EL2310 – Scientific Programming

Lecture 15: OOP in C++

Ramviyas Parasuraman (ramviyas@kth.se)

Royal Institute of Technology – KTH

Overview

Lecture 15: OOP in C++

- Reminders
- Wrap Up
- Operator Overloading
- Inheritance
- Polymorphism and Virtual Functions

So far..

- OOP concepts in C++
- Classes: definition and declaration

Today

- Inheritance, Overloading and Polymorphism

Lecture 15: OOP in C++

Reminders

Wrap Up

Operator Overloading

Inheritance

Polymorphism and Virtual Functions

Group presentation today

- Group 10 (Helmi and Pang)
  - How to optimize C code. Explain with examples
- Group 12 (Victor, Anton.D, and Bjorn)
  - Introduce Genetic Algorithms (GA)
  - Implement a GA solution for a problem in C++, e.g., Traveling Salesman Problem

Group presentation on Wednesday (14/10)

- Group 13 (Nikhil and Sanel)
  - Huffman Coding for compression
  - Implement it in C++

Group presentation on Thursday (15/10)

- Group 14 (Roberto, Paul and Adam):
  - Expectation-Maximization (EM) algorithm
  - Monte Carlo Sampling for inference and approximation
  - Implement an example in C++
- Group 15 (Pablo and Anton.I)
  - Introduce Multi-threading
  - Show some implemented examples in C++

The C++ project

- Is announced! http://www.csc.kth.se/ yaseminb/cplusplus.html
- Deadline: Monday 26.10.2014
- Help session:
  Friday 16.10.2014, 1-3:00pm, Room 22:an, Teknikringen 14
- **Reminder: C project deadline today (extended)!**

Lecture 15: OOP in C++

- Reminders
- Wrap Up
- Operator Overloading
- Inheritance
- Polymorphism and Virtual Functions

Destructor

- To free memory (DMA) when an object is deleted
- Only 1 destructor in a class
- Syntax: `~ClassName();`

-
  ```
  Class A {
  public:
    A(); // Constructor
   ~A(); // Destructor
  ...
  };
  ```

Source and header file

- The *definition* goes into the header file .h
- The *declaration* goes into the source file .cpp
- Header file ex:

```
class A{
public:
  A();
private:
  int m_X;
};
```

- Source file ex:

```
#include "A.h"
A::A():m_X(0)
```

this pointer

- Inside class methods you can refer to the object with `this` pointer
- The `this` pointer cannot be assigned (your program decides it run-time)

const

- To make some parameters as ”read-only”
- `const` function arguments:
- Ex: `void fcn(const string &s);`
- `const function type:`
- `Ex:  void fcn(int arg) const;`

Static members

- A `static` member (data/function) is the same across all objects.
- It's a member of the *class*, not of any single object
- Ex: `int A::m_Counter = 0;` if `m_Counter` is a static data member of class `A`

Lecture 15: OOP in C++

Reminders

Wrap Up

Operator Overloading

Inheritance

Polymorphism and Virtual Functions

Operator overloading

- Operators behave just like functions
- Compare

```
Complex& add(const Complex &c);
Complex& +=(const Complex &c);
```

- You can overload (provide your own implementation of) most operators
- This way you can make them behave in a “proper” way for your class
- It will not change the behavior for other classes only the one which overloads the operator
- Some operators are member functions, some are defined outside class

Task 1

- Use the Complex number class from before. Overload/implement:
- `std::ostream& operator<<(std::ostream &os, const Complex &c);`
- `Complex operator+(const Complex &c1, const Complex &c2)`
- `Complex operator+(const Complex &c);` (member function)
- `Complex& operator=(const Complex &c);` (member function)

Function overloading

- We can create functions and methods with the same name, but different arguments
- It is not possible to overload by changing return type
- Example:

```
void method();
void method(int a);
void method(int b, double c);
void method(int b); WRONG!
int method(int b); WRONG!
```

Dynamic allocation of objects

- One reason to use dynamic memory allocation (`new/delete`):
  - Moving around pointers to BIG chunks of memory (avoiding unnecessary copying)
- Makes sense not only for arrays
- Objects can also be BIG (e.g. database object can be 500MB!)
- Typically, we dynamically allocate objects
- We free memory when the object is no longer needed
- We pass objects by reference (* or &) to functions
- Example:

```
Database db = new Database("mydatabase.db");
useDb(db); // void useDb(Database *db)
delete db;
db = NULL;
```

Lecture 15: OOP in C++

Reminders

Wrap Up

Operator Overloading

Inheritance

Polymorphism and Virtual Functions

Inheritance

- Inheritance is a way to show a relation like “is a”
- Ex: a car is a vehicle
- A car inherits many of its properties from being a vehicle
- These same properties could be inherited by a truck or a bus
- Syntax:
  ```
  class Car :  public Vehicle
  ```
  specifies that Car inherits from Vehicle

Inheritance and Constructors

- If you have three classes A, B and C,
- where
  - B inherits from A (`class B: public A`)
  - C inherits from B (`class C: public B`)
- When you create C:

  ```
  C c;
  ```

  the constructor from the base classes (B and A) will be run first
- Execution order
  1. Constructor of A
  2. Constructor of B
  3. Constructor of C

Access specifiers

- `private`: can be accessed from:
  - inside of the class
- `public`: can be accessed from:
  - inside of the class
  - subclasses
  - outside of the class
- `protected`: can be accessed from:
  - inside of the class
  - subclasses

Lecture 15: OOP in C++

Reminders

Wrap Up

Operator Overloading

Inheritance

Polymorphism and Virtual Functions

Polymorphism

- A variable/function can have more than one form
- Example of polymorphism: operator/function overloading
- We can have sub-type polymorphism:
  a variable can be of more than one form
- A variable of a base type can hold an object of a sub-type
- In C++ implemented using references or pointers to base classes

Polymorphism example

-
  ```
  class Vehicle
  {...}
  class Car:  public Vehicle
  {...}
  ```
- `Vehicle *v1 = new Vehicle();`
- `Vehicle *v2 = new Car();`
- `v2` is a Car hidden inside a variable of type pointer to Vehicle!
- We can then write: `v1 = new Car();`
- So, `v1` can hold both a Car and a Vehicle (or even a Truck!)
  Polymorphism!

Subclasses as arguments to function

- If a function requires as argument a pointer/reference to an object of class `A`
- We can provide a pointer/reference to any subclass of `A`

Accessing methods

-
  ```
  class Vehicle
  {
    void drive();
  }
  class Car:  public Vehicle
  {
    void openTrunk();
  }
  ```
- `Vehicle *v = new Car();`
- `v->drive();` runs drive() from the Vehicle part of the Car
- `v->openTrunk();` NOT POSSIBLE!
- But: `((Car *)v)->openTunk();` WORKS!

Overloading in sub-classes

- We can overload a method in a sub-class
```
class Vehicle {
  void drive();
}
class Car: public Vehicle {
  void drive();
}
```
- `Vehicle *v1 = new Vehicle();`
- `Vehicle *v2 = new Car();`
- `Car *c = new Car();`
- `v1->drive();` and `v2->drive();` run drive() from the Vehicle
- `c->drive();` runs drive() from the Car

`virtual` functions

- What if we want the object know what it “really” is and run the correct `drive()` method?
- Declare the method with the keyword `virtual`

  ```
  class Vehicle {
    virtual void drive();
  }
  class Car:  public Vehicle {
    virtual void drive();
  }
  ```
- `Vehicle *v1 = new Vehicle();`
- `Vehicle *v2 = new Car();`
- `v1->drive();` runs drive() from the Vehicle
- `v2->drive();` runs drive() from the Car

Polymorphism with `virtual` functions

- What `virtual` function to run is determined at run-time
- Depends on the “real” type of objects
- Works for both pointers and references

Interfacing: Abstract class

- In C++, abstract classes provides interfaces
- Not to be confused with data abstraction
- To make a class abstract : declare at least one of its functions as pure ”virtual” function.
- A pure virtual function is specified by placing ”= 0”
-
```
class Car
{
public:
  virtual double getNrWheels() = 0; // pure
virtual function
private:
  double NrWheels
};
```

Abstract class

- Abstract classes cannot be instantiated
- Purpose : A base classes which could be inherited in other classes
- Inherited classes have to overload each of the virtual functions in the base class
- Meaning: B (inherits the base class A) supports the interface provided by A.