Object Oriented Software design

Const, inlines, static members, composition

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

December 6, 2010

Outline

Outline

Exercise 1

- Is the following code correct? Do you see any errors?

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

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) { size = sz; }
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
}
```

Exercise 1

- Is the following code correct? Do you see any errors?
- Modify the code so that it assigns a random number to size

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

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) { size = sz; }
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
}
```

Exercise 2

- The following example does not compile. Can you tell why?

```
class X {
  int i;
public:
  X(int ii);
  int f();
};

X::X(int ii) : i(ii) {}
int X::f() { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

Exercise 2

- The following example does not compile. Can you tell why?
- Modify it so that it will compile correctly

```
class X {
  int i;
public:
  X(int ii);
  int f();
};

X::X(int ii) : i(ii) {}
int X::f() { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

Outline

Mutable members

- Sometimes you need to create an object that is “logically” constant, but in which some of its members can actually be changed.
- Suppose for example that you want to count how many times a certain member function is called, and suppose that the member function is const (a weird example)

```
class Y {
  int i; // only for counting
  int a; // real data
public:
  Y();
  int get() const;
  int inc();
};

Y::Y() : i(0), a(0) {}
int Y::get() const {
   i++;       // Compilation error
   return a;
}
```

Casting away const

- One solution is to cast away constness

```
int Y::get() const {
  ((Y*)this)->i++; // OK: cast away const-ness
  return a;
}
```

- The above code:
  - First it converts `this` into a point of type `Y*` (rather than `const Y*`)
  - Then, it increments its member `i`
- It works but it is not clean: it is not evident in the interface that i is going to be changed

mutable

- A better approach is to use the `mutable` keyword

```
class Y {
  mutable int i; // only for counting
  int a;         // real data
public:
  Y();
  int get() const;
  int inc();
};

Y::Y() : i(0), a(0) {}
int Y::get() const {
   i++;       // no error (it is mutable)
   return a;
}
```

Volatile

- `volatile` in front of a variable means: *this data may change outside the knowledge of the compiler.*
- This may happen through multi-threading or interrupts, or any other indirect method
- Therefore, the compiler is not allowed to make assumptions about the data, especially when optimising
  - If the compiler says, “I read this data into a register earlier, and I haven’t touched that register,” normally it wouldn’t need to read the data again.
  - But if the data is volatile, the compiler cannot make such an assumption because the data may have been changed by another process, and it must reread that data rather than optimising the code to remove what would normally be a redundant read.
- The syntax of volatile is the same as the syntax for const

Example

▸ `./examples/14.cpp-examples/volatile.cpp`

- As with const, you can use volatile for data members, member functions, and objects themselves. You can only call volatile member functions for volatile objects.
- The reason that `isr()` cannot actually be used as an interrupt service routine is that in a member function, the address of the current object (`this`) must be secretly passed, and an ISR generally wants no arguments at all.
- To solve this problem, you can make `isr()` a static member function (will see later)

Outline

On the use of macros

- Let's remind ourselves what is a macro in C/C++
  - It is a pre-processor directive, used to substitute code inside the main body of the program

```
#define PI 3.14
#define SUM(x,y) (x+y)
---

int a, b, c;
a = SUM(b,c);     // translated into a = (b+c);
a = PI*SUM(b+c); // translated into a = 3.14*(b+c);
                  // etc.
```

  - Therefore, it is possible to write simple "functions" without the overhead of the function call

Exercise

- Things can get nasty when using macros
- What does the following macro do?

```
#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)
...
int a = BAND(4);
int b = BAND(7);
int c = 10;
a = BAND(c);
```

- List the value of a, b and c after each instruction

Exercise - cont.

- Anticipate the output of the following code:

```
int main() {
  for(int i = 4; i < 11; i++) {
    int a = i;
    cout << "a = " << a << endl << '\t';
    cout << "BAND(++a)=" << BAND(++a) << endl;
    cout << "\t a = " << a << endl;
  }
}
```

- Check it: ▸ ./examples/14.cpp-examples/macro-output.txt

What happened?

- The problem is that `BAND(++a)` is substituted by:

```
(((++a)>5 && (++a)<10) ? (++a) : 0)
```

What happened?

- The problem is that `BAND(++a)` is substituted by:

```
(((++a)>5 && (++a)<10) ? (++a) : 0)
```

- If ++a is less than or equal to 5, the shortcut does not evaluate the other two terms, and the output is 0 , while a is only incremented once (correctly)

What happened?

- The problem is that `BAND(++a)` is substituted by:

```
(((++a)>5 && (++a)<10) ? (++a) : 0)
```

- If ++a is less than or equal to 5, the shortcut does not evaluate the other two terms, and the output is 0 , while a is only incremented once (correctly)
- However, if ++a is between 6 and 9, the increment is performed 3 times! Not what we expected!

What happened?

- The problem is that `BAND(++a)` is substituted by:

```
(((++a)>5 && (++a)<10) ? (++a) : 0)
```

- If ++a is less than or equal to 5, the shortcut does not evaluate the other two terms, and the output is 0 , while a is only incremented once (correctly)
- However, if ++a is between 6 and 9, the increment is performed 3 times! Not what we expected!
- The only way to avoid this problem is to transform band into a function

What happened?

- The problem is that `BAND(++a)` is substituted by:

```
(((++a)>5 && (++a)<10) ? (++a) : 0)
```

- If ++a is less than or equal to 5, the shortcut does not evaluate the other two terms, and the output is 0 , while a is only incremented once (correctly)
- However, if ++a is between 6 and 9, the increment is performed 3 times! Not what we expected!
- The only way to avoid this problem is to transform band into a function
- But, what is we want the clean and type-safe behaviour of a function, without the extra overhead?

Macros and private members

- Another problem with macros in C++ is that they cannot change the access rules of C++

```
class A {
  int i;
public:
  A();
}
#define GET() A::i
```

- It only works inside function members of `A`, because `i` is a private member
- To solve this issue C++ designers introduced the inline concept

inline functions

- When a function is declared as `inline`, the compiler tries to substitute its code in place of the call
- The main difference is that *the compiler* will do that, not the *pre-processor*, and this changes everything

```
inline band(int x) {return (x>5 && x<10) ? x : 0;}

int a=5;
cout << band(++a) << endl;
```

- The last instruction will produce a result of 6 (correctly), without producing a function call

Inline members

- Members functions that are defined inside the class declaration are automatically inline

```
class A {
  int i;
public:
  A();
  int get() { return i; }    // inline automatically
  void set(int x);
}

void A::set(int y) { i = y; } // not inline
```

- Method `get()` is also called *in-situ*
- Method `set()` can be made inline by specifying the inline keyword and by including it in the .h file

inline members

- Two different styles of programming:

In situ:

```
class A {
  int i;
public:
  int get() { return i; }
  void set(int x) { i=x; }
};
```

Regular inlines:

```
class A {
  int i;
public:
  int get();
  void set(int x);
};

inline int A::get()
{
  return i;
}
inline void A::set(int x)
{
  i=x;
}
```

- The second one is slightly more readable, as it does not *clog* the interface with code

When to inline

- First of all, the compiler is not forced to inline
  - If the function is too long, almost certainly it will not be inlined
- Consider that the whole point of using inlines is to be more efficient:
  - if the function is too long, and it is called in many places, it will bloat the code, reducing the efficiency
  - if the function is too long, the performance gain from avoiding a function call is minimal (or even negative)
- Remember that inline functions must be in the include file, for the compiler to take advantage of it
  - Sometimes, it may not be the case to put functions in the header file
  - if you change them, the user of your library will need to recompile his/her code

Accessor functions

- inlines are convenient for simple *accessor methods*
  - Accessor methods are methods to *set/get* the values of internal members

```
class A {
  int i;
public:
  void set(int x) { i=x; }
  int  get() { return i; }
};
```

- One may think: “what’s the point in writing accessor methods? why not making the member variable just public?”
  - If you make `i` public, it becomes part of the interface
  - it may be difficult to control who and when the variable changes it value
  - it may be impossible to restrict the set of values
  - it may be impossible to change its name without rewriting a lot of user code
- A public member variable introduces an unnecessary dependence between your code and the user code

◂ ret

Exercise

- Modify the previous code so that:
  - Write a main function (*user-code*) that calls the `get()` and `set()` methods
  - A debug string is printed every time the user calls `set()`
  - The value in `set()` is restricted to non-negative values
  - Count how many times the `get()` method is called
- How many lines of “user code” you needed to change?
  - Evaluate what would have been needed if `i` was a public variable without accessor methods

The pre-processor fights back

- Of course, there are things that can only be done by using the pre-processor
- One such thing is effective debugging
  - We want to be able to debug our code by printing traces of execution, values of variables, etc.
  - When we are sure that everything is ok, we would like to remove the debugging code, because it brings unnecessary overhead
  - However, often we are wrong: we were sure everything was ok, we removed the debugging code, and then we realised we needed some more debugging
  - It would be nice if the debugging code would disappear at once, to appear again only when needed

More pre-processor features

- One very useful feature is conditional compilation:

```
#define DEBUG 1

#ifdef DEBUG
cout << "This is a debugging string" << endl;
#endif
cout << "This is a normal string" << endl;
```

  - the first `cout` instruction is only compiled if the `DEBUG` symbol is defined (first line)
  - Therefore, to not compile it, we only need to comment the first line

- A symbol can also be defined on the compiler command line with the -D option:

```
g++ -DDEBUG -c myfile.cpp
```

A debug macro

- Another interesting feature is the *stringizing*: by putting a # symbol in front of an identifier, the identifier is transformed into a string

```
#define DBGPRINT(x) cout << #x " = " << x << endl

int a = 0;
DBGPRINT(a);
// translated into
// cout << "a" " = " << a << endl;
```

- This can be very useful for debugging.
- We can also trace instructions:

```
#define TRACE(s) cerr << #s << endl, s

TRACE(f(i)); // prints "f(i)" and executes it
```

Token pasting

- We can create identifiers by concatenating them into a new identifier

```
#define FIELD(a) string a##_string; int a##_size
class Record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

- Guess what is the output?

Combining debug and conditional compilation

```
#ifdef DEBUG
#define DBGPRINT(x) cout << #x " = " << x << endl
#define TRACE(s) cerr << #s << endl, s
#else
#define DBGPRINT(x)
#define TRACE(s)
#endif
```

- As an exercise, use the above definitions to add debugging strings in ▸ this code
- Then try to compile with debug and without debugging information

Outline

Composite objects

- Composition is an essential technique of object oriented programming
- A C++ class can have member variables that are
  - objects of other classes,
  - pointers to objects of other classes
  - references to objects of other classes
- All the above cases are actually different, so let's analyse them carefully, one by one

Can you tell the difference with Java?

Example: car and wheels

car/wheel.hpp

```
#include <string>

class Wheel {
    std::string name;
public:
    Wheel(const std::string &n);
    Wheel(const Wheel &w);
    std::string getName() const;
};

inline std::string Wheel::getName() const
{
    return name;
}
```

Wheels

car/wheel.cpp

```
#include <iostream>
#include "wheel.hpp"
#include "debug.hpp"

using namespace std;

Wheel::Wheel(const string &n) : name(n)
{
    DBGPRINT("Wheel constructor");
    DBGVAR(name);
}

Wheel::Wheel(const Wheel &w) : name(w.name)
{
    DBGPRINT("Wheel copy constructor");
}
```

Car

car/car.hpp

```
#include "wheel.hpp"

class Car {
    std::string name;
    Wheel leftFrontWheel;
    Wheel rightFrontWheel;
    Wheel leftRearWheel;
    Wheel rightRearWheel;
public:
    Car(const std::string &n);
    Car(const Car &c);
    std::string getName() const;
};

inline std::string Car::getName() const
{
    return name;
}
```

Car

car/car.cpp

```
#include <iostream>
#include "car.hpp"
#include "debug.hpp"

using namespace std;

Car::Car(const string &n)
    : name(n),
      leftFrontWheel("LF"),
      rightFrontWheel("RF"),
      leftRearWheel("LR"),
      rightRearWheel("RR")
{
    DBGPRINT("Car constructor");
    DBGVAR(name);
    DBGPRINT("--------------------");
}

Car::Car(const Car &c)
    : name(c.name),
      leftFrontWheel(c.leftFrontWheel),
      rightFrontWheel(c.rightFrontWheel),
      leftRearWheel(c.leftRearWheel),
      rightRearWheel(c.rightRearWheel)
{
    DBGPRINT("Car copy constructor");
    DBGVAR(name);
    DBGPRINT("--------------------");
}
```

The client

car/carmain.cpp

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

using namespace std;

Car f(Car xx, Car yy)
{
    cout << xx.getName() << endl;
    cout << yy.getName() << endl;
    return xx;
}

const Car &g(const Car &xx)
{
    cout << xx.getName() << endl;
    return xx;
}

int main()
{
    Car c1("Fiat Punto");
    Car c2("Audi A4");
    Car c3("Citroen C3");

    Car c4 = f(c1, c2);
    {
        Car c5 = c3;
        Car c6 = g(c4);
    }
}
```

Exercise

- Defining four variables seems redundant code
  - Try to substitute the 4 variables with an array
  - How to initialise the array?
  - See solution in
    - `./examples/14.cpp-examples/car2/wheel.hpp`,
    - `./examples/14.cpp-examples/car2/wheel.cpp`,
    - `./examples/14.cpp-examples/car2/car.hpp` and
    - `./examples/14.cpp-examples/car2/car.cpp`

Static members

- In the solution to the previous example we used static members
  - A static data member has only one copy for all objects of the class
  - A static data member needs to be initialised
    - This is done in the .cpp file

```
// wheel.hpp
class Wheel {
    static int count;
    std::string name;
    int serial;
public:
    static const std::string positions[];
    ...
};
```

```
// wheel.cpp
int Wheel::count = 0;
const string Wheel::positions[] = {"LF", "RF", "LR", "RR" };
```

Static data members

- In the previous example, variable `count` is used to count the number of `Wheels` that have been created until now
  - It is initialised anytime a new `Wheel` is created, either by the default class constructor, or by the copy constructor
  - Its value is assigned to `serial`, that acts like a serial number
  - Therefore, that can be minimum `MAX_INT` wheels with a different serial number
- Exercise:
  - Introduce a serial number for cars
  - Write `get` methods for getting the serial number for wheels and cars
- `positions` is a static array of constant strings
  - it's initialised once and never changed
  - similar to `enum`, but for objects

Static methods

- A static method is a method of the class that can only access static data members
  - For example, a method to read the actual value of the variable `count` in `Wheel`

```
class Wheel {
  static int count;
  ...
public:
  ...
  static int getCount();
};

int Wheel::getCount()
{
  return count;
}
```

- How to call the function:

```
int howmany = Wheel::getCount();
```

- Notice that we do not need an object of type `Wheel` to call the method
- Static methods are not called *on objects*

String formatting

- Sometimes it is useful to create strings containing different kind of data
  - unlike in Java, the `operator+()` for strings does not convert arbitrary objects (`std::string` **is not** embedded in the C++ language)
- To do this in C++, we need to use another class of the standard template library

```
#include <sstream>
...
inline std::string Wheel::getName() const
{
    std::stringstream ss;
    ss << name << "-" << serial;
    return ss.str();
}
```

- `stringstream` works like an ordinary output stream (e.g. `cout`)
- However, instead of writing to the terminal, the output is sent into a string buffer
- Later, the string buffer can be retrieved with method `str()`

Outline

Backlink

- Now, we want to associate every wheel with the car on which it is mounted
- There are two ways of doing this:
  - By using a pointer to a `Car`
  - By using a reference to a `Car`
- The pointer is used when the association is dynamic (i.e. it may change over time)
  - For example, the wheel is mounted on a different car
- The reference is used when the association is permanent (i.e. it never change during the lifetime of the object)
- Let's use the reference first

Double include

- Now we have a problem
  - A car includes a wheel: this means that the file `car.hpp` needs to include the file `wheel.hpp` before the definition of the class `Car`
  - A wheel references a car: this means that the file `wheel.hpp` needs to include `car.hpp` before defining class `Wheel`
  - If we do both things, the pre-processor will go into an infinite loop!
- To avoid this circular reference we can use the forward declaration
- We also need *include guards*

Forward declaration

- In file `wheel.hpp` we declare that there is a class `Car` that will be fully declared later

```
// wheel.hpp
class Car;

class Wheel {
    Car &car;
    static int count;
    std::string name;
    int serial;
public:
    ...
};
```

- It works because in the file ▸ `./examples/14.cpp-examples/car3/wheel.hpp` we do not use anything of class `Car`, except a reference to it

Include guards

- However, we have another problem: `wheel.cpp` need to include both `wheel.hpp` and `car.hpp`
- `car.hpp` includes `wheel.hpp`
- Therefore, `wheel.hpp` is include two times in `wheel.cpp`
- In C++, it is not possible to declare a class two times, even if the two declarations are the same
- To avoid this, we use a pre-processor trick:

```
// wheel.hpp
#ifndef __WHEEL_HPP__
#define __WHEEL_HPP__
...
class Car;
class Wheel {...};
...
#endif
```

- The first time symbol `__WHEEL_HPP__` is not defined, so the file is expanded correctly in the first inclusion
- For the second inclusion, the symbol has already been defined, so the second time the file is not expanded

The complete Wheel implementation

car3/wheel.hpp

```
#ifndef __WHEEL_HPP__
#define __WHEEL_HPP__

#include <string>
#include <sstream>

class Car;

class Wheel {
    Car &car;
    static int count;
    std::string name;
    int serial;
public:
    static const std::string positions[];
    Wheel(Car &c, const std::string &n);
    Wheel(const Wheel &w);
    std::string getName() const;
};

inline std::string Wheel::getName() const
{
    std::stringstream ss;
    ss << name << "-" << serial;
    return ss.str();
}

#endif
```

The complete Wheel implementation

car3/wheel.cpp

```
#include <iostream>
#include "wheel.hpp"
#include "car.hpp"
#include "debug.hpp"

using namespace std;

int Wheel::count = 0;
const string Wheel::positions[] = {"LF", "RF", "LR", "RR" };

Wheel::Wheel(Car &c, const string &n) : car(c), name(n), serial(count++)
{
    DBGPRINT("Wheel constructor");
}

Wheel::Wheel(const Wheel &w) : car(w.car), name(w.name), serial(count++)
{
    DBGPRINT("Wheel copy constructor");
    DBGPRINT(" -- " << getName());
}
```

vector

- Instead of using an array for storing the wheels for a car, this time we will use a new object from the standard template library: `vector`

```
#include <vector>
...
class Car {
    std::vector<Wheel> wheels;
    ...
}
```

- `vector` is a generic container which extends the array in several ways
- it has dynamic size
  - new objects can be inserted at any time using methods `push_back()` or `insert()`
  - objects can be removed by using `pop_back()` or `erase()`
  - objects can be accessed using the standard array operator `[]`, or through other methods
- `vector` is a *template*
  - similar to generics in Java

Car

Let's see how vector is used in `Car` car3/car.hpp

```
#ifndef __CAR_HPP__
#define __CAR_HPP__

#include <vector>
#include "wheel.hpp"

class Car {
    std::string name;
    std::vector<Wheel> wheels;
public:
    Car(const std::string &n);
    Car(const Car &c);
    std::string getName() const;
};


#endif
```

Car

car3/car.cpp

```
#include <iostream>
#include "car.hpp"
#include "debug.hpp"

using namespace std;

Car::Car(const string &n)
    : name(n),
      wheels{Wheel(*this, Wheel::positions[0]),
        Wheel(*this, Wheel::positions[1]),
        Wheel(*this, Wheel::positions[2]),
        Wheel(*this, Wheel::positions[3])}
{
    DBGPRINT("Car constructor");
    DBGPRINT(getName());
}

Car::Car(const Car &c)
    : name(c.name), wheels(c.wheels)
{
    DBGPRINT("Car copy constructor");
    DBGPRINT(getName());
}

string Car::getName() const
{
    std::stringstream ss;
    ss << name;
    for (int i=0; i<4; i++)
        ss << "/" << wheels[i].getName();
    return ss.str();
}
```

Initialisation

- In the initialisation of Car we see a strange construct

```
Car::Car(const string &n)
    : name(n),
      wheels{Wheel(*this, Wheel::positions[0]),
        Wheel(*this, Wheel::positions[1]),
        Wheel(*this, Wheel::positions[2]),
        Wheel(*this, Wheel::positions[3])}
{...}
```

- The vector is initialised by using curly braces `{}`, and by listing all objects that are contained in it
- This is actually part of the future C++ standard (c++0x), which will be release some time in the next years
- To use it in the current g++ compiler, use the flag `-std=c++0x`
- More about `vector` later

Link with pointer

- As an exercise, let's modify classes `Wheel` and `Car`

1. Write a new class `Tire` that is dynamically associated with the wheel
   - Use the counter/serial number technique to number the tires
   - The tire should have a pointer to the wheel it is mounted on
   - When you create the wheel you also create the tire
   - When you destroy the wheel you also destroy the tire
2. Write a method of class `Wheel` called `Tire change(int pos, Tire *t);` that takes a pointer to a new tire, and returns the old one
3. Write a method `void run()` of a car that first checks if all the tires are correctly mounted on wheels
4. Write a `main` that first creates two cars, then swaps all their tires