Object Oriented Software Design II

Introduction to C++

Giuseppe Lipari

Scuola Superiore Sant'Anna – Pisa

February 20, 2013

Tools

- On Linux:
  - Just install the latest g++ compiler
  - You can use any editor (e.g. gedit, kate, etc.)
  - Eclipse with CDT is an IDE (Integrated Developement Environment) that you can use to simplify multi-file projects
  - However, at the very beginning, I recommend command-line tools
- For Windows
  - You can use Visual C++ (if you have a license available)
  - otherwise I recommend installing Cygwin (`http://www.cygwin.com/`), and from there install the latest g++ compiler
  - Again, use any editor you want, and then the command line for compiling and running the code
  - Nice editors: (`http://notepad-plus-plus.org/`, `http://www.ultraedit.com/products/ultraedit.html`, but also emacs and gVim)

Prerequisites

- To understand this course, you should at least know the basic C syntax
  - functions declaration and function call,
  - global and local variables
  - pointers (will do again during the course)
  - structures
- First part of the course: classes
  - Classes, objects, memory layout
  - Pointer and references
  - Copying
  - Inheritance, multiple inheritance
  - Access rules
  - Public, protected and private inheritance
  - Exceptions

Abstraction

- An essential instrument for OO programming is the support for data abstraction
- C++ permits to define new types and their operations
- Creating a new data type means defining:
  - Which elements it is composed of (*internal structure*);
  - How it is built/destroyed (*constructor/destructor*);
  - How we can operate on this type (*methods/operations*).

Summary - cont.

- Second part: templates
  - Templates
  - The Standard Template Library
- Third part: new standard
  - What does it change
  - lambda functions
  - auto
  - move semantic
  - new STL classes
  - Safety to exceptions
- Fourth part: patterns
  - Some patterns in C++
  - Function objects
  - Template patterns
  - Meta-programming with templates
- Fifth part: libraries
  - Thread library, synchronization
  - Futures and promises
  - The Active Object pattern

Classical example

```
class Complex {
   double real_;
   double imaginary_;
public:
     Complex();
     Complex(double a, double b);
     ~Complex();

     double real() const;
     double imaginary() const;
     double module() const;
     Complex &operator =(const Complex &a);
     Complex &operator+=(const Complex &a);
     Complex &operator-=(const Complex &a));
};
```

How to use complex

```
Complex c1;          // default constructor
Complex c2(1,2);     // constructor
Complex c3(3,4);     // constructor

cout << "c1=(" << c1.real() << ","
     << c1.imaginary() << ")" << endl;

c1  = c2;            // assignment
c3 += c1;            // operator +=
c1  = c2 + c3;       // ERROR: operator + not yet defined
```


Using new data types

- The new data type is used just like a predefined data type
  - it is possible to define new functions for that type:
    - `real()`, `imaginary()` and `module()`
  - It is possible to define new operators
    - =, += and −=
  - The compiler knows automatically which function/operator must be invoked
- C++ is a strongly typed language
  - the compiler knows which function to invoke by looking at the type

Members

- A class contains members
- A member can be
  - any kind of variable (member variables)
  - any kind of function (member functions or methods)

```
class MyClass {
  int a;
  double b;
public:
  int c;

  void f();
  int getA();
  int modify(double b);
};
```

member variables (private)

member variable (public)

member functions (public)

Declaring objects of a class: constructor

- An **object** is an instance of a class
- An object is created by calling a special function called *constructor*
  - A constructor is a function that has the same name of the class and no return value
  - It may or may not have parameters;
  - It is invoked in a special way

```
class MyClass {
public:
  MyClass()
  {
    cout << "Constructor"<<endl;
  }
};

MyClass obj;
```

Declaration of the constructor

Invoke the constructor to create an object

Constructor - II

- Constructors with parameters

```
class MyClass {
  int a;
  int b;
public:
  MyClass(int x);
  MyClass(int x, int y);
};

MyClass obj;
MyClass obj1(2);
MyClass obj2(2,3);

int myvar(2);
double pi(3.14);
```

A class can have many constructors

This is an error, no constructor without parameters

Calls the first constructor

Calls the second constructor

Same syntax is valid for primitive data types

Default constructor

- Rules for constructors
  - If you do not specify a constructor, a default one with no parameters is provided by the compiler
  - If you provide a constructor (any constructor) the compiler will not provide a default one for you
- Constructors are used to initialise members

```
class MyClass {
  int a;
  int b;
public:
  MyClass(int x, int y)
  {
    a = x; b = 2*y;
  }
};
```

Initialization list

- Members can be initialised through a special syntax
  - This syntax is preferable (the compiler can catch some obvious mistake)
  - use it whenever you can (i.e. almost always)

```
class MyClass {
  int a;
  int b;
public:
  MyClass(int x, int y) :
    a(x), b(y)
  {
     // other initialisation
  }
};
```

A comma separated list of constructors, following the :

Access control

- A member can be:
  - private: only member functions of the same class can access it; other classes or global functions can't
  - protected: only member functions of the same class or of derived classes can access it: other classes or global functions can't
  - public: every function can access it

```
class MyClass {
private:
    int a;
public:
     int c;
};
```

```
MyClass data;

cout << data.a;    // ERROR!
cout << data.c;    // OK
```


Accessing member objects

- Members of one object can be accessed using the *dot* notation, similarly to structs in C

```
class MyClass {
public:
  int a;
  int f();
  void g(int i, int ii);
};

MyClass x;
MyClass y;

x.a = 5;
y.a = 7;
x.f();
y.g(5, 10);
```

Assigning to a member variable of object x

Assigning to a member variable of object y

Calling member function f() of object x

Calling member function g() of object y

Friends

```
class A {
   friend class B;
   int y;
   void f();
public:
   int g();
};

class B {
   int x;
public:
   void f(A &a);
};

void B::f(A &a)
{
    x = a.y;
    a.f();
}
```

B is friend of A, hence B can access private members of A

Implementing member functions

- You can implement a member function (including constructors) in a separate .cpp file

complex.h

```
class Complex {
  double real_;
  double img_;
public:
  ...
  double module() const;
  ...
};
```

complex.cpp

```
double Complex::module()
{
  double temp;
  temp = real_ * real_ +
         img_ * img_;
  return temp;
}
```

- This is preferable most of the times
- put implementation in include files only if you hope to use *in-lining* optimisation

Friend functions and operator

- Even a global function or a single member function can be friend of a class

```
class A {
   friend B::f();
   friend h();
   int y;
   void f();
public:
   int g();
};
```

friend member function

friend global function

- It is better to use the *friend* keyword only when it is really necessary because it breaks the access rules.
- *"Friends, much as in real life, are often more trouble than their worth." – Scott Meyers*

Dynamic memory

- Dynamic memory is managed by the user
- In C:
  - to allocate memory, call function `malloc`
  - to deallocate, call `free`
  - Both take pointers to any type, so they are not type-safe
- In C++
  - to allocate memory, use operator `new`
  - to deallocate, use operator `delete`
  - they are more type-safe

Function overloading

- In C++, the argument list is part of the name of the function
  - this mysterious sentence means that two functions with the same name but with different argument list are considered two different functions and not a mistake
- If you look at the internal name used by the compiler for a function, you will see three parts:
  - the class name
  - the function name
  - the argument list

Destructor

- The destructor is called just before the object is deallocated.
- It is always called both for all objects (allocated on the stack, in global memory, or dynamically)
- If the programmer does not define a constructor, the compiler automatically adds one by default (which does nothing)
- Syntax

```
class A {
    ...
public:
    A() { ... }  // constructor
    ~A() { ... } // destructor
};
```

The destructor never takes any parameter

Function overloading

```
class A {
public:
    void f(int a);
    void f(int a, int b);
    void f(double g);
};
class B {
public:
    void f(int a);
};
```

- `__A_f_int`
- `__A_f_int_int`
- `__A_f_double`
- `__B_f_int`

- To the compiler, they are all different functions!
- beware of the type...

New and delete for arrays

- To allocate an array, use this form

```
int *p = new int[5]; // allocates an array of 5 int
...
delete [] p;         // notice the delete syntax

A *q = new A[10];    // allocates an array of 10
...                  // objects of type A
delete [] q;
```

- In the second case, the default constructor is called to build the 10 objects
- Therefore, this can only be done is a default constructor (without arguments) is available

Return values

- Notice that return values are not part of the name
  - the compiler is not able to distinguish two functions that differs only on return values

```
class A {
    int floor(double a);
    double floor(double a);
};
```

- This causes a compilation error
- It is not possible to overload a return value

Default arguments in functions

- Sometime, functions have long argument lists
- Some of these arguments do not change often
  - We would like to set default values for some argument
  - This is a little different from overloading, since it is the same function we are calling!

```
int f(int a, int b = 0);

f(12);    // it is equivalent to f(12,0);
```

- The combination of overloading with default arguments can be confusing
- it is a good idea to avoid overusing both of them

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

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
```


Reference vs pointer

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

Pointers	References
Pointers are like other variables	Must be initialised
Can have a pointer to `void`	Cannot have references to void
Can be assigned arbitrary values	Cannot be assigned
It is possible to do arithmetic	Cannot do arithmetic

- What are references good for?

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);
```


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!

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 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() {...}
```


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;
}
```


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!

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

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++;
}
```

Compile-time constants (i, j)

run-time constants (c2)

ERROR! c2 is const! (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.

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
}
```


Operator overloading

- After all, an operator is like a function
  - binary operator: takes two arguments
  - unary operator: takes one argument
- The syntax is the following:
  - `Complex &operator+=(const Complex &c);`
- Of course, if we apply operators to predefined types, the compiler does not insert a function call

```
int a = 0;
a += 4;

Complex b;
Complex c(1,3);
b += 5;
c = b;
```

Default constructor

Constructor

Sum operator

Assignment operator

A complete example

```
class Complex {
   double real_;
   double imaginary_;
public:
    Complex();                        // default constructor
    Complex(double a, double b = 0); // constructor
    ~Complex();                       // destructor
    Complex(const Complex &c);        // copy constructor

    double real() const;            // member function
    double imaginary() const;       // member function
    double module() const;          // member function
    Complex &operator =(const Complex &a);  // assignment operator
    Complex &operator+=(const Complex &a);  // sum operator
    Complex &operator-=(const Complex &a)); // sub operator
};

Complex operator+(const Complex &a, const Complex &b);
Complex operator-(const Complex &a, const Complex &b);
```


To be member or not to be...

- In general, operators that modify the object (like `++`, `+=`, `--`, etc...) should be member
- Operators that do not modify the object (like `+`, `-`, etc,) should not be member, but friend functions
- Let's write `operator+` for complex:
  `./examples/03.operators-examples/complex.cpp`
- Not all operators can be overloaded
  - we cannot "invent" new operators,
  - we can only overload existing ones
  - we cannot change number of arguments
  - we cannot change precedence
  - . (dot) cannot be overloaded

Strange operators

- You can overload
  - new and delete
    - used to build custom memory allocate strategies
  - operator[]
    - for example, in vector<>...
  - operator,
    - You can write very funny programs!
  - operator->
    - used to make smart pointers!!

Example

```
class MyClass {
    int x;
    int y;
public:
    MyClass(int a, int b) : x(a), y(b) {}
    int getX() const;
    int getY() const;
};

ostream& operator<<(ostream& out, const MyClass &c) {
    out << "[" << c.getX() << ", " << c.getY() << "]";
    return out;
}

int main() {
   MyClass obj(1,3);
   cout << "Oggetto: " << obj << endl;
}
```

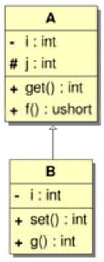

How to overload operator []

- the prototype is the following:

```
class A {
   ...
public:
   A& operator[](int index);
};
```

- Exercise:
  - add operator [] to you Stack class
  - the operator must never go out of range

Inheritance




```
class A {
   int i;
protected:
   int j;
public:
   A() : i(0),j(0) {};
   ~A() {};
   int get() const {return i;}
   int f() const {return j;}
};

class B : public A {
  int i;
public:
  B() : A(),  i(0) {};
  ~B() {};
  void set(int a) {j  = a; i+= j}
  int g() const {return i;}
};
```


Output on streams

- It is possible to overload `operator<<()` and `operator>>()`
- This can be useful to output an object on the terminal
- Typical way to define the operator

```
ostream & operator<<(ostream &out, const MyClass &obj);
```

- An example is worth a thousands words

Syntax

- How to define the derived class

```
class B : public A {
  int i;
public:
  B() : A(),
        i(0)
  {}
  ~B() {}
  void set(int a) {
       j = a;
       i+= j;
  }
  int g() const {
       return i;
  }
};
```

- class B derives publicly from A
- Therefore, to construct B, we must first construct A
- j is a member of A declared as protected; therefore, B can access it
- i instead is a member of B. There if another i that is a private member of A, so it cannot be accessed from B

Overloading and hiding

- There is no overloading across classes

```
class A {
    ...
public:
    int f(int, double);
}

class B : public A {
    ...
public:
    void f(double);
}
```

```
int main()
{
    B b;
    b.f(2,3.0);
// ERROR!
}
```

- A::f() has been hidden by B::f()
- to get A::f() into scope, the using directive is necessary
- using A::f(int, double);

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

```
class A {
public:
    void f() { ... }
};
class B : public A {
public:
    void g() { ... }
};

A* p;

p = new B();

p->f();

p->g();
```

A pointer to the base class

The pointer now points to an object of a derived class

Call a function of the interface of the base class: correct

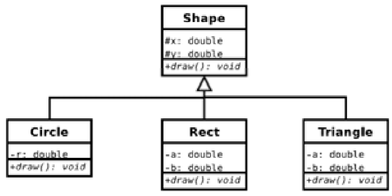
Error! g() is not in the interface of the base class, so it cannot be called through a pointer to the base class!

Extension through inheritance

- Why this is useful?
  - All functions that take a reference (or a pointer) to A as a parameter, continue to be valid and work correctly when we pass a reference (or a pointer) to B
  - This means that we can *reuse* all code that has been written for A, also for B
  - In addition, we can write additional code specifically for B
- Therefore,
  - we can **reuse** existing code also with the new class
  - We can extend existing class to implement new functionality
- What about modifying (customize, extend, etc.) the behaviour of existing code *without changing it*?

Virtual functions

- Let's introduce virtual functions with an example



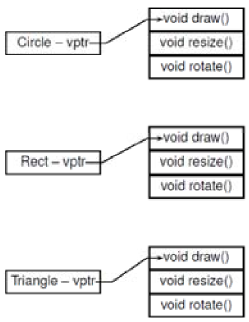
Implementation

```
class Shape {
protected:
  double x,y;
public:
  Shape(double x1, double y2);
  virtual void draw() = 0;
};

class Circle : public Shape {
  double r;
public:
  Circle(double x1, double y1,
         double r);
  virtual void draw();
};
```

```
class Rect : public Shape {
  double a, b;
public:
  Rect(double x1, double y1,
       double a1, double b1);
  virtual void draw();
};

class Triangle : public Shape {
  double a, b;
public:
  Triangle(double x1, double y1,
           double a1, double b1);
  virtual void draw();
};
```


Virtual table

- When you put the virtual keyword before a function declaration, the compiler builds a vtable for each class



Calling a virtual function

- When the compiler sees a call to a virtual function, it performs a late binding, or dynamic binding
  - each object of a class derived from `Shape` has a `vptr` as first element.
    - It is like a hidden member variable
- The virtual function call is translated into
  - get the `vptr` (first element of object)
  - move to the right position into the vtable (depending on which virtual function we are calling)
  - call the function

When inheritance is used

- Inheritance should be used when we have a *isA* relation between objects
  - you can say that a circle is a kind of shape
  - you can say that a rect is a shape
- What if the derived class contains some special function that is useful only for that class?
  - Suppose that we need to compute the diagonal of a rectangle

Pure virtual functions

- A virtual function is pure if no implementation is provided
- Example:

```
class Abs {
public:
   virtual int fun() = 0;
   virtual ~Abs();
};
class Derived public Abs {
public:
   Derived();
   virtual int fun();
   virtual ~Derived();
};
```

This is a pure virtual function. No object of Abs can be instantiated.

One of the derived classes must *finalize* the function to be able to instantiate the object.

isA vs. *isLikeA*

- If we put function `diagonal()` only in `Rect`, we cannot call it with a pointer to shape
  - In fact, `diagonal()` is not part of the interface of shape
- If we put function `diagonal()` in `Shape`, it is inherited by `Triangle` and `Circle`
  - `diagonal()` does not make sense for a `Circle`
  - we should raise an error when `diagonal()` is called on a `Circle`
- One solution is to put the function in the `Shape` interface
  - it will return an error for the other classes, like `Triangle` and `Circle`
- another solution is to put it only in `Rect` and then make a *downcasting* when necessary
  - see `./examples/05.multiple-inheritance-examples/shapes_` for the two solutions

Interface classes

- If a class only provides pure virtual functions, it is an *interface class*
  - an interface class is useful when we want to specify that a certain class *conforms* to an interface
  - Unlike Java, there is no special keyword to indicate an interface class
  - more examples in section multiple inheritance

Downcasting

- One way to downcast is to use the `dynamic_cast` construct

```
class Shape { ... };

class Circle : public Shape { ... };

void f(Shape *s)
{
  Circle *c;

  c = dynamic_cast<Circle *>(s);
  if (c == 0) {
    // s does not point to a circle
  }
  else {
    // s (and c) points to a circle
  }
}
```

Casting

- Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to.
- The subsequent call to member result will produce either a run-time error or a unexpected result.
- There are more safe way to perform casting:

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```


dynamic_cast

- `dynamic_cast` can be used only with pointers and references to objects.
- Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.
- The result is the pointer itself if the conversion is possible;
- The result is `nullptr` if the conversion is not possible:

```
class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;
    pd = dynamic_cast<CDerived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast" << endl;
    pd = dynamic_cast<CDerived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast" << endl;
    return 0;
}
```


static_cast

- `static_cast` can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived.
- however, no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.
- Therefore, it is up to the programmer to ensure that the conversion is safe.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

- `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

Private inheritance

- Until now we have seen **public inheritance**
  - A derived class inherits the interface **and** the implementation of a base class
- With **private inheritance** it is possible to inherit only the implementation

```
class Base {
   int p;
protected:
   int q;
public:
   int f();
};
class Derived : private Base {
public:
   int g();
};
int main() {
   Derived obj;
   obj.g();
}
```

Private inheritance

Can access `q` and `f()`

I can only call `g()` but not `f()`

Private inheritance

- Private inheritance does not model the classical *isA* relationship
- In particular, it is not possible to automatically upcast from derived to base class

```
class Base {};
class DerivedA : public Base {};
class DerivedB : private Base {};

Base *ptr;
DerivedA pub;
DerivedB priv;

ptr = &pub;   // ok
ptr = &priv;  // error!!
```

`DerivedB` cannot be accessed as `Base`

Inheritance rules

Public Inheritance

In Base	In Derived	Client
private	cannot access	cannot access
protected	as protected members	cannot access
public	as public members	can access

Protected Inheritance

In Base	In Derived	Client
private	cannot access	cannot access
protected	as protected members	cannot access
public	as protected members	cannot access

Private Inheritance

In Base	In Derived	client
private	cannot access	cannot access
protected	cannot access	cannot access
public	as private members	cannot access

Private Inheritance

- Why private inheritance?
  - Because we want to re-use implementation but not the interface
  - It can be seen as a sort of composition
- When to use it
  - It is not a popular technique
  - Composition is better done by declaring a member to another class

Composition

```
class B {
    A* ptr;
public:
    B() {
        ptr = new A();
    }
    ~B() {
        delete ptr;
    }
};
```

Private Inheritance

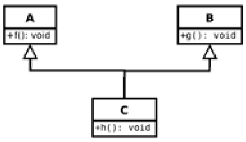
```
class B : private A {
public:
    B() : A() {
    }
    ~B() {
    }
};
```


Why multiple inheritance?

- Is multiple inheritance really needed?
  - There are contrasts in the OO research community
  - Many OO languages do not support multiple inheritance
  - Some languages support the concept of "Interface" (e.g. Java)
- Multiple inheritance can bring several problems both to the programmers and to language designers
- Therefore, the much simpler *interface inheritance* is used (that mimics Java interfaces)

Multiple inheritance

- A class can be derived from 2 or more base classes



- C inherits the members of A and B

Pointer to member

- Can I have a pointer to a member of a class?
- The problem with it is that the address of a member is only defined with respect to the address of the object
- The C++ pointer-to-member selects a location inside a class
  - The dilemma here is that a pointer needs an address, but there is no "address" inside a class, only an "offset";
  - selecting a member of a class means offsetting into that class
  - in other words, a *pointer-to-member* is a "relative" offset that can be added to the address of an object

Multiple inheritance

- Syntax

```
class A {
 public:
   void f();
};

class B {
public:
   void f();
};

class C : public A, public B
{
    ...
};
```

- If both A and B define two functions with the same name, there is an ambiguity
  - it can be solved with the scope operator

```
C c1;

c1.A::f();
c1.B::f();
```


Usage

- To define and assign a pointer to member you need the class
- To dereference a pointer-to-member, you need the address of an object

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

int Data::*pm;           // pointer to member
pm = &Data::x;           // assignment
Data aa;                 // object
Data *pa = &aa;          // pointer to object
pa->*pm = 5;             // assignment to aa.x
aa.*pm = 10;             // another assignment to aa.x
pm = &Data::y;
aa.*pm = 20;             // assignment to aa.y
```

Syntax for pointer-to-member functions

- For member functions, the syntax is very similar:

```
class Simple2 {
public:
  int f(float) const { return 1; }
};

int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;

int main() {
  fp = &Simple2::f;

  Simple2 obj;
  Simple2 *p = &obj;

  p->*fp(.5);        // calling the function
  obj.*fp(.8);       // calling it again
}
```