EE2-12 Software Engineering 2: Object-Oriented Programming

Week 2 - Classes and Objects II: Constructor, and Operator Overloading

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 Objects and Dynamic Memory
- Week 4 Classes Relationships I: Association, Aggregation, and Composition
- Week 5 Classes Relationships II: 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 2 Intended Learning Outcomes (ILOs)

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

Lecture

1. Write classes constructors to instantiate objects
2. Apply operator overloading to member functions of a class
3. Understand the different uses of `const` in members functions (passing parameters by const references, const member functions)

Lab

1. Write a class declaration and definition in C++, and test it by instantiating objects using constructors
2. Overload basic operators, and test them
3. Build a basic C++ software architecture using a Makefile

Outline

Outline

Week 1 Classes and Objects I: Introduction I

- programming paradigm shift: procedural $\triangleright$ object-oriented
  - **classes** vs **objects** (user defined type vs variables, one class vs many objects/instances)
  - oop $\simeq$ writing classes

Encapsulation

- binding data (**member variables**) and behaviour (**member functions**) in the same entity
- access modifiers **public**/**private**

- Good practise:
  - **private** member variables
  - **public** member functions, including variables accessors (`type get_var()`) and mutators (`void set_var(value)`)

Week 1 Classes and Objects I: Introduction II

Operators

- **dot operator** . (member functions calls on objects: `obj.f()`)
- **scope resolution operator** `::`
  (`type_qualifier class_name::function(...)`)

Abstraction

show the class interface (What?), hide the implementation details (How?)

Week 1 Classes and Objects I: Introduction III

1 class, 2 files

- **header file** (**.hpp**): member variables and member functions **declaration**[a]

  ```
  return_type function_name(arguments);
  ```

- **source file** (**.cpp**): member functions **implementation**

  return_type class_name::function_name(arguments) //body

[a] *prototype* (all the declaration) vs *signature* (name and params list only)

Week 1 Classes and Objects I: Introduction IV

Building an application

1. **Separate compilation**

   `g++ -c class1.cpp`

   `g++ -c class2.cpp`

   `[...]`

   `g++ -c main.cpp`

   if success: object files `class1.o`, `class2.o`, `[...]`, `main.o`

2. **Linking**

   `g++ class1.o class2.o [...] main.o -o prog`

   if success: executable file `prog`

3. **Execution**

   `./prog`

(see `man g++` for more details. see later how to use a basic **Makefile**.)

Week 1 Classes and Objects I: Introduction V

UML class diagram

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

- language-independant
- equivalent to `point.hpp` in C++
- implementation details not compulsory (`x_in`, `y_in`, `other`)

Week 1 Classes and Objects I: Introduction VI

Lab: main pitfalls

- ; after class declaration

- ~~`#include <point.hpp>`~~ or
- ~~`#include 'point.hpp'`~~ instead of
- `#include "point.hpp"`

- `point::` before functions definitions in `point.cpp`

- writing the whole class/main before the first compilation
- no writing comments

- state consistency

Makefile

```
all: point.o main.o
	g++ point.o main.o -o prog

point.o: point.cpp point.hpp
	g++ -c point.cpp

main.o: main.cpp
	g++ -c main.cpp

run:
	./prog
#./prog.exe if Windows/Cygwin

clean:
	rm *.o prog
# rm *.o prog.exe if Windows/Cygwin

#structure
#target: [dependencies list ...]
# commands #starts with a tabulation
```

Makefile use I

```
$ ls
main.cpp Makefile point.cpp point.hpp

$ make point.o
g++ -c point.cpp

$ ls
main.cpp Makefile point.cpp point.hpp point.o

$ make main.o
g++ -c main.cpp

$ ls
main.cpp main.o Makefile point.cpp point.hpp point.o

$ make all
g++ point.o main.o -o prog

$ make run
```

Makefile use II

```
./prog
[...execution output...]

$ make clean
rm *.o prog

$ ls
main.cpp Makefile point.cpp point.hpp

$ make
g++ -c point.cpp
g++ -c main.cpp
g++ point.o main.o -o prog

$ make point.o
make: 'point.o' is up to date.
```

Outline

Function overloading - Definition

- Function overloading allows to create multiple functions (member or not) with the **same name**, so long as they have **different parameters**.
- When overloading functions, the definition of the function must differ from each other by the **types** and/or the **number of arguments** in the argument list (*ie* the **signature**).

Function overloading - Example

```cpp
#include <iostream>
#include <string>

void f() {
  std::cout << "f()" << std::endl;
}

void f(int n) {
  std::cout << "f(int) " << n << std::endl;
}

void f(double d) {
  std::cout << "f(double) " << d << std::endl;
}

void f(std::string s) {
  std::cout << "f(string) " << s << std::endl;
}

int main() {
  f();
  f(1);
  f(2.5);
  f("ee2-12");
  //f('b'); //not allowed

  return 0;
```

```
/* Output:
  f()
  f(int) 1
  f(double) 2.5
  f(string) ee2-12
*/
```

Function overloading

You can not overload function declarations that differ only by return type.

The following declaration results in an error.

```
int f(int a) { }
double f(int b) { }
std::string f(int c) { }
```

"*Keep resolution for an individual operator or function call context-independent*". [Bjarne Stroustrup, "The C++ Programming Language"]

References

```cpp
#include <iostream>
int main() {
  int x = 1; //int
  int &r = x; //int reference
  int *p = &x; //int pointer

  std::cout << "x=" << x << std::endl;
  std::cout << "r=" << r << std::endl;
  std::cout << "p=" << p << std::endl;

  std::cout << "&r=" << &r << std::endl;
  std::cout << "*p=" << *p << std::endl;

  r = 5;

  std::cout << "x=" << x << std::endl;

  return 0;
}
```

```
/* Output:
  x=1
  r=1
  p=0x22cc74
  &r=0x22cc74
  *p=1

  x=5
*/
```

a reference is an *alias* of a variable

Passing arguments by reference (call by reference)

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

void swap(int &x, int &y) {
   int temp;
   temp = x;
   x = y;
   y = temp;

   return;
}

int main () {
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a: " << a << endl;
   cout << "Before swap, value of b: " << b << endl;

   swap(a, b);

   cout << "\nAfter swap, value of a: " << a << endl;
   cout << "After swap, value of b: " << b << endl;

   return 0;
}
```

```
/* Output:
  Before swap, value of a: 100
  Before swap, value of b: 200

  After swap, value of a: 200
  After swap, value of b: 100
*/
```

Passing arguments by value (call by value)

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

void swap(int x, int y) {
   int temp;
   temp = x;
   x = y;
   y = temp;

   return;
}

int main () {
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a: " << a << endl;
   cout << "Before swap, value of b: " << b << endl;

   swap(a, b);

   cout << "\nAfter swap, value of a: " << a << endl;
   cout << "After swap, value of b: " << b << endl;

   return 0;
}
```

```
/* Output:
  Before swap, value of a: 100
  Before swap, value of b: 200

  After swap, value of a: 100
  After swap, value of b: 200
*/
```

Call by value vs call by reference I

Call by value

- copy of the **arguments** (`a` and `b`) values into the **formal parameters** (`x` and `y`)
- changes made to the parameters inside the function have no effect on the argument.

Call by reference

- copy of the references of **arguments** (`a` and `b`) into the **formal parameters** (`&x` and `&y`)
- changes made to the parameters inside the function also affect the arguments

Call by value vs call by reference II

Question: what about using pointers?

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

void swap(int *x, int *y) {
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;

   return;
}

int main () {
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a: " << a << endl;
   cout << "Before swap, value of b: " << b << endl;

   swap(&a, &b);

   cout << "\nAfter swap, value of a: " << a << endl;
   cout << "After swap, value of b: " << b << endl;
```

```
/* Output:
  Before swap, value of a: 100
  Before swap, value of b: 200

  After swap, value of a: 200
  After swap, value of b: 100
*/
```

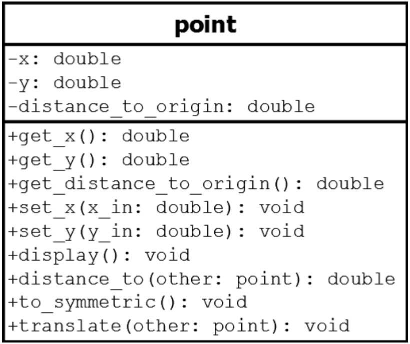
const modifier

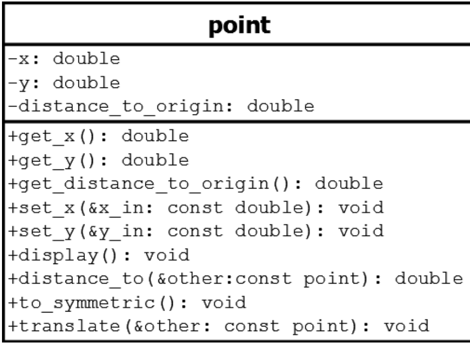
```
#include <iostream>
const double PI = 3.14;

double perimeter(double radius) {
  return 2*PI*radius;
}

double area(double radius) {
  return PI*radius*radius;
}

int main() {
  //PI = 3.141; //not allowed
  //error: assignment of read-only variable 'PI'

  double r = 1.0;
  std::cout << "perimeter=" << perimeter(r) << std::endl;
  std::cout << "area=" << area(r) << std::endl;

  return 0;
```

Call by const reference

- Call by const reference:
  - No need to copy
  - No need to worry about changes
- In C++ const is for more than just defining constants
- Const correctness

point again: call by const reference






```
double distance_to(const point &other);
```

```
double translate(const point &other);
```

const member functions

```
double point::distance_to(const point &other) {
  double delta_x = x - other.get_x(); //other.x works too
  double delta_y = y - other.get_y(); //other.y works too

  return sqrt(delta_x * delta_x + delta_y * delta_y);
}
```

- No change on the current state (local `x` and `y` are not even read!)
- It's useful to mark this: `const` **member function**
- (Notice that from the same class also `private` member data of other objects can be accessed.)

const member functions: declaration/prototype

`const` at the end of the member function prototype:

```
double distance_to(const point &other) const;
```

Details

1. `double`: returns a double
2. `distance_to`: name of the member function
3. `const point &other`: passing other, a point argument, by const reference
    1. const: `distance_to` will not attempt to modify `other`'s state
    2. reference: no local copy of `other`
4. `const;`: const member function, `distance_to` will not attempt to modify the current state

const member functions: definition

```
double distance_to(const point &other) const;
```

```
double point::distance_to(const point &other) const {
  double delta_x = x - other.get_x(); //other.x works too
  double delta_y = y - other.get_y(); //other.y works too

  return sqrt(delta_x*delta_x + delta_y*delta_y); //or pow(
}
```

Outline

Constructor

- Initialisation of objects
  - Initialise some or all member variables
  - Other actions possible as well
- A special kind of member function
  - Automatically called when object declared
- Very useful tool (key principle of OOP)

- Defined like any member function
  Except:
  1. Must have same name as clas
  2. Cannot return a value; not even void!

Constructor - Example

```
point(double x_in , double y_in);
```

- Notice name of constructor: `point` (same name as class itself!)
- Constructor declaration has no return-type (not even void!)
- Constructor in public section
  - It's called when objects are declared
  - If private, could never declare objects!

Calling Constructors

- Declare objects:

```
point p1(1.0, 2.5);
point p2(5.0, 10.0);
```

- Objects are created here
  - Constructor is called
  - Values in parens passed as arguments to constructor
  - Member variables `x`, and `y` initialised

Constructor Equivalency

- Consider:

```
point p1, p2;
p1.point(1.0, 2.5); //ILLEGAL
p1.point(5.0, 10.0); //ILLEGAL
```

- Seemingly OK...
  - CANNOT call constructors like other member functions!

Constructor Implementation

```
point::point(double x_in, double y_in) {
  x = x_in;
  y = y_in;
}
```

- Constructor definition is like all other member functions
- Note same name around `::` clearly identifies a constructor
- Note no return type: just as in class definition

Constructor Implementation - Initialisation list

```
point::point(double x_in, double y_in): x(x_in), y(y_in) {}
```

- TODO
- TODO

Default constructor

- **If and only if no constructor is defined**, a default one (with no arguments) is created by the compiler
  (This is what happened every time we declared a point object, e.g. `point p1;` was calling the default constructor, created by the compiler, of class point.)

- If we define the constructor it's less likely that an object is initialised to a meaningless state

```
point::point() {
  x = 0.0;
  y = 0.0;
}
```

Default arguments (for any function, member or not)

```
point(double x_in = 0.0, double y_in = 0.0);
```

```
point::point(double x_in /*= 0.0*/, double y_in /*= 0.0*/) {
  /* the comment reminds defaults in the declaration */
  x = x_in;
  y = y_in;
}
```

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?

[see after few slides]

- 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

Outline

Operator overloading

```
int n = 10;

cout << (n == n) << endl;
//compares two int
//prints a bool (1) (on the stdout)

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 strings 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.

Point with ==

- We would still like to test equality between points with ==
- if(p1 == p2){
  seems more readable than
  if((p1.get_x() == p2.get_x()) && (p1.get_y() == p2.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?

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.

Point with <

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

Outline

Summary

- **function overloading**: We can define functions (member or not) with the same name and different behaviour (implementation), as long as they have a different parameters list ("signature")
- functions overloading (overload on the signature (arguments), can't overload on the return type)
- `const` **correctness**, call by `const` reference, `const` member functions
- **constructor(s)**
- operators overloading: ==, +, <

What to do next?

Next Lab

- constructor(s), operator overloading and tests on class `point`, class `triangle`
- `const` correctness, call by `const` reference, `const` member functions
- `Makefile`

Next Lecture

Week 3 - Objects and Dynamic Memory