COMP151

Static Methods and Data

Named Constructors

- C++ constructors have the name of the class.
- Different constructors can only be distinguished if they have different argument types.

Example: We want 2 constructors with an integer argument, interpreting it either in HHMM format or # minutes after midnight.

```
class Clock                                              // This won't work!
{
  private:
    int hour, minute;
  public:
    Clock() : hour(0), minute(0) { }
    explicit Clock(int mins) : hour(mins / 60), minute(mins % 60) { }
    explicit Clock(int hhmm) : hour(hhmm / 100), minute(hhmm % 100) { }
    void tick();
    void print();
};
```

One Solution: Global Functions

```
class Clock
{
  private:
    int hour, minute;
  public:
    Clock(int h = 0, int m = 0) : hour(h), minute(m) { }
    void tick();
    void print();
};

Clock make_clock_hhmm(int hhmm)
    { return Clock(hhmm / 100, hhmm % 100); }

Clock make_clock_minutes(int min)
    { return Clock(min / 60, min % 60); }
```

Disadvantages of Global Functions

- Global functions all live in the same namespace, so the names of the "constructor functions" have to be long.
- It is not clear that the functions belong to the class. When the class is modified, it might be easy to forget to look at the "constructor functions."
- Global functions cannot access private data members of the class. (This may be solved by friend functions.)

Static Methods

Static methods of a class are really *global functions* with a “funny name.” They belong to the class, and can access private data.

```
class Clock {
  private:
    int hour, minute;
    Clock(int h, int m) : hour(h), minute(m) { }
  public:
    Clock() : hour(0), minute(0) { }
    void tick();
    void print();

    static Clock HHMM(int hhmm) { return Clock(hhmm / 100, hhmm % 100); }
    static Clock minutes(int i) { return Clock(i / 60, i % 60); }
};

// Now we can set clocks
Clock c1;                                // 0:00
Clock c2 = Clock::HHMM(120);             // 1:20
Clock c3 = Clock::minutes(120);          // 2:00
```

Static Data

- Classes can also have static data members.
- Static data members are really global variables with a funny name and better protection.
- Static data/methods are also called class data/methods.

Compare a class `Car` with a factory:

- The `Car` objects are the products made by the factory.
- Data members are data on the products, and methods are services provided by the objects.
- Class data and class methods are data and services provided by the *factory*.
- Even if no object of this type has been created, we can access the class data and methods.

Example: car.h


```
// File "car.h"

class Car
{
  private:
    static int num_cars;
    int total_km;
  public:
    Car() : total_km(0) { ++num_cars; }
    ~Car() { --num_cars; }
    void drive(int km) { total_km += km; }
    static int cars_produced() { return num_cars; }
};
```

Example: car_main.cpp

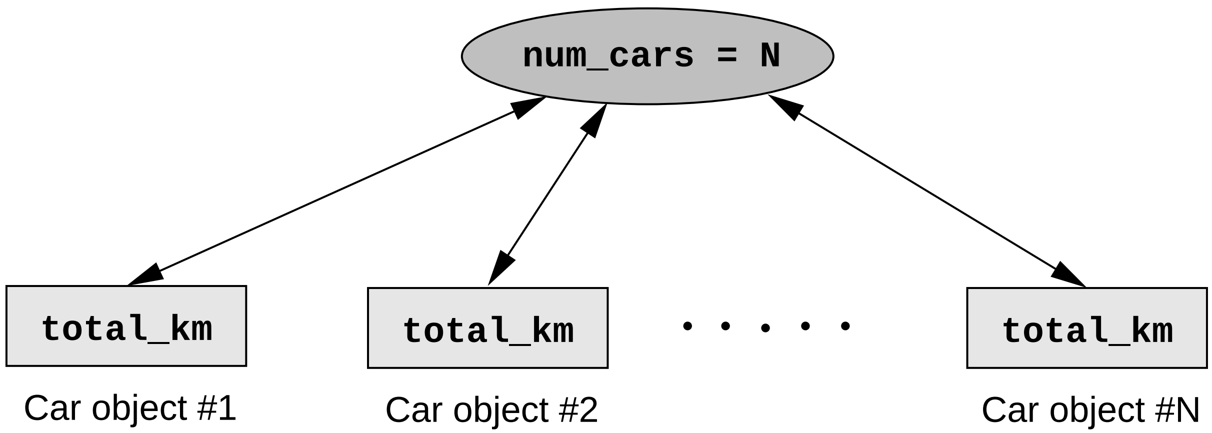
```
#include "car.h"
int Car::num_cars = 0;                    // definition of static member

int main() {
    cout << Car::cars_produced() << endl;
    Car vw; vw.drive(1000);
    Car bmw; bmw.drive(10);
    cout << Car::cars_produced() << endl;

    Car *cp = new Car[100];
    cout << Car::cars_produced() << endl;
    {
        Car kia; kia.drive(400);
        cout << Car::cars_produced() << endl;
    }
    cout << Car::cars_produced() << endl;

    delete [ ] cp;
    cout << Car::cars_produced() << endl;
    return 0;
}
```

Summary (1)



- Static variables are shared among *all objects* of the same class.
- Static variables do *not* take up space inside an object.
- Static variables, though act like global variables, *cannot* be initialized in the class definition. Instead, they must be *defined* outside the class definition.
- Usually the definitions of static variables are put in the class implementation (.cpp) file.

Summary (2)

- Static variables/methods are global variables/functions but with a class scope and are subject to the access control specified by the programmer.
- Static methods can only use static variables of the class. Reason: static methods do not have the implicit *this* pointer like regular member functions.

e.g. a regular member function of Car like

```
    void drive(int km) { total_km += km; }
```

after compilation becomes:

```
void Car::drive(Car* this, int km)
          { this->total_km+=km; }
```

Summary (3)

On the other hand, a static method of Car like

```
static int cars_produced() { return num_cars; }
```

after compilation becomes:

```
int Car::cars_produced() { return num_cars; }
```

Example: student_non_static.h

Without static members:

```
// File: "student_non_static.h"

class Student
{
  private:
    string name;
    vector<string> memory;
  public:
    Student(string s) : name(s) { }
    void memorize(string txt) { memory.push_back(txt); }
    void do_exam();
};
```

Example: student_non_static.cpp


```
#include "student_non_static.h"

void Student::do_exam()
{
    if(memory.empty())
        cout << name << ":   " << "Huh???" << endl;
    else
    {
        vector<string>::const_iterator p;
        for (p = memory.begin(); p != memory.end(); ++p)
            cout << name << ":   " << *p << endl;
    }

    cout << endl;
}
```

Example: exam.cpp


```
#include "student_non_static.h"
int main()
{
    Student Jim("Jim");
    Jim.memorize("Data consistency is important");
    Jim.memorize("Copy constructor != operator=");

    Student Steve("Steve");
    Steve.memorize("Overloading is convenient");
    Steve.memorize("Make data members private");
    Steve.memorize("Default constructors have no arguments");

    Student Mary("Mary");

    Jim.do_exam();
    Steve.do_exam();
    Mary.do_exam();
}
```

Example: exam.cpp Output

```
Jim: Data consistency is important
Jim: Copy constructor != operator=

Steve: Overloading is convenient
Steve: Make data members private
Steve: Default constructors have no arguments

Mary: Huh???
```

Example: student_static.h

With static members:

```
// File: "student_static.h"

class Student
{
  private:
    string name;
    static vector<string> memory;
  public:
    Student(string s) : name(s) {}
    void memorize(string txt) { memory.push_back(txt); }
    void do_exam();
};
```

Example: student_static.cpp


```
// File: "student_static.cc"
#include"student_static.h"

vector<string> Student::memory;

void Student::do_exam()
{
    if (memory.empty())
        cout << name << ":   "<< "Huh???" << endl;
    else
    {
        vector<string>::const_iterator p;
        for (p = memory.begin(); p != memory.end(); ++p)
            cout << name << ":   " << *p << endl;
    }
    cout << endl;
}
```

Example: Collective Memory

In this version of the Student class, all students share their memory. So even though Mary didn not memorize anything, she can access all the knowledge memorized by Jim and Steve.

```
Jim: Data consistency is important
Jim: Copy constructor != operator=
Jim: Overloading is convenient
Jim: Make data members private
Jim: Default constructors have no arguments

Steve: Data consistency is important
Steve: Copy constructor != operator=
Steve: Overloading is convenient
Steve: Make data members private
Steve: Default constructors have no arguments

Mary: Data consistency is important
Mary: Copy constructor != operator=
Mary: Overloading is convenient
Mary: Make data members private
Mary: Default constructors have no arguments
```

Example: Linked List

Here is an example of a `Person` class that automatically links together all persons in a linked list.

```
class Person
{
  private:
    static Person* first;
    string name;
    Person* next;
  public:
    Person(string s) : name(s), next(first) { first = this; }
    Person(const Person &p) : name(p.name), next(first) { first = this; }
    ~Person()
    {
        if (first == this) { first = next; return; }
        for (Person* p = first; p; p = p->next)
            if (p->next == this) { p->next = next; return; }
        abort("Destruct PANIC!");
    }
    Person& operator=(const Person& p) { name = p.name; }
};
```