C++

Static members

Introduction

- There are two types of static members:
  - Static data members
  - Static method members
- Static members do not belong to the objects but to the class:
  - Static methods do not have access to this (the object pointers)
  - Static data is shared by all instances of a class

Static methods

- Contrary to regular methods, static methods can access only static data
- Static methods do not have access to the implicit **this** pointer (that points to the current object)

this pointer

- Inside member functions, a special variable named this acts as a pointer to the current object
- This variable is passed to any member functions as an hidden argument
- For example if we consider the class Point:

```
class Point {
public:
  Point(int xloc, int yloc);
  int getX();
  int getY();
private:
  int x, y;
};
```

Example

- The code for the method getX() is:
```
int Point::getX() {
  return x;
}
```
- Which is equivalent to:
```
int Point::getX() {
  return this->x;
}
```
- In fact, when we write the following code:
```
int x = point.getX();
```
- The C++ compiler converts it to something like:
```
int x = Point::get(&point); // this is not valid C++ code
```
- getX() is interpreter by the compiler as getX(Point* this)

Static methods

- Calling a static method is done by prefixing its name by the class name and "::"
  - For a class A and a static method f(): A::f(); corresponds to a call to the static method f() of the class A
- Like any other methods, static methods can access public, protected and private data of the class

Example of usage

```
class A {
private:
  int _a;
public:
 A() : _a(0) {}
 static void f() {
   std::cout << “static method f” << std::endl;
 }
};

int main() {
  A::f(); // will print “static method f”
}
```

Static data

- Static variables (static data members) are shared among the instances of the class
- Static data members are similar to global variables but they have a better protection (provided by the class)
- A global variable is accessed by prefixing its name by the name of the class and "::"
  - Example: for a class A and a static var _a, it can be accessed by A::_a;
- Static data members have the same protection mechanism as other data members.
  - For example: A private static data member can not be accessed from outside of the class
- Static data / method members are also called **class data / method**
- Static data members can not be initialized in the class definition. They must be defined outside of the class definition (for example in the class implementation (.cpp) file)

Example

```
// In file Cars.h
class Cars {
private:
  static int num_produced;

public:
  Cars() { num_produced++; }
  static int get_num_produced() {
     return num_produced;
  }
};
```

```
// In file Cars_main.cpp
#include “Cars.h”
int Cars::num_produced = 0;

int main() {
  cout << Cars::get_num_produced() << endl;
  Cars c1;
  cout << Cars::get_num_produced() << endl;
  Cars c2;
  cout << Cars::get_num_produced() << endl;
}
```

Initialization of static data

- Static data members can not be initialized in the class definition
- There is actually an exception to this rule: const static data (i.e. constants) can be initialized in the class definition
- Example:

```
class ConstantExample{
 private:
   static const int constant = 137;
};
```

- This is valid only for integral types (i.e. static const double or static const float can not be initialized this way)

Example of usage

- Static methods can be used in the named constructor idiom
  - Constructors always have the same name of the class
  - Constructors can be differentiated by the parameter list only (as any overloaded function)
  - Sometimes we want to have different constructors with the same number of parameters and same type

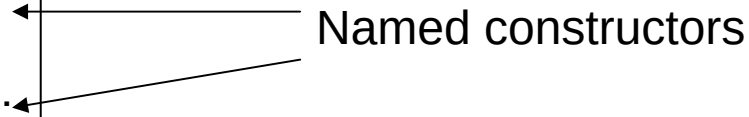
Named constructor idiom

We would like to have a class Point with two constructors:
- one to construct a point from Cartesian coord
- one to construct a point from Polar coord

```
class Point {
 public:
  Point (float x, float y); // Cartesian coord
  Point (float r, float t); // Polar coord
  // Error: ambiguous ctor
};

int main() {
 Point p(1.0, 3.0); // ambiguous: which ctor ?
}
```

Named constructor idiom

```
class Point {
public:
 // construct a point from Cart coord
 static Point createFromCart(float x, float y);
 // construct a point from Polar coord
 static Point createFromPolar (float r, float t);

private:
 Point(float x, float y); // Cart coord
 float x, y; // Cart coord
};

int main() {
  Point p1 = Point::createFromCart(1.0, 3.0);
}
```



Named constructor idiom

- These methods (createFromCart and createFromPolar) have to be static because they are used to create an object of type Point
- Since these methods are called for creation of an object, no object (i.e. no this pointer) exists at the time when the methods are called
- Note this is the same type of techniques used for the factory design pattern

Other possible usages

- Static methods can also be used to:
  - Track the number of instances of a class
  - For the factory design pattern (to create objects)
  - To share resources between objects (i.e. a Palette object shared by different Window objects in a GUI system)