Object Oriented Software Design II

C++ intro

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

Scuola Superiore Sant'Anna – Pisa

February 26, 2012

Outline

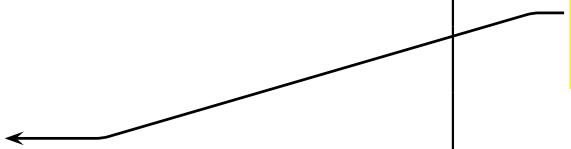
Outline

Pointers

- We can define a pointer to an object

```
class A { ... };

A myobj;
A *p = &myobj;
```



Pointer `p` contains the address of `myobj`

Pointers

- We can define a pointer to an object

```
class A { ... };

A myobj;
A *p = &myobj;
```

Pointer p contains the address of myobj

- As in C, in C++ pointers can be used to pass arguments to functions

```
void fun(int a, int *p)
{
  a = 5;
  *p = 7;
}
...
int x = 0, y = 0;
fun(x, &y);
```

x is passed by value (i.e. it is copied into a), so this assignment only modifies the copy

Pointers

- We can define a pointer to an object

```
class A { ... };

A myobj;
A *p = &myobj;
```

Pointer p contains the address of myobj

- As in C, in C++ pointers can be used to pass arguments to functions

```
void fun(int a, int *p)
{
  a = 5;
  *p = 7;
}
...
int x = 0, y = 0;
fun(x, &y);
```

x is passed by value (i.e. it is copied into a), so this assignment only modifies the copy

y is passed by address (i.e. we pass its address, so that it can be modified inside the function)

Pointers

- We can define a pointer to an object

```
class A { ... };

A myobj;
A *p = &myobj;
```

Pointer p contains the address of myobj

- As in C, in C++ pointers can be used to pass arguments to functions

```
void fun(int a, int *p)
{
  a = 5;
  *p = 7;
}
...
int x = 0, y = 0;
fun(x, &y);
```

x is passed by value (i.e. it is copied into a), so this assignment only modifies the copy

y is passed by address (i.e. we pass its address, so that it can be modified inside the function)

After the function call, x=0, y=7

Another example

pointerarg.cpp

```
class MyClass {
    int a;
public:
    MyClass(int i) { a = i; }
    void fun(int y) { a = y; }
    int get() { return a; }
};

void g(MyClass c) {
    c.fun(5);
}

void h(MyClass *p) {
    p->fun(5);
}

int main() {
    MyClass obj(0);

    cout << "Before calling g: obj.get() = " << obj.get() << endl;
    g(obj);
    cout << "After calling g: obj.get() = " << obj.get() << endl;
    h(&obj);
    cout << "After calling h: obj.get() = " << obj.get() << endl;
}
```

What happened

- Function `g()` takes an object, and makes a copy
  - `c` is a copy of `obj`
  - `g()` has no side effects, as it works on the copy
- Function `h()` takes a pointer to the object
  - it works on the original object `obj`, changing its internal value

More on pointers

- It is also possible to define pointers to functions:
  - The portion of memory where the code of a function resides has an address; we can define a pointer to this address

```
void (*funcPtr)();          // pointer to void f();
int (*anotherPtr)(int)      // pointer to int f(int a);

void f(){...}

funcPtr = &f();     // now funcPtr points to f()
funcPtr = f;        // equivalent syntax

(*funcPtr)();       // call the function
```

Pointers to functions – II

- To simplify notation, it is possible to use typedef:

```
typedef void (*MYFUNC)();
typedef void* (*PTHREADFUN)(void *);

void f() { ... }
void *mythread(void *) { ... }

MYFUNC funcPtr = f;
PTHREADFUN pt = mythread;
```

- It is also possible to define arrays of function pointers:

```
void f1(int a) {}
void f2(int a) {}
void f3(int a) {}
...
void (*funcTable []) (int) = {f1, f2, f3};
...
for (int i =0; i<3; ++i) (*funcTable[i])(i + 5);
```

Field dereferencing

- When we have to use a member inside a function through a pointer

```
class Data {
public:
  int x;
  int y;
};

Data aa;           // object
Data *pa = &aa;    // pointer to object
pa->x;             // select a field
(*pa).y;           //   "       "
```

Outline

References

- In C++ it is possible to define a reference to a variable or to an object

```
int x;              // variable
int &rx = x;        // reference to variable

MyClass obj;        // object
MyClass &r = obj; // reference to object
```

- `r` is a reference to object `obj`
  - WARNING!
  - C++ uses the same symbol & for two different meanings!
  - Remember:
    - when used in a declaration/definition, it is a reference
    - when used in an instruction, it indicates the address of a variable in memory

References vs pointers

- There is quite a difference between references and pointers

```
MyClass obj;       // the object
MyClass &r = obj; // a reference
MyClass *p;        // a pointer
p = &obj;          // p takes the address of obj

obj.fun();     // call method fun()
r.fun();       // call the same method by reference
p->fun();      // call the same method by pointer

MyClass obj2;  // another object
p = & obj2;    // p now points to obj2
r = obj2;      // compilation error! Cannot change a reference!
MyClass &r2;   // compilation error! Reference must be initialized
```

- Once you define a reference to an object, the same reference cannot refer to another object later!

Reference vs pointer

- In C++, a reference is an *alternative name* for an object

  Pointers

  - Pointers are like other variables
  - Can have a pointer to `void`
  - Can be assigned arbitrary values
  - It is possible to do arithmetic
- What are references good for?

Reference vs pointer

- In C++, a reference is an *alternative name* for an object

Pointers

- Pointers are like other variables
- Can have a pointer to `void`
- Can be assigned arbitrary values
- It is possible to do arithmetic

References

- Must be initialised
- Cannot have references to void
- Cannot be assigned
- Cannot do arithmetic

- What are references good for?

Reference example

referencearg.cpp

```
class MyClass {
    int a;
public:
    MyClass(int i) { a = i; }
    void fun(int y) { a = y; }
    int get() { return a; }
};

void g(MyClass c) {
    c.fun(5);
}

void h(MyClass &c) {
    c.fun(5);
}

int main() {
    MyClass obj(0);

    cout << "Before calling g: obj.get() = " << obj.get() << endl;
    g(obj);
    cout << "After calling g: obj.get() = " << obj.get() << endl;
    h(obj);
    cout << "After calling h: obj.get() = " << obj.get() << endl;
}
```

Differences

- Notice the differences:
  - Method declaration: `void h(MyClass &c);` instead of `void h(MyClass *p);`
  - Method call: `h(obj);` instead of `h(&obj);`
  - In the first case, we are passing a reference to an object
  - In the second case, the address of an object
- References are much less powerful than pointers
- However, they are **much safer** than pointers
  - The programmer cannot accidentally misuse references, whereas it is easy to misuse pointers

Outline

Copying objects

- In the previous example, function `g()` is taking a object by value

```
void g(MyClass c) {...}
...
g(obj);
```

- The original object is copied into parameter c
- The copy is done by invoking the *copy constructor*

```
MyClass(const MyClass &r);
```

- If the user does not define it, the compiler will define a default one for us automatically
  - The default copy constructor just performs a bitwise copy of all members
  - Remember: this is not a deep copy!

Example

- Let's add a copy constructor to `MyClass`, to see when it is called

`./examples/02.basics-examples/copy1.cpp`

- Now look at the output
  - The copy constructor is automatically called when we call `g()`
  - It is not called when we call `h()`

Usage

- The copy constructor is called every time we initialise a new object to be equal to an existing object

```
MyClass ob1(2);    // call constructor
MyClass ob2(ob1);  // call copy constructor
MyClass ob3 = ob2; // call copy constructor
```

- We can prevent a copy by making the copy constructor private:

```
// can't be copied!
class MyClass {
    MyClass(const MyClass &r);
public:
    ...
};
```

Const references

- Let's analyse the argument of the copy constructor

```
MyClass(const MyClass &r);
```

- The const means:
  - This function accepts a reference
  - however, the object will not be modified: it is *constant*
  - the compiler checks that the object is not modified by checking the *constness* of the methods
  - As a matter of fact, the copy constructor does not modify the original object: it only reads its internal values in order to copy them into the new object
  - If the programmer by mistake tries to modify a field of the original object, the compiler will give an error

Outline

Meaning of static

- In C/C++ static has several meanings
  - for global variables, it means that the variable is not exported in the global symbol table to the linker, and cannot be used in other compilation units
  - for local variables, it means that the variable is not allocated on the stack: therefore, its value is maintained through different function instances
  - for class data members, it means that there is only one instance of the member across all objects
  - a static function member can only act on static data members of the class

Static members

- We would like to implement a counter that keeps track of the number of objects that are around
  - we can use a static variable

```
class ManyObj {
    static int count;
    int index;
public:
    ManyObj();
    ~ManyObj();

    int getIndex();
    static int howMany();
};
```

```
int ManyObj::count = 0;

ManyObj::ManyObj() {
    index = count++;
}
ManyObj::~ManyObj() {
    count--;
}
int ManyObj::getIndex() {
    return index;
}
int ManyObj::howMany() {
    return count;
}
```

Static members

```
int main()
{
    ManyObj a, b, c, d;
    ManyObj *p = new ManyObj;
    ManyObj *p2 = 0;
    cout << "Index of p: " << p->getIndex() << "\n";
    {
        ManyObj a, b, c, d;
        p2 = new ManyObj;
        cout << "Number of objs: " << ManyObj::howMany() << "\n";
    }
    cout << "Number of objs: " << ManyObj::howMany() << "\n";
    delete p2; delete p;
    cout << "Number of objs: " << ManyObj::howMany() << "\n";
}
```

```
Index of p: 4
Number of objs: 10
Number of objs: 6
Number of objs: 4
```

Static members

- There is only one copy of the static variable for all the objects
- All the objects refer to this variable
- How to initialize a static member?
  - cannot be initialized in the class declaration
  - the compiler does not allocate space for the static member until it is initiliazed
  - So, the programmer of the class must define and initialize the static variable

Static data members

- Static data members need to be initialized when the program starts, before the main is invoked
  - they can be seen as global initialized variables (and this is how they are implemented)
- This is an example

```
// include file A.hpp
class A {
  static int i;
public:
  A();
  int get();
};
```

```
// src file A.cpp
#include "A.hpp"

int A::i = 0;

A::A() {...}
int A::get() {...}
```

Initialization

- It is usually done in the .cpp file where the class is implemented

```
int ManyObj::count = 0;

ManyObj::ManyObj() { index = count++;}
ManyObj::~ManyObj() {count--;}
int ManyObj::getIndex() {return index;}
int ManyObj::howMany() {return count;}
```

- There is a famous problem with static members, known as the *static initialization order failure*

The static initialization fiasco

- When static members are complex objects, that depend on each other, we have to be careful with the order of initialization
  - initialization is performed just after the loading, and before the main starts.
  - Within a specific translation unit, the order of initialization of static objects is guaranteed to be the order in which the object definitions appear in that translation unit. The order of destruction is guaranteed to be the reverse of the order of initialization.
  - However, there is no guarantee concerning the order of initialization of static objects across translation units, and the language provides no way to specify this order. (undefined in C++ standard)
  - If a static object of class A depends on a static object of class B, we have to make sure that the second object is initialized before the first one

Solutions

- The **Nifty counter** (or Schwartz counter) technique
  - Used in the standard library, quite complex as it requires an extra class that takes care of the initialization
- The **Construction on first use** technique
  - Much simpler, use the initialization inside function

Construction on first use

- It takes advantage of the following C/C++ property
  - Static objects inside functions are only initialized on the first call
- Therefore, the idea is to declare the static objects inside global functions that return references to the objects themselves
- access to the static objects happens only through those global functions (see Singleton)

Copy constructors and static members

- What happens if the copy constructor is called?

```
void func(ManyObj a)
{
    ...
}

void main()
{
    ManyObj a;
    func(a);
    cout << "How many: " << ManyObj::howMany() << "\n";
}
```

- What is the output?

Copy constructors and static members

- What happens if the copy constructor is called?

```
void func(ManyObj a)
{
    ...
}

void main()
{
    ManyObj a;
    func(a);
    cout << "How many: " << ManyObj::howMany() << "\n";
}
```

- What is the output?
- Solution in `./examples/02.basics-examples/manyobj.cpp`

Outline

Constants

- In C++, when something is const it means that it cannot change. Period.
- Now, the particular meanings of const are a lot:
  - Don't to get lost! Keep in mind: const $=$ cannot change
- Another thing to remember:
  - constants must have an initial (and final) value!

Constants - I

- As a first use, const can substitute the use of **`#define`** in C
  - whenever you need a constant global value, use const instead of a define, because it is clean and it is type-safe

```
#define PI 3.14          // C style
const double pi = 3.14; // C++ style
```

- In this case, the compiler does not allocate storage for pi
- In any case, the const object has an *internal linkage*

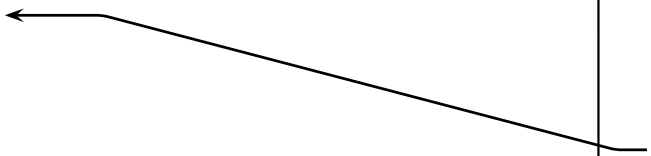
Constants - II

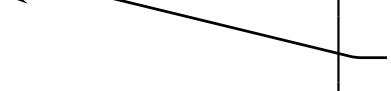
- You can use const for variables that never change after initialization. However, their initial value is decided at run-time

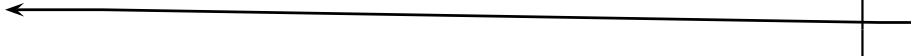
```
const int i = 100;
const int j = i + 10;

int main()
{
    cout << "Type a character\n";
    const char c = cin.get();
    const char c2 = c + 'a';
    cout << c2;

    c2++;
}
```







Constant pointers

- There are two possibilities
  - the pointer itself is constant
  - the pointed object is constant

```
int a
int * const u = &a;

const int *v;
```

the pointer is constant

the pointed object is constant (the pointer can change and point to another const int!)

- Remember: a const object needs an initial value!

Const function arguments

- An argument can be declared constant. It means the function can't change it
- it's particularly useful with references

```
class A {
public:
    int i;
};

void f(const A &a) {
  a.i++;      // error! cannot modify a;
}
```

- You can do the same thing with a pointer to a constant, but the syntax is messy.

Passing by const reference

- Remember:
  - we can pass argument by value, by pointer or by reference
  - in the last two cases we can declare the pointer or the reference to refer to a constant object: it means the function cannot change it
  - Passing by constant reference is equivalent, from the user point of view, to passing by value
  - From an implementation point of view, passing by const reference is much faster!!

Constant member functions

- A member function can be declared constant
- It means that it will not modify the object

```
class A {
    int i;
public:
    int f() const;
    void g();
};
void A::f() const
{
    i++;        // ERROR! this function cannot
                // modify the object
    return i;   // Ok
}
```

Constant member functions II

- The compiler can call only const member functions on a const object!

```
const A a = ...;

a.f();        // Ok
a.g();        // ERROR!!
```

Constant return value

- This is tricky! We want to say: “the object we are returning from this function cannot be modified”
- This is meaningless when returning predefined types

```
const int f1(int a) {return ++a;}

int f2(int a) {return ++a;}

int i = f1(5);    // legal
i = f2(5);

const int j = f1(5); // also legal
const int k = f2(5); //also legal
```

these two functions are equivalent!

Return mechanism

- When returning a value, the compiler copies it into an appropriate location, where the caller can use it

```
int f2(int a)
{
    return ++a;
}

int i = f2(5);
```

1. a is allocated on the stack
2. the compiler copies 5 into a
3. a is incremented
4. the modified value of a is then copied directly into i
5. a is de-allocated (de-structed)

- why const does not matter?
  - since the compiler copies the value into the new location, who cares if the original return value is constant? It is deallocated right after the copy!
- For objects, it is much more complex... (next lecture)