EE2-12 Software Engineering 2: Object-Oriented Programming

Week 3 - Objects and Dynamic Memory

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 3 Intended Learning Outcomes (ILOs)

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

Lecture

1. Extend operator overloading to the insertion and assignment operators (**operator<<** and **operator=**)
2. Write classes **destructors** and understand their call mechanism
3. Apply the **dynamic memory allocation** to create new objects, and use **pointers** and **references**

Lab

1. Write a complete C++ class declaration and definition including a copy constructor and a destructor, and test it.
2. Apply operator overloading to operator<< and operator=
3. Apply dynamic memory allocation and understand the use of pointers and references
4. Apply operator overloading to operator[]

Outline

1. Introduction
   - Week 2 Summary
2. More on classes
   - Constructors
   - More on Operator Overloading
3. Dynamic memory
   - “new” keyword
   - Destructor
   - delete
4. Conclusion

Outline

1. Introduction
   - Week 2 Summary
2. More on classes
3. Dynamic memory
4. Conclusion

Week 2 Classes and Objects II: Constructors, and Operator Overloading I

Function overloading

- functions with the same name and different behaviour
  `void f();` and `void f(int n);` and `void f(double d);`
- different **signature**
- no overloading on the return type

Week 2 Classes and Objects II: Constructors, and Operator Overloading II

Constructor

- called on objects declaration

```
point(double x_in , double y_in);
point::point(double x_in, double y_in) ...
point p(2, 3);
```

- **default constructor**: created by the compiler if and only if no constructor is defined

```
point p;
```

Default arguments

- only trailing parameters

Week 2 Classes and Objects II: Constructors, and Operator Overloading III

const correctness

- call by value vs call by reference
  `void swap(int x, int y)` vs `void swap(int &x, int &y)`
- call by `const` reference (no local copy, read-only)
- `const` member functions

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

Week 2 Classes and Objects II: Constructors, and Operator Overloading IV

Operator overloading

- **Friend** functions

```
friend bool operator==(const point& p1 , const point& p2);
if(p1 == p2) ...
```

Makefile

Week 2 - Lab I

Lab: main pitfalls

- Separate compilation (see Week 2 Lecture [Building an application])
- `#include "point.cpp"`
- `g++ -c class.hpp`
- multiple inclusion of `point.hpp`
  (`error: redefinition of class point`)
- operators overloading: <, ==, <<
- Makefile: syntax, tabulation (`missing separator`), filename, use

Week 2 - Lab II

Never, ever

- include a .cpp file (~~`#include "point.cpp"`~~)
- compile a .hpp file (if done accidentally, remove any *.gch files) (~~g++ point.hpp~~)
- add `using namespace std;` in a .hpp file (use `std::xxx` instead)
- write a header file without include guards (to avoid multiple inclusions)

Week 2 - Lab III

Good practise

- in a .cpp file, `using namespace xxx;` directive should be written AFTER the `#include` statements (to avoid name conflicts)
- in a .hpp file, systematically add pre-processor guards
  #ifndef CLASS_HPP
  #define CLASS_HPP
  ...
  #endif
- minimise the `#include` directives in .hpp files (e.g. no `<iostream>` in triangle.hpp)

Week 2 - Lab IV

Good practise for operator overloading

- overloading built-in C++ operators: as friend functions (not member functions)

```
friend bool operator==(const point& p1, const point& p2);
friend bool operator<(const point& p1, const point& p2);
std::ostream& operator<<(std::ostream& os, const point& p);
```

Week 2 - Lab: possible continuation

- operator!= for class point

```
bool operator!=(const point& p1, const point& p2);
```

- operator overloading for class triangle

```
friend bool operator==(const triangle& t1, const triangle& t2);
friend bool operator<(const triangle& t1, const triangle& t2);
std::ostream& operator<<(std::ostream& os, const triangle& t);
```

- operator+ for class point. and triangle?

Outline

1 Introduction

2 More on classes
- Constructors
- More on Operator Overloading

3 Dynamic memory

4 Conclusion

Initialisation list

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

- We have a chance to construct our objects “before the curly brace”
- Declaration is separated from construction
- Not only for C++ elegance (it is the only way to initialise const variables for example)

Initialisation list - Example

```
#include <iostream>

class A {
  public:
    A();
    A(int max_in);

  private:
    const int MAX;
};

int main() {
  A a(100);
}

A::A(){}

A::A(int max_in) {
  MAX = max_in;
}
```

Copy constructor

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

```
point p2(p1);
```

- initialise a point from another point

Constructors: question 1

```cpp
#include <iostream>

class A {
  public:
    //no user-defined constructor

    void display();

  private:
    int n;
};

int main() {
  A a;
  a.display();
}

void A::display() {
  std::cout << "n = " << n << std::endl;
}
```

Constructors: question 2

```
#include <iostream>

class A {
  public:
    A(int n_in); //parameterised constructor

    void display();

  private:
    int n;
};

int main() {
  A a;
  a.display();
}

void A::display() {
  std::cout << "n = " << n << std::endl;
}

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

Constructors: question 3

```
#include <iostream>

class A {
  public:
    A(); //default constructor
    A(int n_in); //parameterised constructor

    void display();

  private:
    int n;
};

int main() {...}

void A::display() {...}

A::A() { //default constructor
  n=0; //or A(0);
  std::cout << "constructor: A()"<< std::endl;
}

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

Constructors: question 4

```
#include <iostream>

class A {
  public:
    A(); //default constructor
    A(int n_in = 0); //parameterised constructor with default value

    void display();

  private:
    int n;
};

int main() {...}

void A::display() {...}

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

A::A(int n_in /* = 0 */) {
  n = n_in;
  std::cout << "constructor: A(int=0)"<< std::endl;
}
```

Constructors: question 5

```
#include <iostream>

class A {
  public:
    A(); //default constructor
    A(int n_in); //parameterised constructor with initialisation list

    void display();

  private:
    int n;
};

int main() {
  A a1; a1.display();

  A a2(5); a2.display();
}

void A::display() {...}

A::A(): n(0) {
  std::cout << "constructor: A()"<< std::endl;
}

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

Constructors: question 6

```
#include <iostream>

class A {
  public:
    A(int n_in, int m_in); //parameterised constructor with initialisation list
    A();
    void display();

  private:
    int n;
    int m;
};

int main() {
  A a1; a1.display();

  A a2(2, 3); a2.display();
}

void A::display() {...}

A::A(int n_in, int m_in): n(n_in), m(m_in) {}

A::A(): n(0), m(0){
  //or A(0, 0);
}
```

operator[] overloading

- see Week 3 - Lab, class `equation`

Outline

1. Introduction
2. More on classes
3. Dynamic memory
   - "new" keyword
   - Destructor
   - delete
4. Conclusion

Dynamic memory allocation for objects

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

using namespace std;

int main() {
  point* p1 = new point();
  point* p2 = new point(1,2);

  cout << p1->get_x() << " " << p1->get_y() << endl;
  cout << p2->get_x() << " " << p2->get_y() << endl;

  return 0;
}
```

The destructor I

~point();

```
class point{
  public:
     . . .
     ~point() {
       cout << "point " << x << " " << y << " is leaving" <<
     }
     . . .
};
```

The destructor II

```cpp
int main() {
  //the following curly brace is not a typo ...
  {
    cout << "a new scope begins" << endl;
    point p1;
    point* p2 = new point(1 ,2);
    cout << "the new scope ends" << endl;
  }

  cout << "goodbye everyone!" << endl;

  return 0;
}

/* Output:
  a new scope begins
  the new scope ends
  point 0 0 is leaving
```

The destructor III

```
int main() {
  {
    cout << "a new scope begins" << endl;
    point p1;
    point* p2 = new point(1 ,2);
    cout << "the new scope ends" << endl;
  }
  //delete p2;
  //error: 'p2' was not declared in this scope

  cout << "goodbye everyone!" << endl;

  return 0;
}
```

The destructor IV

```
int main() {
  {
    cout << "a new scope begins" << endl;
    point p1;
    point* p2 = new point(1 ,2);
    delete p2;
    cout << "the new scope ends" << endl;
  }

  cout << "goodbye everyone!" << endl;

  return 0;
}

/* Output:
  a new scope begins
  point 1 2 is leaving
  the new scope ends
```

The destructor V

- In the previous example: defined our destructor in order to better understand when it is called, what happens to dynamic memory etc.
- As far as `point` is concerned, even without our destructor, no memory leaks as long as `delete` is called on dynamically created objects.
- This is not always the case.
- There can also be other reasons to define a destructor.

Outline

1. Introduction
2. More on classes
3. Dynamic memory
4. Conclusion

Summary

- operators overloading: $==$, $+$, $<$
- dynamic memory allocation
- `new`
- destructor, `delete`

What to do next?

Next Lab

- More on operator overloading (`operator[]`, `operator=`)
- More on dynamic allocation (`new`, `destructor`, `delete`)

Next Lecture

Week 3 end - More on Dynamic Memory
Week 4 - Classes Relationships I: Association, Aggregation, and Composition