MASTER UNIVERSITARIO in COMPUTER GAME DEVELOPMENT

Advanced C++ Development

Classes II – Inheritance & Polymorphism

Giorgio Ugazio

MILESTONE


Object Oriented Programming

C++ support **Object Oriented Programming** via Inheritance and Polymorphism.

The Inheritance support is via **Class Derivation**.

Class Derivation syntax:

```
class B
{
};

class D: public B // D inherits from B. B is a base class. D is a derived class
{
};
```

Derivation can be **public**, **protected**, **private**. Defaults are public for structs, private for classes

We call **class hierarchy** a set of class related by inheritance relationships

Construction and destruction of Derived

In derived class ctors initializer list you can (sometimes you must) call a base class ctor.

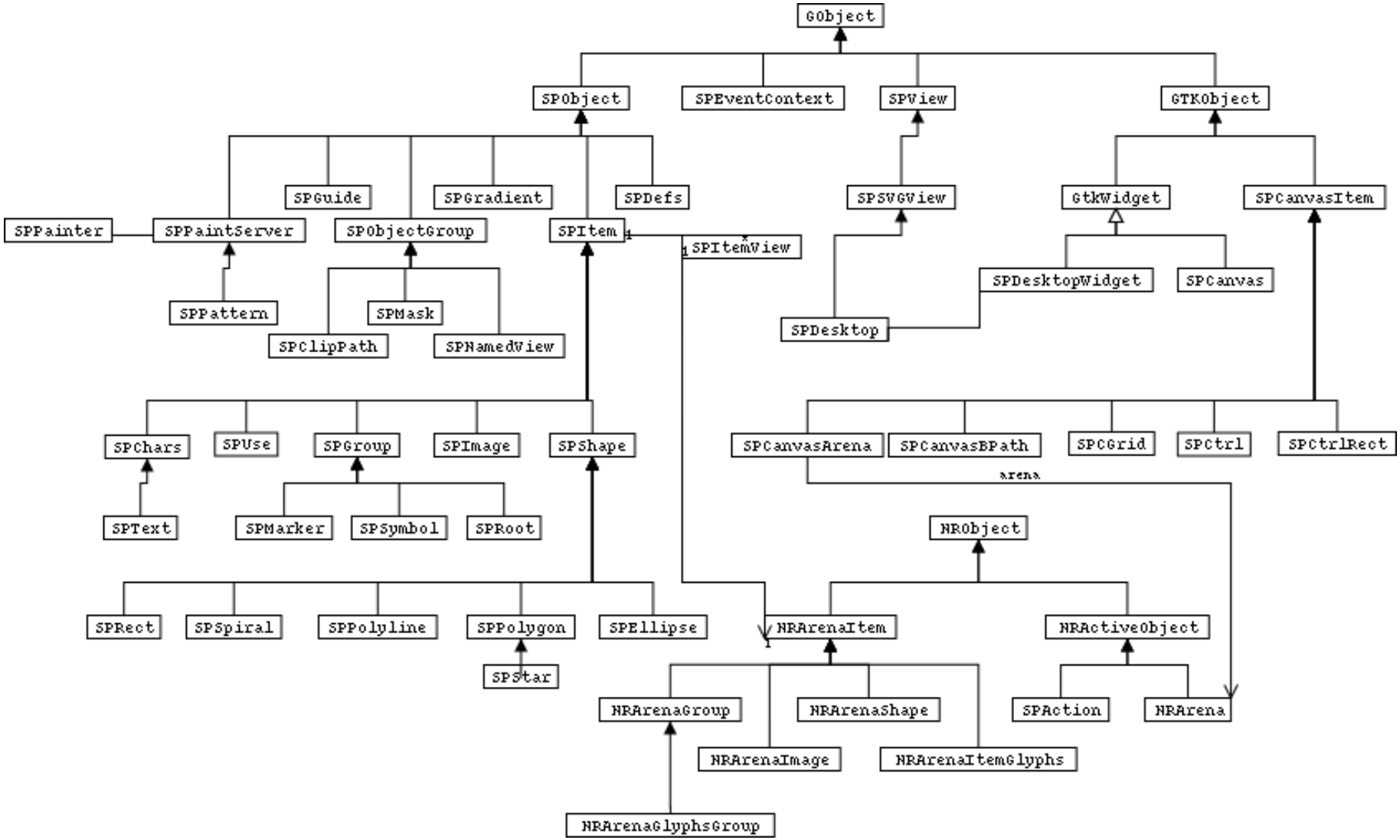
```
class B{B(int);};
class D{D();}

D::D() : B(0) // here you must!
{
}

void f()
{
    D d;
}
```

The order of invocation is B ctor, D ctor, D dtor, B dtor.

Class Hierarchy Example



Access Specifier in Derivation

If a class is declared to be a base class for another class using the **public** access specifier, the **public** members of the base class are accessible as **public** members of the derived class and **protected** members of the base class are accessible as **protected** members of the derived class.

If a class is declared to be a base class for another class using the **protected** access specifier, the **public and protected** members of the base class are accessible as **protected** members of the derived class.

If a class is declared to be a base class for another class using the **private** access specifier, the **public and protected** members of the base class are accessible as **private** members of the derived class.

In all cases, **private** members of the base class are **not accessible** in the derived class.

Protected

The **protected** access specifier, when used in **class scope**, is useful for declaring class data members (but you should avoid it) and member functions that are accessible to the classes that derive from that class.

*(When used in **class inheritance list**, it means that the inheritance is "protected", so the public interface of the base class is not exposed by the derived class, but it becomes protected.)*

```
class C
{
public:
    int GetX() const {return m_x;}
protected:
    void SetX(const int i_x){m_x = i_x;}
private:
    int m_x;
};
```

Non-Public Inheritance

Avoid it.

Avoid private or protected inheritance… private inheritance should model the concept “is implemented in term of” that usually can be modeled with containment. **[Exceptional C++ Item 24]**

There are some notable exceptions:
- If you need to **override a virtual functions** of the Base class
- If you need access to the **protected interface** of the Base class
- few minor stuffs

In those cases you can (sometimes you must) implement the containment via nonpublic inheritance.

… see also:
- **[Effective C++ 3rd Item 39] [More Exceptional C++ Item 23]**
- http://stackoverflow.com/questions/656224/when-should-i-use-c-private-inheritance
- http://www.parashift.com/c++-faq-lite/private-inheritance.html

The lack of “final” keyword

In other languages you can say a class is **final**, i.e. not derivable.

In C++ you cannot (and that would have been useful!)

You can simulate the final behavior, as you can see in the **[code]**.

See also:
- http://www.codeguru.com/cpp/cpp/cpp_mfc/stl/article.php/c4143
- http://www2.research.att.com/~bs/bs_faq2.html#no-derivation

Polymorphism

The C++ language has several tools supporting **polymorphism**:

Function pointers for free function (global, namespace, class static): function pointer can take the address of a set of function with identical signature and invoke them.

Member Function Pointers for non static class member functions: similar to the Function Pointers, a little bit less used.

Virtual Functions (see you soon) in class hierarchy: a base class defines a virtual function, derived classes **override** it, with the same signature

Template Policy Class (see you later): sometimes called "**static polymorphism**", a compile-time tool for generate classes with given behavior that supports a specific function invocation Policy. Does not need to have the same signature, just a given invocation form.

Prefer **polymorphism** (and eventually RTTI) over manual type check and/or big switch-case over behaviors

Virtual Functions

The most used tool for implementing **polymorphism** is virtual function in class hierarchies.

When a base class defines a **virtual function** and a derived one defines the same function, with **the same signature** (name, modifiers, parameters, return value), we say that the derived class **overrides it**.

```
class A
{
    virtual void f();
};

class B: public A
{
    virtual void f(); // B overrides f. Here “virtual” is not necessary: if a method is
                      // specified “virtual” every overrides is virtual too
};
```

If f() in the base class **is not** specified virtual, B::f **hides** A::f for B objects.

Static member function cannot be declared virtual.

Access Rules for Virtual Function

The access rules for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it.

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

class D : public B
{
private:
    int f();
};

void f()
{
    D d;
    B* pb = &d;
    D* pd = &d;
    pb->f(); //OK: B::f() is public, D::f() is invoked
    pd->f(); //error: D::f() is private
}
```

Late Binding for virtual function

A **normal function call** is usually **statically bound**: assembly code generated from a function call is a “jump” to the address of the subroutine, known at compile time.

A **virtual function call** can be **dynamically bound**, so the address of the subroutine is not known at compile time but at runtime.

I said “can” because it depends **how** you access to the virtual function:
- via an **object**: no late binding, no polymorphism
- via pointer or reference: late binding, polymorphism

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

A a1;
B b1;
A& a2 = a1;
A& a3 = b1; // yes, you can assign a derived object to a reference to a base one
a1.f(); // calls A::f
b1.f(); // calls B::f
a2.f(); // calls A::f
a3.f(); // calls B::f
```

Polymorphism Carrier

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

Polymorphism cannot be handled by objects. When full objects come in play, no late binding can be produced.

To behave polymorphically you must access to your entities by either **Pointers** or **References**:

```
A a; B b;
A* a1 = new A;
A* a2 = new B;
A* a3 = &a;
A* a4 = &b;
A& a5 = b;
A& a6 = *a2;
A& a7 = *a3;
a2->f();
a5.f();

A a8 = b; // slicing
a8.f(); // calls A::f
```

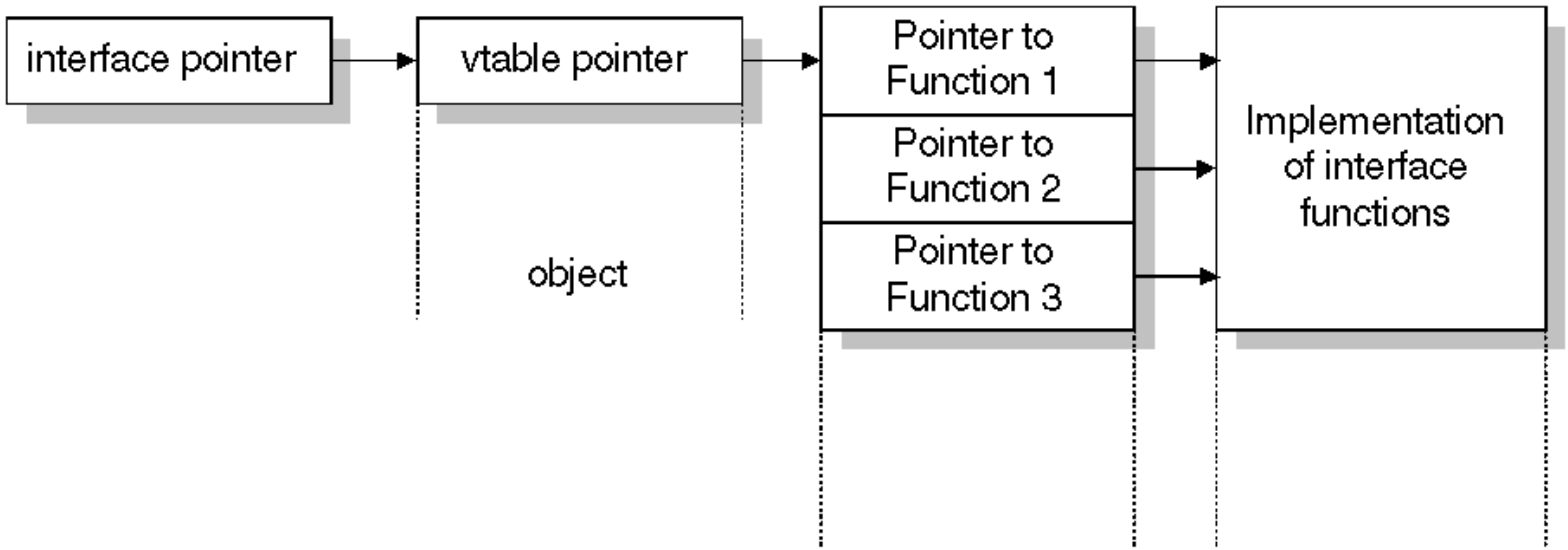
Late Binding: Hot is it implemented?

Each class has a (compile time built) runtime component named “Virtual table” or **VTable** that contains all addresses of virtual functions.

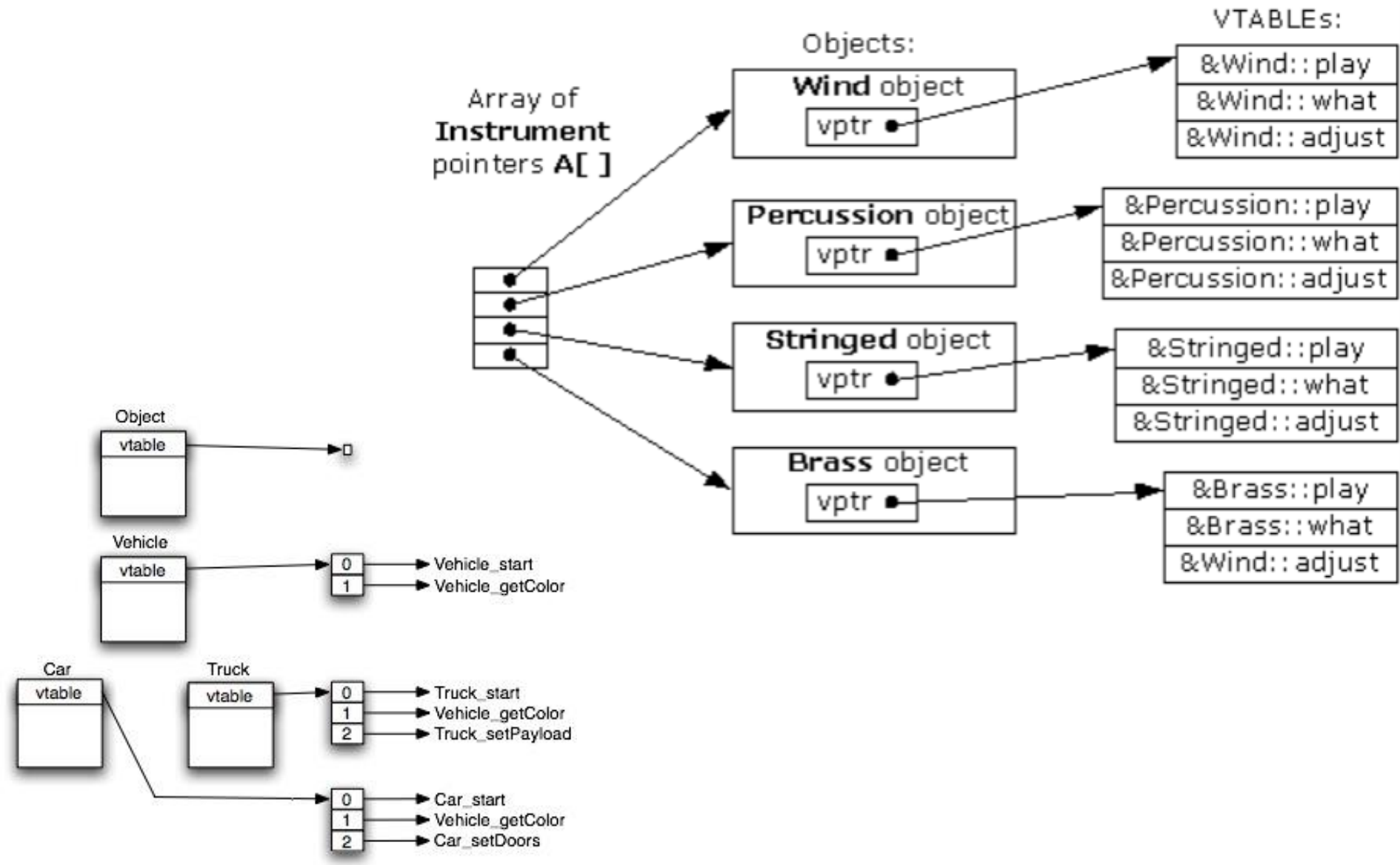
(maybe not all: if a function is defined virtual but never inherited, or never called, etc... The compiler is allowed to remove/optimize the virtual function call)

Each object of a class with polymorphic behavior (at least a virtual function, i.e. a class with a VTable) has **a pointer** to its actual type’s VTable. This pointer is usually called **vfptr** (virtual function pointer).

So objects of polymorphic classes has **size** increased by the space occupied by the vfptr (plus alignment)



VTables at work




About Virtual Functions

We've seen you can always qualify a function call to invoke the right version (avoid it), for example a->A::f();

You can also **call the base version** of a virtual function inside the overridden one:

```
void B::f()
{
    A::f(); // call the base version
    /*do other stuffs*/
}
```

Virtual **can be inline**, useful when you invoke them with qualification or through an object instead of pointer/reference

```
inline void A::f(){/*do some stuff*/}
A a;
B b;
A* a2 = &b;
a.f(); // inline
b.f(); // if B declares f inline, then inline.
a2->A::f(); // inline
```

Virtual Functions Binding in ctor&dtor

In **ctors** and **dtor** the virtual function binding is not “late”. Virtual functions are **statically bound** to the constructing class.

This is because the derived object **is not yet built** (or yet **destroyed**)

```
class B
{
    B()
    {
        f(); // always calls B::f
    }
    virtual void f(){cout << “asd”;}
};

class D: public B
{
    virtual void f(){cout  << “lol”;}
};

D d;
```

The lack of “super” keyword

Some languages has a special keyword “**super**” for detecting the direct base class of a given class.

C++ hasn’t it (and that would have been useful!)

```
class B: public A
{
    virtual void f()
    {
        //super::f(); // sorry, you cannot ☹
        A::f(); // ok, but what if someone introduce a class in hierarchy
                // between A and B?
        /*do more stuffs*/
    }
};
```

You can do nothing for this problem...

Virtual function good practices

(up to now...)

Avoid declaring **all virtual** in a **polymorphic class**: virtual functions have time costs due to VTable access.

Avoid declaring **a function virtual** in a **non polymorphic class**, vfptr has space cost and VTable access has time costs.

You should be clear if a class is designed to be polymorphic, in a hierarchy, or not!

Value classes usually are not designed to be polymorphic (in the virtual sense of the term)

Time to make a test! **[Test]** (from: **[Exceptional C++ item 21]**)

Abstract Classes

A virtual function can be declared **pure virtual**.

```
class B
{
    virtual void f() = 0; // “= 0” is the syntax for declaring pure virtual
};
```

A pure virtual function means that it is **defined** and **unimplemented**.
An object of class B **cannot be constructed**.
B is called **Abstract Class**.
Classes that can be instantiated is called **Concrete Classes**.

(In C++ you can declare a function virtual and provide a default implementation. Derived classes can use the default but they must call it explicitly. Base class is still abstract. **[More Exceptional C++ Item 27]***)*

```
class B
{
    virtual void f() = 0 {cout << “lol”;} // still pure virtual
};

void D::f(){ B::f(); } // assuming D derives from B, D uses the default impl. of f()
```

The lack of “interface” keyword

Some languages has a special keyword “**interface**” to declare a special case of “class” that has only pure virtual methods and no data.

C++ doesn’t (and that would have been useful!)

```
interface D // eventually extends some other interface
{  // default public: no need to declare private stuffs (maybe protected)
    void f(); // no need to declare virtual, it’s all virtual!
    int g(const char*, float&);
};
```

No, you **can’t**. Interfaces are great tools. Really great. It’s one of the greatest “**missing feature**” of the C++ language. You can **emulate** it

```
/* hey, other developers of my team, this is an *interface*, I can ensure it! I’ll
add some conventional things to let you search it in the code, like a trailing
‘Interface’ postfix, or a big ‘I’ in front of the class!*/
class IC_Interface // please, maintainers, do not add derivation to non-interface
{
public: // yeah, I need to specify it... Please do not remove
    virtual void f() = 0; // do not implement, please
    virtual int g(const char*, float&) = 0; // do not implement, please
    // please, do not add data, or non pure virtual function. Please.
}; // oh, shit! I forgot the virtual destructor...
```


Virtual Destruction

delete operator cannot be virtual (we'll see it must be "static" if you decide to override it for your class).

The following code will **slice** the object, deleting only the A subobject.

```
class A{int x;}; // sizeof 4;
class B: public A{int y;}; // sizeof 8
A* a = new B; // we've not yet seen the new operator...
              // It returns a pointer to the object it creates: a B*
              // we know we can assign a B* to an A*
delete a; // deleted first 4 bytes... Leaked the others 4
```

The solution is to declare the **dtor virtual**.
Always declare the dtor virtual for your **base classes**
[Effective C++ Item 14] [Exceptional C++ Item 41]

```
class A
{
   int x;
   virtual ~A(){} // you can declare it pure virtual too
};
```


Object Slicing

We've seen how the lack of a virtual dtor can slice (and leak) an object in a hierarchy if deleted through pointers to base classes.

It is a common case: you'll have **vector** or **array** of **base class pointers** that actually points to derived objects. When you'll delete them you slice. **Always declare dtor virtual for base classes**. (derived will inherit it)

It's not the only slicing point! You'll **slice objects** when you copy or delete them **misinterpreting** the actual type.

```
void f(B b){} // not taken by reference...
D d; // D derives from B
F(d); // sliced! When B dtor will execute at the end of f bad things could happen
```

Ok, easily solved: **never take a parameter by value** if it is in a class hierarchy (in general avoid pass by value, ok?)

See also
http://bytes.com/topic/c/answers/163565-slicing-problem


Object Slicing in copy operations

We've solved two easy slicing problems with virtual dtor and avoiding pass by value of classes in hierarchy.

Now look at the **copy ctor** of class with pointers data members, with ownership, to a class with **polymorphic** behavior (yeah, the test):

```
class C
{
    T* x; // T is a class in a hierarchy. T has virtual functions.
          // x owns the memory pointed. We don't want to share the mem
    C(const C& i_that)
    {
        if(i_that.x)
            x = new T(i_that.x); // deep copy of (*x)
        else
            x = 0;
    }
};
```

Does it work? Why? Why not?

Object Slicing in copy operations (2)

...It doesn't work!

T is a type in hierarchy. x **actual type is not known in C copy ctor**.

T could be an **abstract class**!

The **new operator is not virtual**, it cannot be.
(spoiler: its mechanic is to allocate memory and to invoke the T ctor. Ctors cannot be virtual. Think about it... What it would mean?)

You have to create an object of the same type of i_that.x, in this case you have to get a "clone" of i_that.x.

We must introduce **virtual construction** and **virtual copy [More Effective C++ Item 25]**

Virtual Construction & Copy

Virtual construction: I need to create an instance of an object of unknown type: I've a pointer to a class in a hierarchy and I want to create an object of the same type.

Virtual copy: I need to create a copy of an object of unknown type, as before.

Constructors cannot be virtual!
Try to define operator=() virtual!

(see: http://icu-project.org/docs/papers/cpp_report/the_anatomy_of_the_assignment_operator.html and: http://icu-project.org/docs/papers/cpp_report/the_assignment_operator_revisited.html)

We need a mechanism to provide these services:

Cloning and **Create** (Factory)

Clone & Create

```
class Shape {
public:
  virtual ~Shape() { }                    // A virtual destructor
  virtual void draw() = 0;                // A pure virtual function
  virtual void move() = 0;
  ...
  virtual Shape* clone()  const = 0;      // Uses the copy constructor
  virtual Shape* create() const = 0;      // Uses the default constructor
};

class Circle : public Shape {
public:
  Circle* clone()  const;    // Covariant Return Types; see below
  Circle* create() const;    // Covariant Return Types; see below
  ...
};

Circle* Circle::clone()  const { return new Circle(*this); }
Circle* Circle::create() const { return new Circle();      }
```

```
void userCode(Shape& s)
{
  Shape* s2 = s.clone();
  Shape* s3 = s.create();
  ...
  delete s2;     // You need a virtual destructor here
  delete s3;
}
```

Construction & Copy good practices

When working with value classes or not polymorphic classes, copy is easy. No bad things could happen.

When working with polymorphic types, you have to decide if:

Let classes itself handle construction & copy, providing create & clone member functions and instructing clients and maintainers to adopt the strategy.

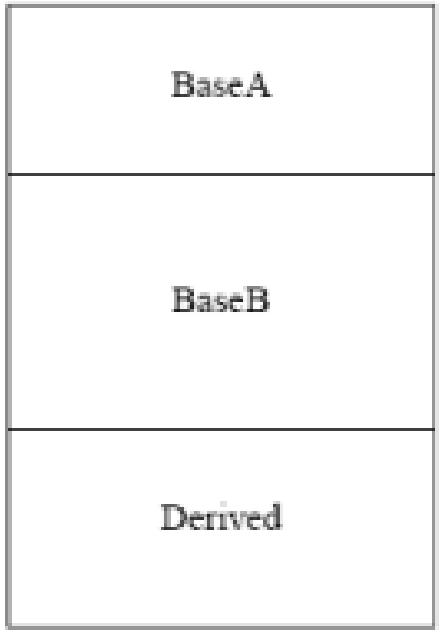
Provide a **factory class** (see factory pattern **[GOF – factory method]**) that handles creations and copy, define all ctors and copy assignment private (and the factory will be friend)

Disable copy for everyone, defining copy ctor and copy assignment private and undefined.

[Test]

Multiple inheritance

```
class A{};
class B{};
class C: public A, public B{}; // multiple inheritance
```



Memory layout of the class is something like this:

The subobjects doesn't start at the same address,
While in normal inheritance they will.

Here it comes in play the importance of right casting.

See **[code]** for casts, memory layout for MI and for deep inheritance.

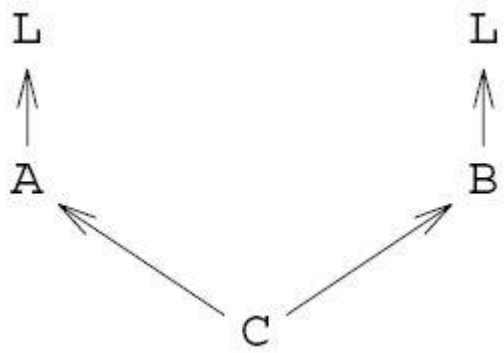
Avoid downcasting **[Effective C++ item 39]**

See also **[Exceptional C++ item 44]**

Remember that these stuffs are **impl.dep**. Gcc and MSVC implements vfptr differently, but problems are similar.

Virtual Class Derivation

```
class L { public: int next; /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { void f(); /* ... */ };
```

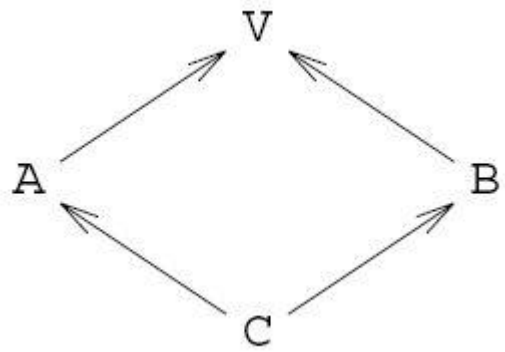



```
void C::f() { A::next = B::next; } // well-formed
…
void C::f() { next = 0; } // error: ambiguous
```

This is called Siamese twins problem **[More Exceptional C++ Item 26]**

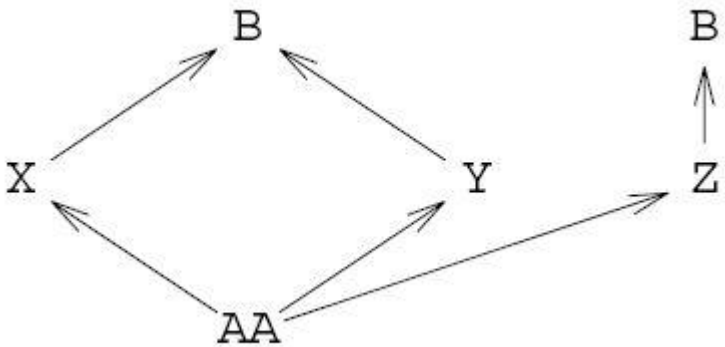
Virtual Class Derivation

```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```



Virtual Class Derivation

```
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```



Multiple inheritance Good Practices

AVOID IT

...unless you are deriving from no more than **ONE** non interface class.

(we've not seen all the problems if you have to override the same virtual member functions in more than one base class...)

See also:
http://www.newlc.com/en/Multiple-Inheritance.html


static_cast

The result of the expression static_cast<T>(expr) is the result of converting the expression expr to type T.

If T is a reference type, the result is an lvalue;

static_cast must not cast away const-ness

You can use it for documentation, where an implicit conversion will take place, or when it's mandatory, like you are converting a void* to another pointer:

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**. This ensures that at least the classes are compatible if the proper object is converted, but **no safety check** is performed during runtime to check if the object being converted is in fact a full object of the destination type.

reinterpret_cast

The result of the expression reinterpret_cast<T>(v) is the result of converting the expression v to type T.

If T is a reference type, the result is an lvalue.

reinterpret_cast must not cast away const-ness

The mapping performed by reinterpret_cast is impl.dep

reinterpret_cast converts **any pointer type to any other pointer type**, even of unrelated classes. The operation result is a simple **binary copy of the value from one pointer to the other**. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast **pointers to or from integer types**. The format in which this integer value represents a pointer is impl.dep. The **only guarantee** is that a pointer cast to an integer type **large enough** to fully contain it, is granted to be able to be **cast back** to a valid pointer.


const_cast

Casting away const can be necessary to call const-incorrect APIs.

Once you go const, you (should) never go back. **If you cast away const on an object whose original definition really did define it to be const, all bets are off and you are in undefined behavior land.**

For example, compilers can (and do) put constant data into write-protected RAM pages. Casting away const from such a truly const object is a punishable lie, and often manifests as a memory fault.

C++ has one implicit const_cast, the "conversion of death" from a string literal to char*:

```
char* weird = "Trick or treat?";
```

The compiler performs a silent const_cast from const char[16] to char*. This was allowed for compatibility with C APIs, but it's a hole in C++'s type system. String literals are ROM-able, and trying to modify the string is liable to cause a memory fault.

dynamic_cast

The result of the expression dynamic_cast<T>(expr) is the result of converting the expression expr to type T.

dynamic_cast can be used **only with pointers and references** to polymorphic objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, dynamic_cast is always successful when we cast a class to one of its base classes.

If it fails on a pointer, it return 0. if it fails on a reference, it throws an exception.

dynamic_cast requires the **Run-Time Type Information** (RTTI) to keep track of dynamic types.

Some companies (like **Milestone**) choose to **not support RTTI** in non-debug build.

checked_cast

```
template <typename DestTypePtr, typename SourceType>
inline DestTypePtr checked_cast(SourceType* const i_srcPtr)
{
    /* If this fails to compile, you are trying to cast UNRELATED
    TYPES or ONE OF THE TYPES IS CONST (you need an
    additional const_cast for that), so your code is wrong. Maybe the
    types used to belong to the same hierarchy but not anymore.

    On run-time error: the source object can NOT be down-casted to
    the destination type because it is NOT actually of that type. DO
    NOT CONTINUE! the code behaviour is undefined (probable
    crash).

    Note: NULL pointers can be safely cast to any type, so they must
    verify the assertion. */

    assert(i_srcPtr==NULL || dynamic_cast<DestTypePtr>(i_srcPtr));
    return static_cast<DestTypePtr>(i_srcPtr);
}
```


cast guidelines

Avoid C cast, prefer C++ casts **[More Effective C++ Item 2]**

Avoid const_cast (but you'll see a lot of them in code) whenever possible. At least document it (you'll see a few of them)

Avoid reinterpret_cast, unless you are working in very low-level stuff.

Prefer static_cast when suitable.

Use custom **checked_cast** when you need dynamic_cast (usually you don't have RTTI enabled in non-debug builds)

Read articles **[Exceptional C++ item 44]**

See the sample **[Code]**

Object Oriented vs Object Driven

Avoid inheritance abuses **[Exceptional C++ item 24]**

Know when to design a hierarchy and when to use encapsulation
[Exceptional C++ item 22]

Use MI judiciously **[Effective C++ item 43]**

When to use MI?
[More Exceptional C++ Item 24] [Modern C++ Design chapter 1]

Classes should be either abstract or not derivable.
- never derive from a concrete class
- only the leaves should be concrete

[More Effective C++ Item 33]

Functions should be either public or virtual (NVI, Template pattern)
[Exceptional C++ item 24]

Never “override” (hide) a non virtual function. **[Effective C++ item 37]**

Respect the Open/close principles

Inheritance VS Containment

Inheritance should only model “is a” (LSP) **[Effective C++ item 35]**

use containment for “has a” or “has many”. **[Effective C++ item 40]**

Implement “has a” through private inheritance only if you need to override a virtual method or use protected interface...

If multiple classes share **common data but not behavior**, then create a common object that those classes can contain.

If multiple classes share **common behavior** but not data, then derive them from a common base class that defines the common routines.

If multiple classes share **common data and behavior**, then inherit from a common base class that defines the common data and routines.

Inherit when you want the base class to control your interface; contain when you want to control your interface.

Class Nomenclature

An **abstract class** is a class that cannot be instantiated, a class that defines or inherits at least a pure virtual function. A **concrete class** is a class that can be instantiated.

In a relationship between two classes in a hierarchy we call C1 **base class** of C2 if C2 inherits from (or derives) C1. In such that case we call C2 **derived class**.

An abstract class can be base class for other classes, and can be derived class from other classes. The same is possible for concrete classes, the two traits are independents.

An **interface class** is an abstract class with no non-static data members, and all pure virtual non-static member functions.

Interface is not a C++ enforced concept (unlike java), and I think that's a big hole. It's very important to handle pure interface, and to enforce developer to not violate a interface promise.

Class Nomenclature

A **data aggregator class** is a collection of objects, with a few optional methods, without data protection and data hiding. What we call "**a struct**".

A **data abstraction** class is a user-defined type, with interface and implementation, designed to work in "has a" relationships, in generic algorithms, in containers.

A **value class** is a data abstraction class that implements value behaviours: copy, operators overloading, etc...

A **manager class**, sometimes a singleton, handles resources, objects, usually in a dynamic fashion. Usually it's non-copyable.

A **factory class** is designed to create and destroy objects of other classes.

A **façade class** is a module interface to avoid too many connections between modules.


Class Nomenclature

A **functor class** is a class that defines the function call operator "operator()". It is used mainly as function objects in generic algorithms, it is usually stateless.

A **behavioural class** is a class that implements specific algorithms. Usually it is or stateless or non copyable.

A **class hierarchy** is a set of classes with a derivation tree (that can be complex and deep) that usually are used polymorphically.

A **generic class** is a class thought for generic programming. Could also be a policy collector class (in policy based design), a container, a smart pointer class, traits class, etc

... It's impossible to give each class category a name, but 90-95% of classes in common projects are of one of those types.


Reasons to create a class

Basic Reasons:
Model real world objects or Abstract Objects
Reduce and Isolate Complexity (hide, encapsulate)
Reuse code
Limit effect of changes
Provide a central point of control

Be suspicious of:
Derived classes with only one instance
Base classes with only one derived class
Overriden and empty member functions
Deep inheritance tree (maximum depth should be 5 ~ 7, maximum derived classes for a single base class should be no more than 10)

Links

Web
(too many...)

Books
(same as Classes I)