter() {
  perimeter = a.distance_to(b)+ b.distance_to(c) + c.distance_to(a);
}

void triangle::translate(const point& p) {
  a.translate(p);
  b.translate(p);
  c.translate(p);
  //no need to update the perimeter after a translation
}

std::ostream& operator<<(std::ostream& os, const triangle& t) {
  os << "[" << t.a << ", " << t.b << ", " << t.c << "], perimeter = " << t.
      perimeter << std::endl;
  return os;
}
```

# main.cpp

```
#include <iostream>

#include "point.hpp"
#include "triangle.hpp"
//pre-processor guards: point.hpp included once

using namespace std;

int main() {
  /////////////////////////////////
  // PART 1: Points
  /////////////////////////////////
  cout << "#PART 1: Points" << endl;

  point p1(0, 0), p2(1, 2), p3(0.0, 0.0);

  cout << "p1 == p2: " << (p1 == p2) << endl;
  cout << "p1 != p2: " << (p1 != p2) << endl;

  cout << "p1 == p3: " << (p1 == p3) << endl;
  cout << "p1 != p3: " << (p1 != p3) << endl;

  cout << "p1 < p2: " << (p1 < p2) << endl;

  cout << "p1 = " << p1 << endl;
```

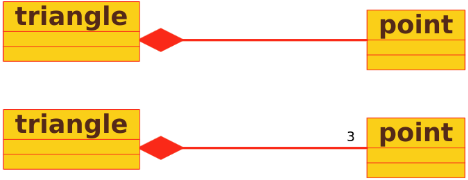
```
/////////////////////////////////
// PART 2: Triangles
/////////////////////////////////
cout << "\n#PART 2: Triangles" << endl;

p3.set_y(2);

triangle t(p1, p2, p3);
cout << t << endl;

t.translate(point(1, 1));
cout << t << endl;

//check original points: translated? NO
cout << "p1: " << p1 << endl;
cout << "p2: " << p2 << endl;
cout << "p3: " << p3 << endl;
}
```

# Output

```
Output:
$ make run
./prog
#PART 1: Points
p1 == p2: 0
p1 != p2: 1
p1 == p3: 1
p1 != p3: 0
p1 < p2: 0
p1 = [0, 0]

#PART 2: Triangles
[[0, 0], [1, 2], [0, 2]], perimeter = 5.23607

[[1, 1], [2, 3], [1, 3]], perimeter = 5.23607

p1: [0, 0]
p2: [1, 2]
p3: [0, 2]
```

# Class composition

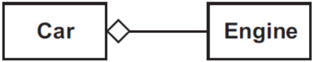
- Composition: “has-a” relationship
- Component objects share life cycle of owner class
- In UML: filled diamond and information about cardinality (or multiplicity) (and sometimes ‘role’)



# UML Note: Association and Aggregation

- **Association**: Objects of one class are associated with objects of another class
- **Aggregation**: Strong association
  an instance of one class is made up of instances of another class.
- **Composition**: Strong aggregation
  the composed object can't be shared by other objects and dies with its composer

# Examples




**Depicting aggregation in UML**

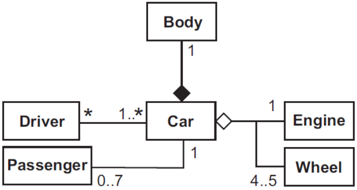

**Depicting composition in UML**


**Depicting association in UML**

# Cardinality (Multiplicity)

- n :Exactly n
- m..n : Any number in the range m to n (inclusive)
- p..* : Any number in the range p to infinity
- * : Shorthand for 0..*
- 0..1 : Optional

# Cardinality (Multiplicity) - Example




Depicting multiplicities in UML

- A Car has one Engine
- An Engine is part of one Car
- A Car has four or five Wheels
- Each Wheel is part of one Car
- A Car is always composed of one Body
- A Body is always part of one Car and it dies with that Car
- A Car can have any number of Drivers
- A Driver can drive at least one Car
- A Car has up to seven Passengers at a time
- A Passenger is only in one Car at a time

# Class labeled_point vs point: UML

```
point
-----------------------------------------
-x: double
-y: double
-distance_to_origin: double
-----------------------------------------
+point(x_in:double,y_in:double)
+get_x(): double
+get_y(): double
+get_distance_to_origin(): double
+set_x(x_in: double): void
+set_y(y_in: double): void
+distance_to(other: point): double
+to_symmetric(): void
+translate(other: point): void
```

```
labeled_point
-----------------------------------------
-x: double
-y: double
-label: string
-distance_to_origin: double
-----------------------------------------
+labeled_point(x_in:double,y_in:double,label_in:string
+get_x(): double
+get_y(): double
+get_label(): string
+get_distance_to_origin(): double
+set_x(x_in: double): void
+set_y(y_in: double): void
+set_label(label_in:string): void
+distance_to(other: point): double
+to_symmetric(): void
+translate(other: point): void
```

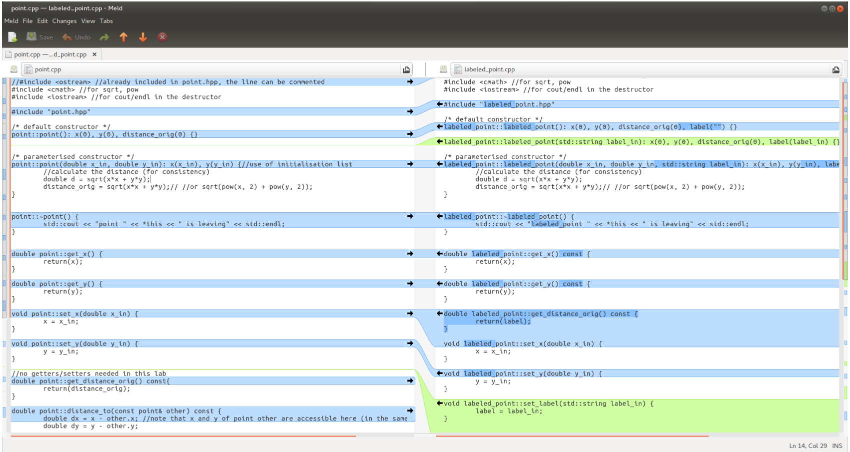
# Class `labeled_point` vs `point`: copy/paste .cpp

## Ctrl-c, ctrl-v?

- We could just copy the code of `point` and edit it but:
- Although in this case we have the source, in some cases we do not.
- What happens when we discover a bug or a way to improve the implementation which affects the base code?
- **Maintenance** becomes a nightmare.
- We are not **reusing** in the right way.

# Class labeled_point vs point: copy/paste .cpp

see code difference between point.cpp and labeled_point.cpp (using Meld or diff)



# labeled_point as composition

```
labeled_point
-------------------------------------------------------------
-p: point
-label: string
-------------------------------------------------------------
+labeled_point(p_in:point,label_in:string)
+labeled_point(x_in:double,y_in:double,label_in:string
+get_point(): point
+get_label(): string
+get_x(): double
+get_y(): double
+get_distance_to_origin(): double
+set_point(p_in:point): void
+set_label(label_in:string): void
+set_x(x_in:double): void
+set_y(y_in:double): void
+distance_to(other:labeled_point): double
+distance_to(other:point): double
+to_symmetric(): void
+translate(other: point): void
```

# labeled_point as composition

## Critiques

- duplicated code
- what about

```
point p(1, 2);
labeled_point lp(p, "A");
lp.set_x(6);
lp.get_point().set_y(6);
```

- are x and y the attributes of a `labeled_point` or `the attributes of it's point p`?
- `does a labeled_point have a point? (`*`has-a`*` relationship)`
- or is `a labeled_point already a point? (`*`is-a`*` relationship)`

# labeled_point as...?

## We would like to

- Factorize the code keeping each distinct functionality in a 'single entry point'
- Centralize corrections and improvements
- Reduce redundant, duplicated, conceptually irrelevant code
- Keep protection and **encapsulation** of member data
- Have more **flexibility** in the selective exposure of member data
- Conceptually organize classes in terms of what **extends** what

# labeled_point as a subclass (inheritance)

```cpp
#ifndef LABELD_POINT_HPP
#define LABELD_POINT_HPP

#include <string>
#include <iostream>
#include "point.hpp"

class labeled_point: public point {
  public:
    labeled_point();
    labeled_point(double x_in, double y_in, std::string label_in);

    ~labeled_point();

    std::string get_label() const;
    void set_label(std::string label_in);

    double distance_to(const labeled_point& other) const;
    void translate(const labeled_point& other);

    void to_symmetric();

    friend std::ostream& operator<<(std::ostream& os, const point& p);

  private:
    std::string label;
};

#endif
```

# Inheritance conceptually

```
class labeled_point: public point
```

- Inheritance is an ‘is-a’ relationship (for some meanings of ‘is-a’)
- Classes which inherit are called ‘**subclasses**’ of a ‘**base class**’ or ‘superclass’
- The superclass is also said to ‘generalise’ its subclasses (**inheritance** = **generalisation**))
- The subclass is also said to ‘**extend**’ its base class.

# Inheritance operationally

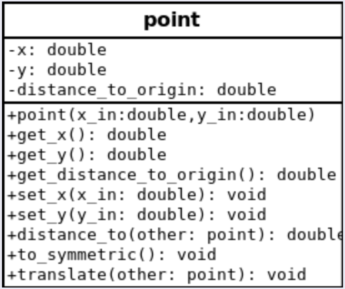
```
labeled_point::labeled_point(): x(0), y(0), distance_orig(0), label("") {}
```

```
error: double point::x is private
```

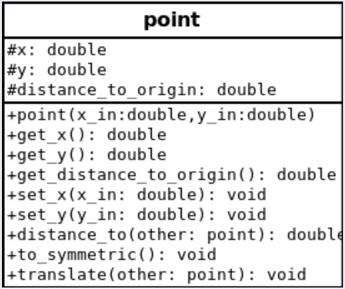
- member: **inherited** but not necessarily **accessible**

# private vs protected (access modifier)

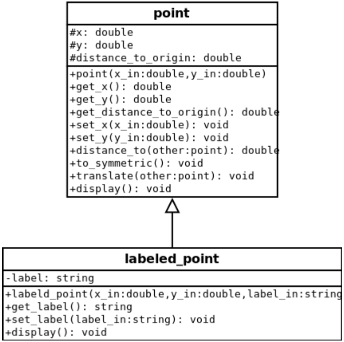
## private member variables




## protected member variables



# Inheritance: UML diagram

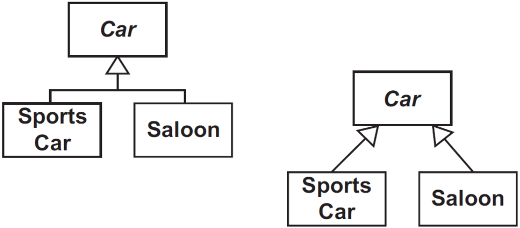



- `point` is the base class
- labeled_point is a **derived class**

- class `point` is a **generalisation** of class labeled_point
- class labeled_`point` is a **specialisation** of class point

- labeled_point **inherits** all the members of the base class `point` (including **private** ones, if any)
- labeled_point **has access** to all the **protected** and **public** members from the base class `point`

# Other examples




**Depicting inheritance in UML**

# Outline

# Summary

- Memory management
  - *Big Three*
  - std::vector
- Classes relationships: UML and C++
  - Composition
  - Inheritance (generalisation)

# What to do next?

## Next Lab

- Memory management: finish class polynomial properly
- Inheritance: class `labeled_point`

## Next Lecture

Week 6 - Polymorphism and Virtual Functions