Design Patterns in C++

Summary of C++ features

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

Scuola Superiore Sant'Anna – Pisa

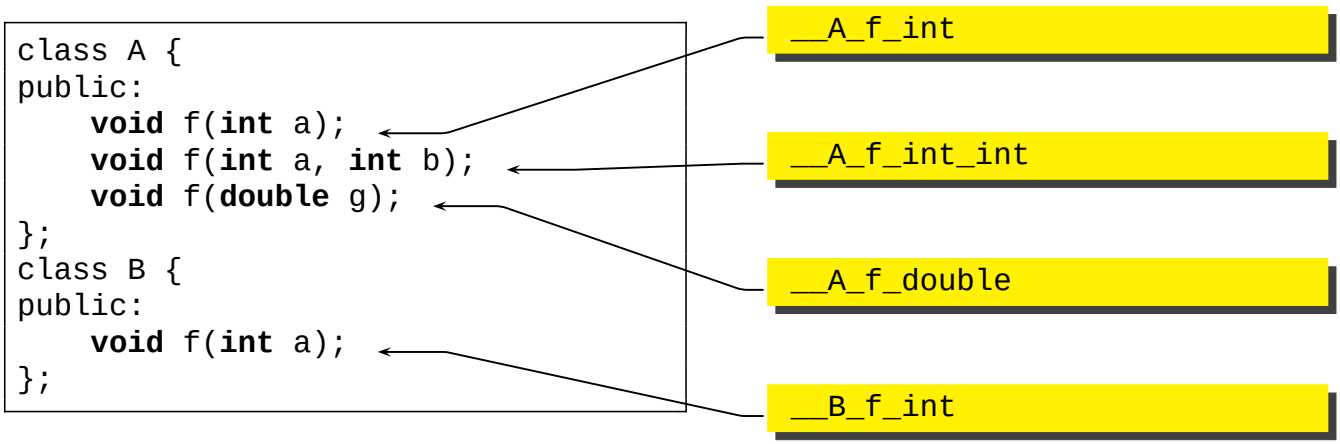
March 13, 2011

Outline

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

Function overloading

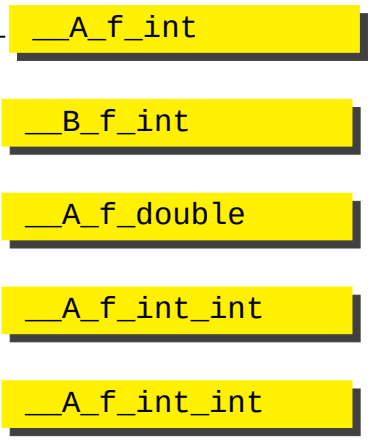


- To the compiler, they are all different functions!
- beware of the type...

Which one is called?

```
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 a;
B b;

a.f(5);

b.f(2);

a.f(3.0);
a.f(2,3);
a.f(2.5, 3);
```



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

Pointers

- We can define a pointer to an object

```
class A { ... };

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

- Pointer p contains the address of `myobj`

Pointers - II

- 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);
```

- After the function call, x=0 while y = 7
- x is passed by value (i.e. it is copied into a)
- y is passed by address (i.e. we pass its address, so that it can be modified inside the function)

Another example

pointerarg.cpp

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

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;          //   "       "
```

- Until now, all is normal

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

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 ponter-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
}
```


Another example

```
class Widget {
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
  enum { cnt = 4 };
  void (Widget::*fptr[cnt])(int) const;
public:
  Widget() {
    fptr[0] = &Widget::f; // Full spec required
    fptr[1] = &Widget::g;
    fptr[2] = &Widget::h;
    fptr[3] = &Widget::i;
  }
  void select(int i, int j) {
    if(i < 0 || i >= cnt) return;
    (this->*fptr[i])(j);
  }
  int count() { return cnt; }
};

int main() {
  Widget w;
  for(int i = 0; i < w.count(); i++)
    w.select(i, 47);
}
```

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

References

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

- What are references good for?

Reference example

referencearg.cpp

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

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

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

copy1.cpp

```
    c.fun(5);
}

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

int main() {
    MyClass obj(2);

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

- 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:

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

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() {...}
```


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

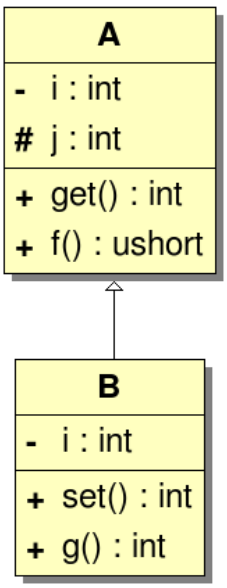
Construction on first use

- 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)

Code reuse

- In C++ (like in all OO programming), one of the goals is to re-use existing code
- There are two ways of accomplishing this goal: composition and inheritance
  - Composition consists defining the object to reuse inside the new object
  - Composition can also expressed by relating different objects with pointers each other
  - Inheritance consists in enhancing an existing class with new more specific code

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;}
};
```

Use of Inheritance

- Now we can use B as a special version of A

```
int main()
{
   B b;
   cout << b.get() << endl; // calls A::get();
   b.set(10);
   cout << b.g() << endl;
   b.g();
   A *a = &b;  // Automatic type conversion (upcasting)
   a->f();
  B *p = new A;
}
```


Overloading and hiding

- There is no overloading across classes

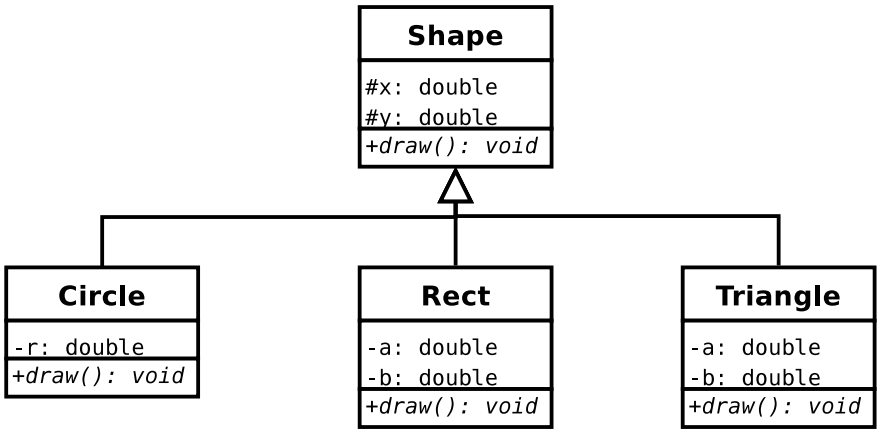
```
class A {
   ...
public:
   int f(int, double);
}

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);`

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();
};
```

We would like to collect shapes

- Let's make a vector of shapes

```
vector<Shapes *> shapes;

shapes.push_back(new Circle(2,3,10));
shapes.push_back(new Rect(10,10,5,4));
shapes.push_back(new Triangle(0,0,3,2));

// now we want to draw all the shapes ...

for (int i=0; i<3; ++i) shapes[i]->draw();
```

- We would like that the right draw function is called
- However, the problem is that Shapes::draw() is called
- The solution is to make draw virtual

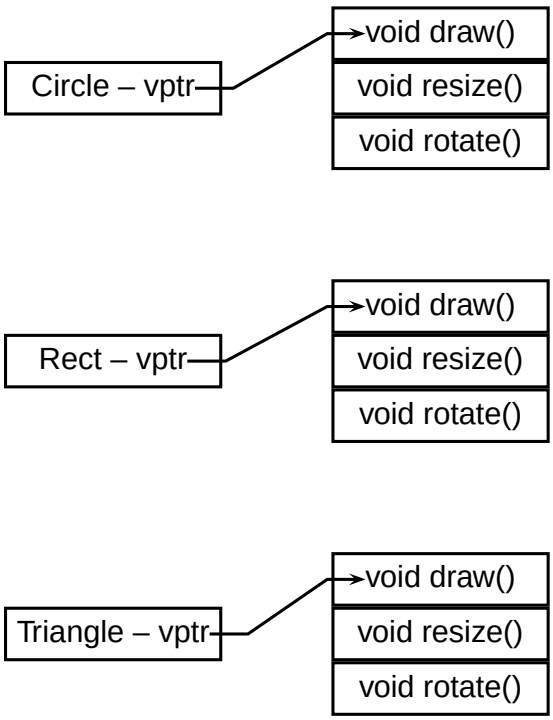
Virtual functions

```
class Shape {
protected:
  double x,y;
public:
  Shape(double xx, double yy);
  void move(double x, double y);
  virtual void draw();
  virtual void resize(double scale);
  virtual void rotate(double degree);
};

class Circle : public Shape {
  double r;
public:
  Circle(double x, double y,
         double r);
  void draw();
  void resize(double scale);
  void rotate(double degree);
};
```

- `move()` is a regular function
- `draw()`, `resize()` and `rotate()` are virtual
- see shapes/

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

Dynamic binding vs static binding

Which function are called in the following code?

```
class A {
public:
    void f() { cout << "A::f()" << endl; g(); }
    virtual void g() { cout << "A::g()" << endl; }
};
class B : public A {
public:
    void f() { cout << "B::f()" << endl; g(); }
    virtual void g() { cout << "B::g()" << endl; }
};
...

A *p = new B;
p->g();
p->f();

B b;
A &r = b;
r.g();
r.f();
```


Overloading and overriding

- When you override a virtual function, you cannot change the return value
  - Simply because the compiler will not know which function to actually call
- There is only one exception to the previous rule:
  - if the base class virtual method returns a pointer or a reference to an object of the base class . . .
  - . . . the derived class can change the return value to a pointer or reference of the derived class

Overload and override

- Examples

Correct

```
class A {
public:
   virtual A& f();
   int g();
};

class B: public A {
public:
   virtual B& f();
   double g();
};
```

Wrong

```
class A {
public:
   virtual A& f();
};

class C: public A {
public:
   virtual int f();
};
```


Overloading and overriding

- When you override a virtual function, you cannot change the return value
  - Simply because the compiler will not know which function to actually call
- There is only one exception to the previous rule:
  - if the base class virtual method returns a pointer or a reference to an object of the base class . . .
  - . . . the derived class can change the return value to a pointer or reference of the derived class

Overload and override

- Examples

Correct

```
class A {
public:
   virtual A& f();
   int g();
};

class B: public A {
public:
   virtual B& f();
   double g();
};
```

Wrong

```
class A {
public:
   virtual A& f();
};

class C: public A {
public:
   virtual int f();
};
```


Destructors

- What happens if we try to destruct an object through a pointer to the base class?

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

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

int main() {
  A *p;
  p = new B;
  // ...
  delete p;
}
```

Virtual destructor

- This is a big mistake!
  - The destructor of the base class is called, which “destroys” only part of the object
  - You will soon end up with a segmentation fault (or illegal access), or memory corruption
- To solve the problem, we have to declare a virtual destructor
  - If the destructors are virtual, they are called in the correct order
  - See

Restrictions

- Never call a virtual function inside a destructor!
  - Can you explain why?
- You can not call a virtual function inside a constructor
  - in fact, in the constructor, the object is only half-built, so you could end up making a wrong thing
  - during construction, the object is not yet ready! The constructor should only build the object
- Same thing for the destructor
  - during destruction, the object is half destroyed, so you will probably call the wrong function

Restrictions

- Example

```
class Base {
  string name;
public:
  Base(const string &n) : name(n) {}
  virtual string getName() { return name; }
  virtual ~Base() { cout << getName() << endl;}
};
```

```
class Derived : public Base {
  string name2;
public:
  Derived(const string &n) : Base(n), name(n + "2") {}
  virtual string getName() {return name2;}
  virtual ~Derived() {}
};
```


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.

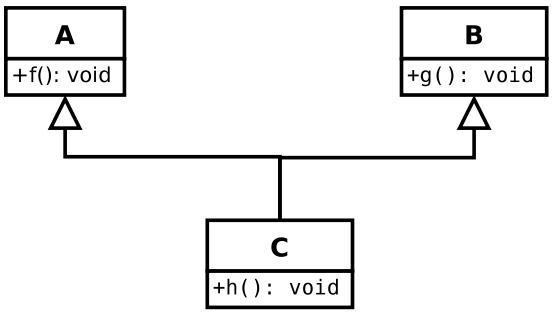
One of the derived classes must *finalize* the function to be able to instantiate the object.

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

Multiple inheritance

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



- C inherits the members of A and B

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

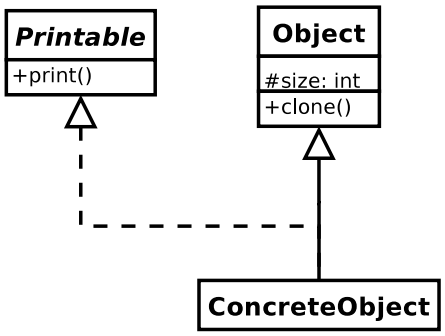
```
C c1;

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


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)

Interface inheritance

- It is called interface inheritance when an onjecy derives from a base class and from an *interface class*
- A simple example

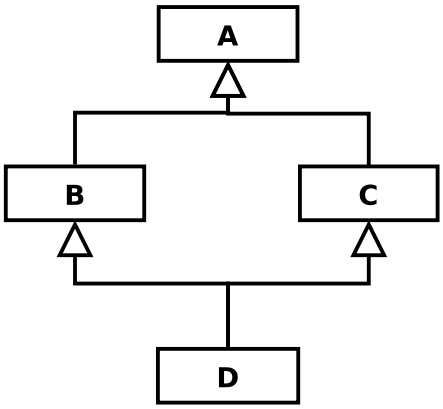


Interface and implementation inheritance

- In *interface inheritance*
  - The base class is abstract (only contains the interface)
  - For each method there is only one final implementation in the derived classes
  - It is possible to always understand which function is called
- Implementation inheritance is the one normally used by C++
  - the base class provides some implementation
  - when inheriting from a base class, the derived class inherits its implementation (and not only the interface)

The diamond problem

- What happens if class D inherits from two classes, B and C which both inherith from A?
- This may be a problem in object oriented programming with multiple inheritance!

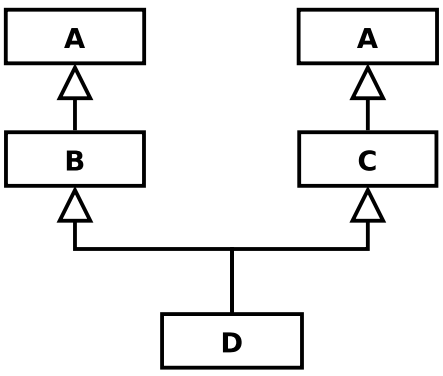


- Problem:
  - If a method in D calls a method defined in A (and does not override the method),
  - and B and C have overridden that method differently,
  - from which class does D inherit the method: B, or C?
  - In C++ this is solved by using keyword “virtual” when inheriting from a class

Virtual base class

- If you do not use virtual inheritance

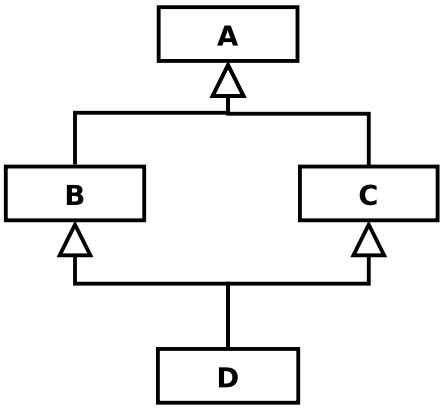
```
class A {...};
class B : public A {...};
class C : public A {...};
class D : public B, public C
{
  ...
};
```



- With public inheritance the base class is duplicated
- To use one of the methods of A, we have to specify which “path” we want to follow with the scope operator
- Cannot upcast!
- see `duplicate.cpp`

Virtual base class

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



- With virtual public inheritance the base class is inherited only once
- see `vbase.cpp` for an example

Initializing virtual base

- The strangest thing in the previous code is the initializer for Top in the Bottom constructor.
- Normally one doesn't worry about initializing subobjects beyond direct base classes, since all classes take care of initializing their own bases.
- There are, however, multiple paths from Bottom to Top,
  - who is responsible for performing the initialization?
- For this reason, the most derived class must initialize a virtual base.
- But what about the expressions in the Left and Right constructors that also initialize Top?
  - they are ignored when a Bottom object is created
  - The compiler takes care of all this for you

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

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
- What to do?

The fat interface

- 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 diagonal/ for the two solutions
- This is a problem of inheritance! Anyway, the second one it probably better

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
  }
}
```

Dynamic cast

- The dynamic_cast() is solved at run-time, by looking inside the structure of the object
- This feature is called *run-time type identification* (RTTI)
- In some compiler, it can be disabled at compile time

Operator oveloading

- 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 = 0;
b += 5;       // function call
```

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 (see complex/)
- 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!!

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

How to overload new and delete

```
class A {
   ...
public:
    void* operator new(size_t size);
    void operator delete(void *);
};
```

- Everytime we call new for creating an object of this class, the overloaded operator will be called
- You can also overload the global version of new and delete

How to overload * and ->

- This is the prototype

```
class Iter {
...
public:
  Obj operator*() const;
  Obj *operator->() const;
};
```

- Why should I overload `operator*()` ?
  - to implement iterators!
- Why should I overload `operator->()` ?
  - to implement smart pointers

Type conversion via constructor

- If you define a constructor that takes as its single argument an object (or reference) of another type, that constructor allows the compiler to perform an automatic type conversion.
- For example,

```
class One {
public:
  One() {}
};

class Two {
public:
  Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
  f(one); // Wants a Two, has a One
}
```

Another example

```
class AA {
  int ii;
public:
  AA(int i) : ii(i) {}
  void print() { cout << ii << endl;}
};
void fun(AA  x) {
  x.print();
}
int main()
{
   fun(5);
}
```

- The integer is “converted” into an object of class AA

Preventing implicit conversion

- To prevent implicit conversion, we can declare the constructor to be `explicit`

```
class AA {
  int ii;
public:
  explicit AA(int i) : ii(i) {}
  void print() { cout << ii << endl;}
};
void fun(AA  x) {
  x.print();
}
int main()
{
   fun(5); // error, no implicit conversion
   fun(AA(5)); // ok, conversion is explicit
}
```

Type conversion through operator

- This is a very special kind of operator:

```
class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
  int x;
public:
  Four(int xx) : x(xx) {}
  operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
  Four four(1);
  g(four);
  g(1);  // Calls Three(1,0)
}
```


Templates

- Templates are used for generic programming
- The general idea is: what we want to reuse is not only the abstract concept, but **the code itself**
- with templates we reuse algorithms by making them general
- As an example, consider the code needed to swap two objects of the same type (i.e. two pointers)

```
void swap(int &a, int &b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
...
int x=5, y=8;
swap(x, y);
```

- Can we make it generic?

Solution

- By using templates, we can write

```
template<class T>
void swap(T &a, T &b)
{
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
...

int x=5, y=8;
swap<int>(x, y);
```

- Apart from the first line, we have just substituted the type `int` with a generic type T

How does it work?

- The template mechanism resembles the macro mechanism in C
  - We can do the same in C by using pre-processing macros:

```
#define swap(type, a, b) { type tmp; tmp=a; a=b; b=tmp; }
...
int x = 5; int y = 8;

swap(int, x, y);
```

- in this case, the C preprocessor substitutes the code
  - it works only if the programmer knows what he is doing
- The template mechanism does something similar
  - but the compiler performs all necessary type checking

Code duplicates

- The compiler will instantiate a version of `swap()` with integer as a internal type
- if you call `swap()` with a different type, the compiler will generate a new version
  - Only when a template is instantiated, the code is generated
    - If we do not use `swap()`, the code is never generated, even if we include it!
    - if there is some error in `swap()`, the compiler will never find it until it tries to generate the code
- Looking from a different point of view:
  - the template mechanism is like cut&paste done by the compiler at compiling time

Swap for other types

- What happens if we call swap for a different type:

```
class A { ... };
A x;
A y;
...

swap<A>(x, y);
```

- A new version of swap is automatically generated
  - Of course, the class A must support the assignment operator, otherwise the generation fails to compile
  - see `./examples/01.cpp_summary-examples/swap.cpp`

Type Deduction

- Parameters can be automatically implied by the compiler

```
int a = 5, b = 8;

swap(a, b);   // equivalent to swap<int>(a, b);
```

- Sometimes, this is not so straightforward . . .

Parameters

- A template can have any number of parameters
- A parameter can be:
  - a class, or any predefined type
  - a function
  - a constant value (a number, a pointer, etc.)

```
template<T, int sz>
class Buffer {
   T v[sz];
   int size_;
public:
   Buffer() : size_(0) {}
};
...
Buffer<char, 127> cbuf;
Buffer<Record, 8> rbuf;
int x = 16;
Buffer<char, x> ebuf; // error!
```

Default values

- Some parameter can have default value

```
template<class T, class Allocator = allocator<T> >
class vector;
```


Templates of templates

- The third type of parameter a template can accept is another class template

```
template<class T>
class Array {
  ...
};

template<class T, template<class> class Seq>
class Container {
  Seq<T> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};

int main() {
  Container<int, Array> container;
  container.append(1);
  container.append(2);
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
}
```

Using standard containers

- If the container class is well-written, it is possible to use any container inside

```
template<class T, template<class U, class = allocator<U> >
         class Seq>
class Container {
  Seq<T> seq; // Default of allocator<T> applied implicitly
public:
  void push_back(const T& t) { seq.push_back(t); }
  typename Seq<T>::iterator begin() { return seq.begin(); }
  typename Seq<T>::iterator end() { return seq.end(); }
};

int main() {
  // Use a vector
  Container<int, vector> vContainer;
  vContainer.push_back(1);
  vContainer.push_back(2);
  for(vector<int>::iterator p = vContainer.begin();
      p != vContainer.end(); ++p) {
    cout << *p << endl;
  }
  // Use a list
  Container<int, list> lContainer;
  lContainer.push_back(3);
  lContainer.push_back(4);
  for(list<int>::iterator p2 = lContainer.begin();
      p2 != lContainer.end(); ++p2) {
    cout << *p2 << endl;
  }
```


The typename keyword

- The typename keyword is needed when we want to specify that an identifier is a type

```
template<class T> class X {
  typename T::id i;    // Without typename, it is an error:
public:
  void f() { i.g(); }
};

class Y {
public:
  class id {
  public:
    void g() {}
  };
};

int main() {
  X<Y> xy;
  xy.f();
}
```

General rule

- if a type referred to inside template code is qualified by a template type parameter, you must use the typename keyword as a prefix,
- unless it appears in a base class specification or initializer list in the same scope (in which case you must not).

Usage

- The typical example of usage is for iterators

```
template<class T, template<class U, class = allocator<U> >
         class Seq>
void printSeq(Seq<T>& seq) {
  for(typename Seq<T>::iterator b = seq.begin();
       b != seq.end();)
    cout << *b++ << endl;
}

int main() {
  // Process a vector
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  printSeq(v);
  // Process a list
  list<int> lst;
  lst.push_back(3);
  lst.push_back(4);
  printSeq(lst);
}
```

Making a member template

- An example for the complex class

```
template<typename T> class complex {
public:
  template<class X> complex(const complex<X>&);
  ...
};

complex<float> z(1, 2);
complex<double> w(z);
```

- In the declaration of `w`, the complex template parameter `T` is double and `X` is float. Member templates make this kind of flexible conversion easy.

Another example

```
int data[5] = { 1, 2, 3, 4, 5 };
  vector<int> v1(data, data+5);
  vector<double> v2(v1.begin(), v1.end());
```

- As long as the elements in `v1` are assignment-compatible with the elements in `v2` (as `double` and `int` are here), all is well.
- The vector class template has the following member template constructor:

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
       const Allocator& = Allocator());
```

- `InputIterator` is interpreted as `vector<int>::iterator`

Another example

```
template<class T> class Outer {
public:
  template<class R> class Inner {
  public:
    void f();
  };
};

template<class T> template<class R>
void Outer<T>::Inner<R>::f() {
  cout << "Outer == " << typeid(T).name() << endl;
  cout << "Inner == " << typeid(R).name() << endl;
  cout << "Full Inner == " << typeid(*this).name() << endl;
}

int main() {
  Outer<int>::Inner<bool> inner;
  inner.f();
}
```


Restrictions

- Member template functions cannot be declared virtual.
  - Current compiler technology expects to be able to determine the size of a class's virtual function table when the class is parsed.
  - Allowing virtual member template functions would require knowing all calls to such member functions everywhere in the program ahead of time.
  - This is not feasible, especially for multi-file projects.

Function templates

- The standard template library defines many function templates in algorithm
  - `sort`, `find`, `accumulate`, `fill`, `binary_search`, `copy`, etc.
- An example:

```
#include <algorithm>
...
int i, j;
...
int z = min<int>(i, j);
```

- Type can be deducted by the compiler
- But the compiler is smart up to a certain limit . . .

```
int z = min(x, j); // x is a double, error, not the same types

int z = min<double>(x, j); // this one works fine
```


Return type

```
template<typename T, typename U>
const T& min(const T& a, const U& b) {
  return (a < b) ? a : b;
}
```

- The problem is: which return value is the most correct? T or U?
- If the return type of a function template is an independent template parameter, you must always specify its type explicitly when you call it, since there is no argument from which to deduce it.

Example

```
template<typename T> T fromString(const std::string& s) {
  std::istringstream is(s);
  T t;
  is >> t;
  return t;
}
template<typename T> std::string toString(const T& t) {
  std::ostringstream s;
  s << t;
  return s.str();
}
int main() {
  int i = 1234;
  cout << "i == \"" << toString(i) << "\"" << endl;
  float x = 567.89;
  cout << "x == \"" << toString(x) << "\"" << endl;
  complex<float> c(1.0, 2.0);
  cout << "c == \"" << toString(c) << "\"" << endl;
  cout << endl;

  i = fromString<int>(string("1234"));
  cout << "i == " << i << endl;
  x = fromString<float>(string("567.89"));
  cout << "x == " << x << endl;
  c = fromString<complex<float> >(string("(1.0,2.0)"));
  cout << "c == " << c << endl;
}
```


Try/catch

- An exception object is *thrown* by the programmer in case of an error condition
- An exception object can be caught inside a try/catch block

```
try {
   //
   // this code can generate exceptions
   //
} catch (ExcType1& e1) {
   // all exceptions of ExcType1 are handled here
}
```

Try/catch

- If the exception is not caught at the level where the function call has been performed, it is automatically forwarded to the upper layer
  - Until it finds a proper try/catch block that *cathes* it
  - or until there is no upper layer (in which case, the program is aborted)

More catches

- It is possible to put more catch blocks in sequence
- they will be processed in order, the first one that catches the exception is the last one to execute

```
try {
   //
   // this code can generate exceptions
   //
} catch (ExcType1&e1) {
   // all exceptions of ExcType1
} catch (ExcType2 &e2) {
  // all exceptions of ExcType2
} catch (...) {
  // every exception
}
```

Re-throwing

- It is possible to re-throw the same exception that has been caught to the upper layers

```
catch(...) {
  cout << "an exception was thrown" << endl;
  // Deallocate your resource here, and then rethrow
  throw;
}
```


Terminate

- In case of abort, the C++ run-time will call the terminate(), which calls abort()
  - It is possible to change this behaviour

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

void terminator() {
  cout << "I'll be back!" << endl; exit(0);
}
void (*old_terminate)() = set_terminate(terminator);

class Botch {
public:
  class Fruit {};
  void f() {
    cout << "Botch::f()" << endl;
    throw Fruit();
  }
  ~Botch() { throw 'c'; }
};

int main() {
  try {
    Botch b; b.f();
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
}
```

Inheritance

```
double mylog(int a)
{
  if (a < = 0) throw LogErr();
  else return log(double(a));
}

void f(int i)
{
  mylog(i);
}

...

try {
  f(-5);
} catch(MathErr &e) {
  cout << e.what() << endl;
}
```

- This code will print “Log of a negative number - log module”
- you can also pass any parameter to LogErr, like the number that cause the error, or the name of the function which caused the error, etc.

Exception specification

- It is possible to specify which exceptions a function might throw, by listing them after the function prototype
- Exceptions are part of the interface!

```
void f(int a) throw(Exc1, Exc2, Exc3);
void g();
void h() throw();
```

- f() can **only** throw exception Exc1, Exc2 or Exc3
- g() can throw **any** exception
- h() **does not** throw any exception

Listing exceptions

- Pay attention: a function must list in the exception list all exception that it may throw, and all exception that all called functions may throw

```
int f() throw(E1) {...}

int g() throw(E2)
{
   ...
   if (cond) throw E2;
   ...
   f();
}
```

It should contain E1 in the list,because g() calls f()

Exception list and inheritance

- if a member function in a base class says it will only throw an exception of type A,
- an override of that function in a derived class must not add any other exception types to the specification list
  - because that would break any programs that adhere to the base class interface.
- You can, however, specify fewer exceptions or none at all, since that doesn't require the user to do anything differently.

Exception list and inheritance

- It is possible to change the specification of an exception with a derived exception

```
class Base {
public:
  class BaseException {};
  class DerivedException : public BaseException {};
  virtual void f() throw(DerivedException) {
    throw DerivedException();
  }
  virtual void g() throw(BaseException) {
    throw BaseException();
  }
};

class Derived : public Base {
public:
  void f() throw(BaseException) {
    throw BaseException();
  }
  virtual void g() throw(DerivedException) {
    throw DerivedException();
  }
}; ///:~
```

- Which one is correct?

Stack unrolling

```
void f() {
   A a;

   if (cond) throw Exc();

}

void g() {
   A *p = new A;

   if (cond) throw Exc();

}
```

At this point, a is destructed

memory pointed by p is **not** automatically deallocated

Resource management

- When writing code with exceptions, it's particularly important that you always ask, "If an exception occurs, will my resources be properly cleaned up?"
- Most of the time you're fairly safe,
- but in constructors there's a particular problem:
  - if an exception is thrown before a constructor is completed, the associated destructor will not be called for that object.
  - Thus, you must be especially diligent while writing your constructor.
- The difficulty is in allocating resources in constructors.
  - If an exception occurs in the constructor, the destructor doesn't get a chance to deallocate the resource.
  - see exceptions/rawp.cpp

How to avoid the problem

- To prevent such resource leaks, you must guard against these "raw" resource allocations in one of two ways:
  - You can catch exceptions inside the constructor and then release the resources
  - You can place the allocations inside an object's constructor, and you can place the deallocations inside an object's destructor.
- The last technique is called Resource Acquisition Is Initialization (RAII for short) because it equates resource control with object lifetime.
- Example: exception_wrap.cpp

Auto ptr

- Dynamic memory is the most frequent resource used in a typical C++ program,
- the standard provides an RAII wrapper for pointers to heap memory that automatically frees the memory.
- The **auto_ptr** class template, defined in the <memory> header, has a constructor that takes a pointer to its generic type
- The **auto_ptr** class template also overloads the pointer operators * and -> to forward these operations to the original pointer
- So you can use the **auto_ptr** object as if it were a raw pointer.