EE2-12 Software Engineering 2: Object-Oriented Programming

Week 4 - 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 4	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 3 Intended Learning Outcomes (ILOs)

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

Lecture

1. Write classes **destructors** and understand their call mechanism
2. Write classes **copy constructors** and apply operator overloading to the **assignment operator** (**operator=**)
3. Unserstand the **dynamic memory allocation** to create new objects, and the use of **pointers** and **references**

Lab

1. Apply operator overloading to the **assignment operator** (operator=)
2. Apply the **copy constructor** to construct new objects from existing ones
3. Apply **dynamic memory allocation** and understand the use of **pointers** and references

Problematic: copying, assigning

```
class point {
  public:
    point(double x_in, double y_in) : x(x_in), y(y_in) {}
  //...
  private:
    double x;
    double y
};

int main() {
  point p(2.0, 3.0);

  point p1(p); // What happens here?
  point p2 = p; // And here?

  point* p3; //pointer to a point
  p3 = &p; // And here as well?

  point* p4 = new point(p); //Yet another "what happens here?"
}
```

Outline

Outline

Bjarne Stroustrup - Draper Prize 2018

- The 2018 Charles Stark Draper Prize for Engineering is awarded to Dr. Bjarne Stroustrup “For conceptualizing and developing the C++ programming language.”
- annual award, by the U.S. National Academy of Engineering
- one of three prizes that constitute the "*Nobel Prizes of Engineering*"

- Youtube video https://www.youtube.com/watch?v=fS2QF9uPMWc (start at 7’53" https://youtu.be/fS2QF9uPMWc)

Week 3 More on Classes, Objects, and Operator Overloading I

Constructors

- called on objects declaration

  `point p(2, 3);`
- **default constructor**: created by the compiler if and only if no constructor is defined

  `point p;`
- default values
- initialisation list

```
point::point(double x_in=0, double y_in=0) : x(x_in), y(y_in)
```

Week 3 More on Classes, Objects, and Operator Overloading II

const correctness

- call by const reference

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

- const member functions

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

Week 3 More on Classes, Objects, and Operator Overloading III

Operator overloading

- Binary comparison operators:

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

- Insertion operator

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

Week 3 More on Classes, Objects, and Operator Overloading IV

Pre-processor guards

```
#ifndef POINT_HPP
#define POINT_HPP
...
#endif
```

Makefile

- separate compilation
- linking
- running
- cleaning
- (new features in the lab: **Implicit Rules** and **variables**)

Week 3 Lab - main pitfalls/tips I

const-correctness

```
double distance_to(const point& other);
friend bool operator<(const point& p1, const point& p2);

bool operator<(const point& p1, const point& p2) {
  point origin(0, 0);
  return (p1.distance_to(origin) < p2.distance_to(origin));
}
```

```
error: passing ‘const point’ as ‘this’ argument discards
qualifiers [-fpermissive]
return (p1.distance_to(origin) < p2.distance_to(origin));
note: in call to ‘double point::distance_to(const point&)
```

$\hookrightarrow$ make `distance_to` a `const` member function

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

Week 3 Lab - main pitfalls/tips II

Operator overloading, friend

error: ‘friend’ used outside of classe

```
friend bool operator<(const point& p1, const point& p2) ...
```

↪ `friend` keyword only at the declaration (.hpp), NOT the definition (.cpp)

```
friend bool operator<(const point& p1, const point& p2); //.hpp
bool operator<(const point& p1, const point& p2) //.cpp
```

Note: private members are (still) directly accessible inside the class

```
void point::translate(const point& other) {
  x = x + other.x; //without other.get_x()
  y = y + other.y; //without other.get_y()
}
```

Week 3 Lab - main pitfalls/tips III

Files as input/output

Software Engineering 1: Introduction to Computing, 5a 5b Text and files

```
#include <iostream>
#include <fstream>

int main() {
  std::ifstream infile;
  infile.open("numbers.txt");

  std::ofstream outfile;
  outfile.open("numbers-squares.txt");

  int n ;
  while(infile >> n) {
    std::cout << n << std::endl;
    outfile << n << " " << n*n << std::endl;
  }

  infile.close();
  outfile.close();
}
```

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

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Week 3 Lab - main pitfalls/tips IV

main full prototype - command line arguments

```
int main(int argc, char* argv[])
```

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

//argc: argument count
//argv: argument vector
int main(int argc, char* argv[]) {
  cout << "argc = " << argc << endl;

  for(int i=0; i<argc; i++) {
    cout << "argv[" << i << "] = " << argv[i] << endl;
  }

  return 0;
}
```

```
$./prog 2 10.5 true file
argc = 5

argv[0] = ./prog
argv[1] = 2
argv[2] = 10.5
argv[3] = true
argv[4] = file.txt
```

Outline

Warm up - Pointers and references

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

int main() {
  int n = 1;

  int* ptr; //pointer to int
  ptr = &n;

  //int& r; //NOT ALLOWED
  int& r = n; //reference to int
  //r is an alias of the variable n

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

  cout << "n = " << n << endl;
  cout << "&n = " << &n << endl;
  cout << "ptr = " << ptr << endl;
  cout << "*ptr = " << *ptr << endl;
  cout << "r = " << r << endl;

  n = 5;

  //...

  return 0;
}
```

```
/* Output:
  $./prog
  n = 1
  &n = 0x22cc74
  ptr = 0x22cc74
  *ptr = 1
  r = 1

  n = 5
  &n = 0x22cc74
  ptr = 0x22cc74
  *ptr = 5
  r = 5
*/
```

Assigning through a pointer

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

int main() {
  int* a ;
  *a = 10;
  cout << *a << endl;
  return 0;
}
```

- Compiles with (often) no warnings.
- Segmentation fault.
- Where are we storing our value (10)?

Assigning through a pointer (to something)

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

int main() {
  int n;

  int* p;
  p = &n;

  *p = 10;
  cout << *p << endl;

  p = new int;
  *p = 15;
  cout << *p << endl;

  return 0;
}
```

- Using some other (automatically allocated) variable's address.
- Dynamic memory allocation (on the heap).

The predefined this pointer

```
double point::get_x() {
  return(this->x);
}

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

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

The new operator: dynamic memory allocation for objects I

- keyword new
- arrow operator -> (vs dot operator .)

```
#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;
}
```

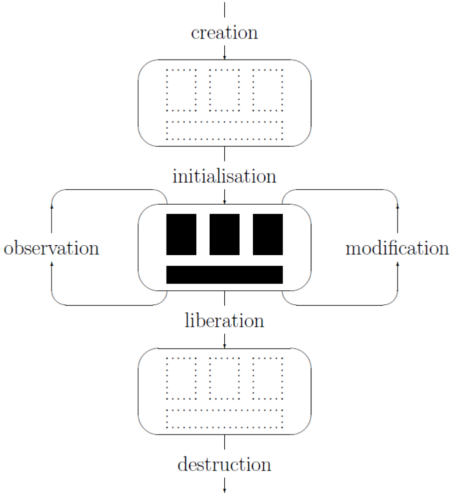
- pointer->member: p1->get_x()
- object.member: (*p1).get_x()

The new operator: dynamic memory allocation for objects II

- can also initialise non-class types

```
int* n;
n = new int(13); //Initialises *n to 13
```

Objects life cylce



The destructor I

~point();

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

The destructor II

```
int main() {
  //the following curly brace
  //is not a typo!
  {
    cout << "a new scope begins" << endl; /* Output:
    point p1;                               a new scope begins
    point* p2 = new point(1 ,2);            the new scope ends
    cout << "the new scope ends" << endl;   point 0 0 is leaving
  }                                         goodbye everyone !
                                          */
  cout << "goodbye everyone!" << endl;

  return 0;
}
```

↪ Dynamically allocated objects need to be manually de-allocated

```
delete p2; //"destroy" memory allocated for p2
delete[] points_array;
```

The destructor and the delete operator I

```
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 and the delete operator II

```
int main() {
  {
    cout << "a new scope begins" << endl;
    point p1;                                  /* Output:
    point* p2 = new point(1 ,2);                 a new scope begins
    delete p2;                                   point 1 2 is leaving
    cout << "the new scope ends" << endl;        the new scope ends
  }                                              point 0 0 is leaving
                                                 goodbye everyone !
  cout << "goodbye everyone!" << endl;         */

  return 0;
}
```

The destructor

- 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.

Returning a pointer I

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

int* square(int n) {
  int square_n = n*n;
  return &square_n;
}

int main() {
  int n = 10;
  cout << * square(n) << endl;
  return 0;
}
```

- Does it work?

Returning a pointer II

```
$ g++ retpoint.cpp -o ret
retpoint.cpp:  In function int* square(int):
retpoint.cpp:5:  warning:  address of local variable
square_n returned
$ ./ret
100
```

- Undefined behaviour.
- Very bad idea.

Returning a pointer (to something) I

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

int* square(int n) {
  int* square_n = new int;
  *square_n = n*n;
  return square_n;
}

int main() {
  int n;

  while(1) {
    cin >> n;
    cout << *square(n) << endl;
  }
  return 0;
}
```

- Dynamic memory allocation on the heap.
- Is it a good idea?

Returning a pointer (to something) II

```
int main() {
  int n;
  int *sq;

  while(1) {
    cin >> n;
    sq = square(n);
    cout << *sq << endl;

    delete sq;
  }
  return 0;
}
```

- Beware of memory leaks.

Returning a pointer (to some member variable) I

```
class point{
  public:
    . . .
    double* get_x() {
      return &x;
    }
    . . .
};

int main() {
  point p1(1 ,5);
  cout << *p1.get_x() << endl;
  return 0;
}
```

- Does it work? Is it a good idea?

Returning a pointer (to some member variable) II

```
class point{
  public:
    . . .
    double* get_x() /* look , no const here */ {
      return &x;
    }
    . . .
};

int main() {
  point p1(1 , 5);
  cout << *p1.get_x() << endl;
  double* change_x = p1.get_x();

  *change_x = 20;
  cout << *p1.get_x() << endl;

  *(p1.get_x()) = 25;
  cout << *p1.get_x() << endl;

  return 0;
}
```

```
/* Output:
  1
  20
  25
*/
```

- Breaking encapsulation!

Returning a pointer (to some member variable) III

```
class point{
  public:
    ...
    double* get_x() const {
      return &x;
    }
    ...
};

/* Output:
  $ g++ main.cpp
  main.cpp: In member function 'double* point::get_x() const':
  main.cpp:9:11: error: invalid conversion from 'const double*' to 'double*' [-fpermissi
    return &x;
*/
```

- `const`-correctness, yet again!

Returning a pointer (to const) to some member variable

```
class Point{
  public :
    ...
    const double* get_x() const {
      return &x;
    }
    ...
};
int main() {
  Point p1(1 ,5);
  cout << *p1.get_x() << endl;

//     double * change_x = p1.get_x();
//     *change_x = 20;
//     error: invalid conversion from 'const double*' to 'double *'

//     *(p1.get_x()) = 20;
//     e r r o r : assignment of read-only location
//     ' *p1.Point::get_x()'

  return 0;
}
```

‘Pointer to const’

- ‘Pointer to const’ doesn’t mean that the pointed memory area is guaranteed to be constant.
- It means we cannot change its value using that specific pointer.
- It can still be changed, e.g. by other pointers not declared as const.

Pointers to const vs. const pointers

```cpp
int main(){
    const point* p1;
    p1 = new point();

//  point* const p2;
//  error: uninitialized const 'p2'
    point* const p2 = new point();

//  p1->origin_symmetric();
//  passing 'const point' as 'this' argument of
//  'void point::translate()' discards qualifiers
    p2->origin_symmetric();

    point p3;
    p1 = &p3;
//  p2 = &p3;
//  error: assignment of read-only variable 'p2'

    delete p2;
//  delete p1;
//  not a good idea (why?)
    return 0;
}
```

Are we missing something?

```
p1 = new point();
...
point p3;
p1 = &p3;
```

- The address to the memory area allocated by new and previously pointed by p1 is irremediably lost.
- We cannot access it anymore, we can't free it (using delete) either.

  `delete p1;`
- Segmentation fault.
- p1 now points to an automatically allocated variable (p3).
- We cannot 'dynamically free' memory which wasn't dynamically allocated (and why should we?).

Memory leaks I

- **Q**: How do I deal with memory leaks?
- **A**: By writing code that doesn't have any.

Clearly, if your code has new operations, delete operations, and pointer arithmetic all over the place, you are going to mess up somewhere and get leaks, stray pointers, etc.

This is true independently of how conscientious you are with your allocations: eventually the complexity of the code will overcome the time and effort you can afford.

It follows that successful techniques rely on hiding allocation and deallocation inside more manageable types.

Good examples are the standard containers. They manage memory for their elements better than you could without disproportionate effort.
[B. Stroustrup, C++ FAQ]

Memory leaks II

- Easy to miss `undeleted` memory areas here and there.
- Avoid 'bare' `new` (although sometimes useful).
- Use containers (e.g. `vector`) which take care of memory management for you (or write your own wrapper class).
- Garbage collection (default in other languages e.g. Java).

Returning a reference to a member variable

```
class point{
    public:
    ...
    const double& get_x() const {
        return x;
    }
    ...
};

int main(){
    ...
    point* p1 = new point(1, 5);
    const double& save_x = p1->get_x();
    double copy_x = p1->get_x();

    delete p1;
    // copy_x is ok
    // save_x is a reference to...?
    ...
}
```

Conclusion: Returning pointers and references

- Returning references, like returning pointers, can break encapsulation.
- We have already seen how to avoid it: `const` references.
- In some cases exposing 'what's inside' to change or assignment is the desired behaviour (for instance when?).
- The syntax for pointers can be cumbersome, references are quite smooth.
- Accessor/getter member functions for object member data often return `const` references.

The copy constructor - default

- Created by default by the compiler (if we don't provide ours).
- Default copy constructor does a bitcopy replication of the member data.
- Ok when member data can be 'trivially' copied:

  ```
  point p1(10, 10);
  point p2(p1); //copies p1.x into p2.x and p1.y into p2.y
  ```
- What happens if a class member variable is an array? a vector? of ints? of points?

  ```
  int* t;, std::vector<point> points_vec;
  ```
- $\hookrightarrow$ need to define our own copy constructor

The copy constructor - User defined I

```
/* copy constructor */
//point.hpp
point(const point& other);


//point.cpp
point::point(const point& other) {
  point(other.x, other.y); //sure?
}
```

```
int main() {
  point p1(2, 3);
  point p2(p1); //copy constructor
  point p3(point(3, 4));

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

```
/* Output:
  ./prog
  point(double, double)
  point(double, double)
  [2, 3]
  [0, 0]
  [0, 0]
*/
```

The copy constructor - User defined II

```
/* copy constructor */
//point.hpp
point(const point& other);


//point.cpp
point::point(const point& other):x(other.x), y(other.y) {}
```

```
int main() {
  point p1(2, 3);
  point p2(p1); //copy constructor
  point p3(point(3, 4));

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

```
/* Output:
  ./prog
  [2, 3]
  [2, 3]
  [3, 4]
*/
```

Assignment operator

```
/* assignment operator, as member function */
//point.hpp

point& operator=(const point& other);

//point.cpp
point& point::operator=(const point& other) {
  x = other.x;
  y = other.y;
  distance_orig = other.distance_orig;

  return *this;
}
```

```
/* Output:
  ./prog
  [2, 3]
  [2, 3]
  [3, 4]
*/
```

```
int main() {
  point p1(2, 3);
  point p2 = p1;
  point p3 = (point(3, 4));

```

Assignment operator

```
/* assignment operator, as member function */
//point.hpp

point& operator=(const point& other);

//point.cpp
point& point::operator=(const point& other) {
  x = other.x;
  y = other.y;
  distance_orig = other.distance_orig;

  return *this;
}
```

```
int main() {
  point p1(2, 3);
  point p2 = p1;
  point p3 = (point(3, 4));

```

```
/* Output:
  ./prog
  [2, 3]
  [2, 3]
  [3, 4]
*/
```

Outline

Summary

- C++ inherits from C all the pointer power/responsibility (and clumsiness).
- References help in terms of syntax.
- Memory leaks in C++ are a serious issue (and a productivity bottleneck).
- dynamic memory allocation
  - `new`
  - `destructor`, `delete`
- copy constructor, assignment operator overloading

What to do next?

Next Lab

- Assignment operator overloading(`operator=`)
- More on dynamic allocation (`new`, `destructor`, `delete`)
- `Makefile` with **Implicit Rules** and **variables**

Next Lecture

Week5 - Classes Relationships: Association, Aggregation/Composition and Generalisation (Inheritance)