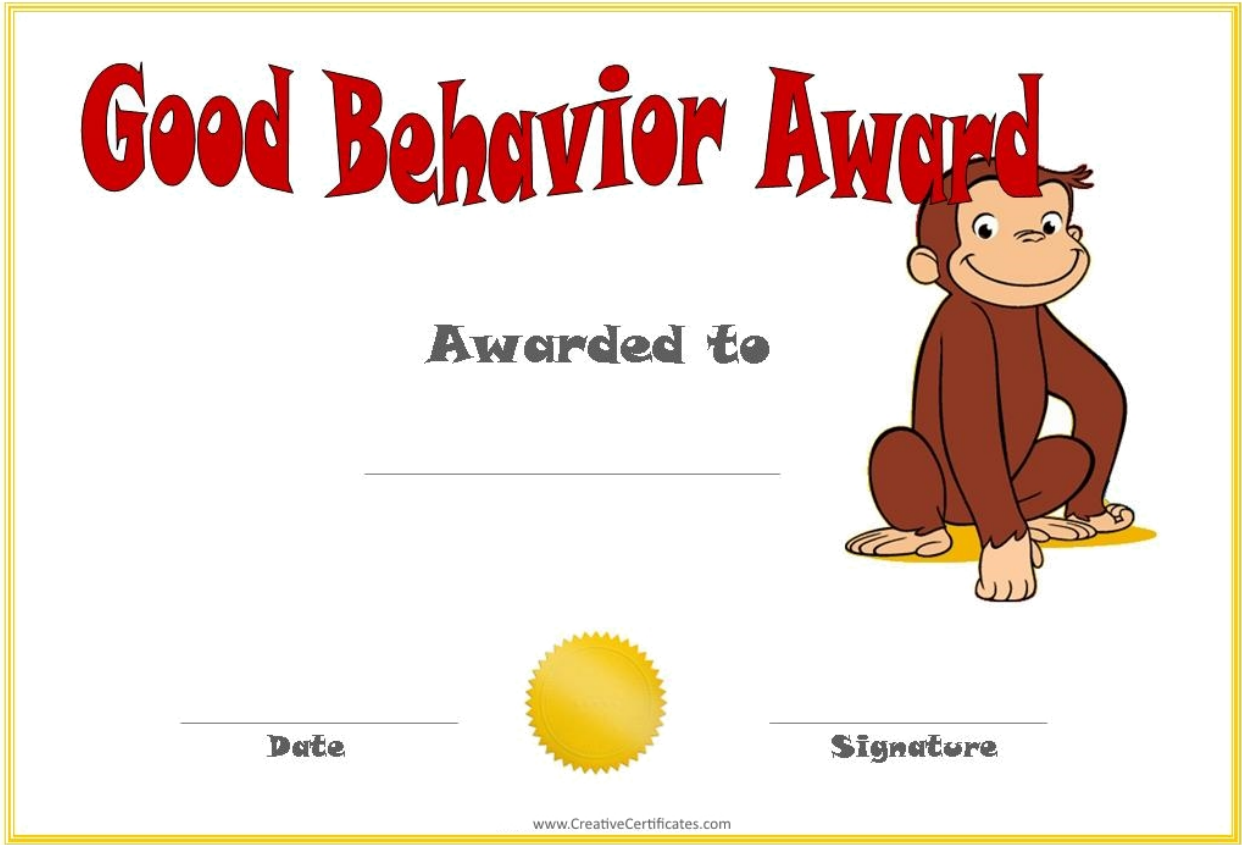



# C++: Well-Behaved Objects

Yngvi Björnsson

# C++ Objects

See: EC [#5]

- Have (typically) a:
  - **Constructor**
  - **Copy-Constructor**
  - **Assignment Operator**
  - **Destructor**
- If not provided by the programmer, then
  - Automatically generated (if needed and sensible)
    - as **public inline** methods
    - **copy-constr** and **assignment** do field-by-field copy.
      - Of non-static data members

For example, assignment operator will not be automatically generated if any **const** or **reference** data members in the class.
Why?

# Implicitly Created Methods

```
class Point {
public:
  int x_;
  int y_;
};
```

```
class Point {  // Implicitly created methods.
public:
  Point() { // Constructor
  }

  Point(const Point& rhs) : // Copy-constructor
    x_(rhs.x_), y_(rhs.y_) {
  }

  Point& operator=(const Point& rhs) { // Assign op
    if ( this != &rhs ) {
      x_ = rhs.x_;
      y_ = rhs.y_;
    }
    return *this:
  }

 ~Point() { // Destructor
  }

  int x_;
  int y_;
};
```

# Constructor

```cpp
class Point {
public:
  // Constructor
  Point(int x, int y)
    : x_(x), y_(y) {
  }
  // No implicit default constr.
  // . . .
};
```

```cpp
// Could write constructor here
// as below, but inferior style
{
  // Constructor
  Point(int x, int y) {
    x_ = x;
    y_ = y;
  }
};
```

- If **no** constructor is provided, then **default (no argument)** constructor created.
- Can have **many overloaded** constructors
- However, if **any** constructor is provided, then no default constructor automatically created.

# Copy Constructor

```
class Point {
public:
  // . . .

  // Copy Constructor
  Point(const Point& rhs) :
    x_(rhs.x_), y_(rhs.y_) {
  }

  // . . .
};
```

- A constructor where parameter is of same type as the object itself.
- Newly created object a field-by-field copy of the old object.
  - Default implementation
- Remember to copy all members

See: EC [#5]

For example, when one adds an extra data member later. Also, when inheriting, see later (and EC[#5]).

# Assignment Operator

```
class Point {
public:
  Point& operator=(const Point& rhs)
  {
    if ( this != &rhs ) {
      x_ = rhs.x_;
      y_ = rhs.y_;
    }
    return *this:
  }

  // . . .
};
```

- Assign a value to an already existing object.
- May have to explicitly delete existing values.
  - Not needed here.
- Must guard against self-assignment
  - **p1 = p1;** See: EC [#11]
- Must return the object itself (by ref.)
  - **p1 = p2 = p3;** See: EC [#10]

# Why?

- In C++
  - Give classes, like other types (int, char, …) , **value semantics** (i.e., assignment copies value)
    - *copy constructor* and *assignment operator* thus needed
    - can use *reference semantics* by using pointers, if wanted.
  - Programmers responsible for **memory management**
    - no automatic garbage collection
    - *destructor* thus needed
- Unlike Java (and many other OO languages):
  - which use **reference semantics** only

For details see e.g. C++ FAQ [31]

# Example

Are you familiar with C++ new / delete / delete [] ?

**C++**

```
{
  // Basic type.
  int i1 = 0;
  int i2 = i1, i3=i1;
  i1 = i2;

  // Class type.
  Point p1(0, 0);
  Point p2(p1),
        p3=p1;
  p1 = p2;

  // Reference to obj
  Point& rp = p1;
}
```

**C++ (more Java-like style)**

```
{
  // Basic type.
  int i1 = 0;
  int i2 = i1, i3=i1;
  i1 = i2;

  // Using ptrs.
  Point *p1 = new Point(0, 0);
  Point *p2 = new Point(*p1),
        *p3 = new Point(*p1);
  p1=p2->copy(); //User def. method

  // Pointer to object.
  Point *rp = p1;

  delete p3;
  delete p2;
  delete p1;
}
```

# What gets Called ?

```
{
  // Basic type.
  int i1 = 0;
  int i2 = i1, i3=i1;
  i1 = i2;

  // Class type.
  Point p1(0, 0);
  Point p2(p1),  p3=p1;
  p1 = p2;

  // call-by-value
  foo_v( p1 );
  // call-by-reference
  foo_r( p1 );
}
```

constructor

Copy -constructor

Copy -constructor

Assignment operator

Copy -constructor

Destructors (p3,p2,p1)

# Disable Compiler-Generated Functions

See: EC [#6] and Google C++ SG

- We might not want our object to be copied.
  - Cannot simply skip implementing methods, because then default created.
  - Instead, must create copy-const and assignment-op, but make them private.
  - Linker error if we try to copy/assign somewhere in code.

```
// Some class we do not want to be copied.
class Foo {
public:
    // ...
private:
    Foo( const Foo& );
    Foo& operator=( const Foo& );
    // ...
};
```

# User Provided Copy Methods

- Compiler generated methods most often sufficient to conserve value semantic and memory integrity.
- However, when using data-member that are pointers or references then it breaks badly.
- Then the user must provide:
  - copy-constructor, assignment operator, destructor
  - if you need one of the above you (most likely) need them all.

# Example: MyStr Class (using pointers and new)

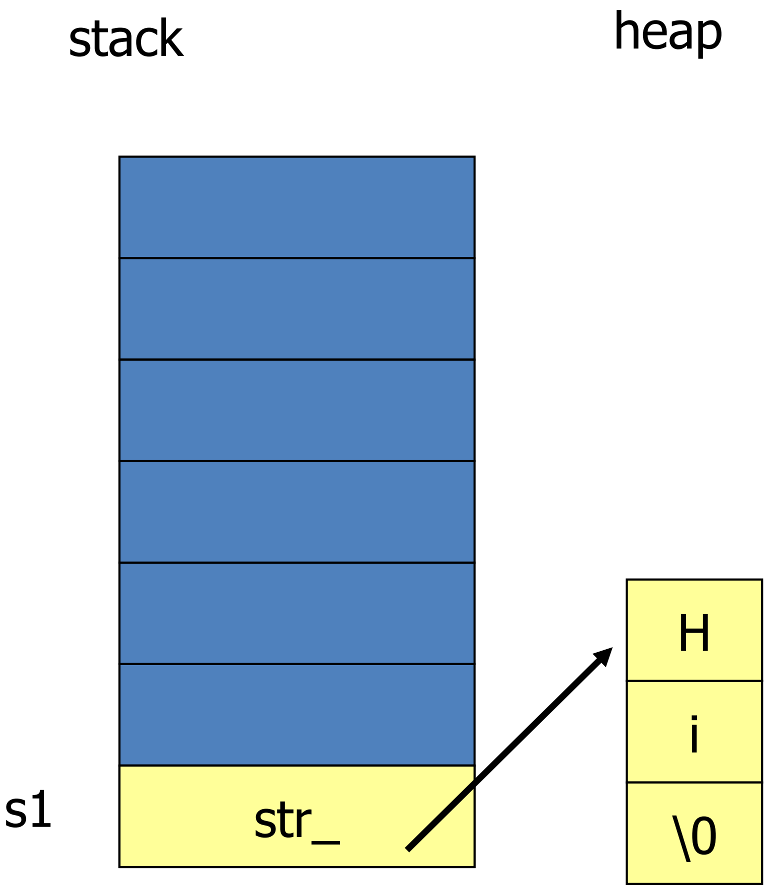
```
#include <cstring>

class MyStr
{
public:
   MyStr( const char *str );

    // . . .

private:
   char *str_; // Lookout, a pointer!
};
```

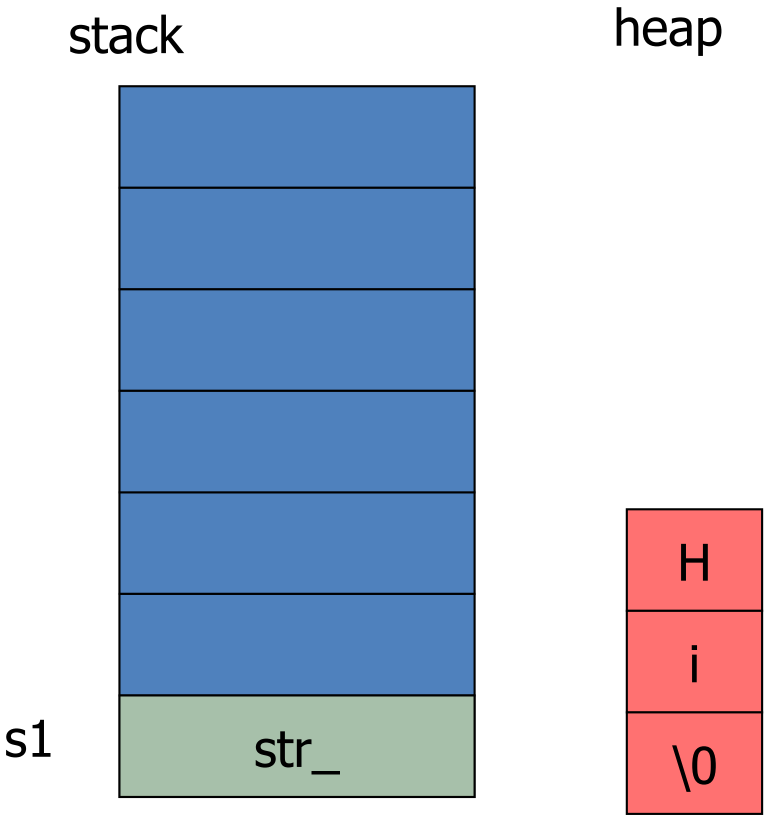
# MyStr: Constructor




- Constructor called:
  - function()
    ```
    {
      MyStr s1(“Hi”);
    }
    ```
- In our example constructor allocates memory with **new**:
  - MyStr::MyStr(const char *str)
    ```
    {
      str_ = new char[strlen(str)+1];
      strcpy(str_, str);
    }
    ```

Are you familiar with **c-style** strings?
(‘\0’ terminated char arrays)

# MyStr: Pitfall #1

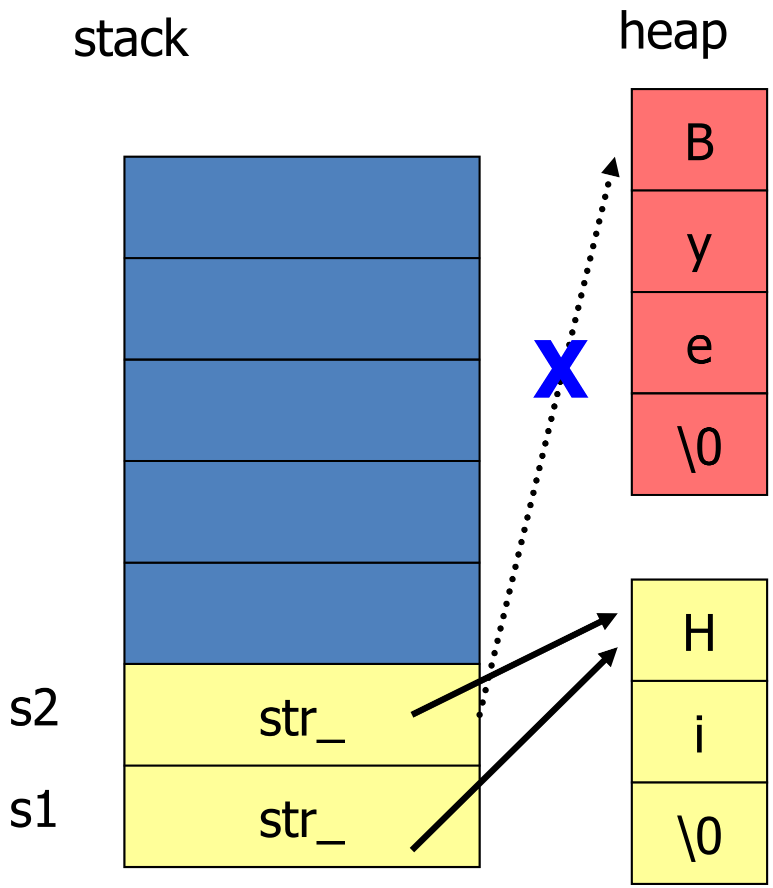



- Constructor called:
  - ```
    function()
    {
      MyStr s1("Hi");
      // object destructed when s1 goes
      // out of scope.
    }
    ```

- **Must** provide own destructor to return dynamic memory when object's lifetime ends.
  - ```
    MyStr::~MyStr()
    {
      delete [] str_;
    }
    ```

Unless we do so, s1 removed from stack, but "Hi" still left in heap ➔ MEMORY LEAK

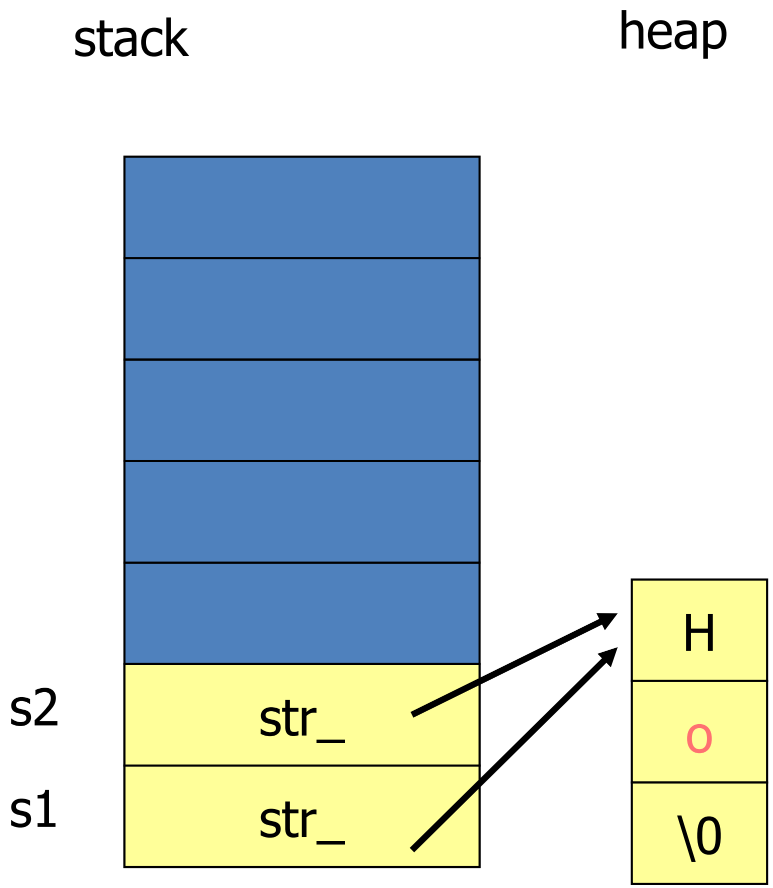
# MyStr: Pitfall #2




```
{
  MyStr s1("Hi"), s2("Bye");
  s2 = s1;
  // … later modify s1, what with s2?
}
```

- **Not** what we want:
  - Want to make a copy of all the dynamically allocated items too!
  - Default just a member-copy ☹
  - Need to provide our own assignment operator to do the job correctly.

# MyStr: Pitfall #3




- Call-by-value parameters:
  - ```
    void foo(MyStr s2)
    {
      s2[1] = 'o';
    }

    int main()
    {
     MyStr s1("Hi");
     foo(s1);
     // s1 is now: Ho
    }
    ```
- Need to provide own copy-constructor.
    ```

# Classes with Pointers (4 commandments)

- Thy shall provide your own:
  - Constructor(s)
    - Dynamically allocates memory
  - Copy constructor
    - For call-by-value parameter passing (and object initialization)
  - Overloaded assignment operator
    - For assigning (copying) one object to another
  - Destructor
    - Returns back dynamically allocated memory
- Always provide all four if using pointers/references
  - (even though you think they might not be necessary)

# Example: MyStr Class (using pointers and new)

```
#include <cstring>

class MyStr
{
public:
   MyStr( const char *str );

   // . . .

private:
   char *str_; // Lookout, a pointer!
};
```

# MyStr: Constructor(s)

- ```
   MyStr( const char *str )
  {
       str_ = new char[strlen(str)+1];
       strcpy( str_, str );
   }
  ```
- Purpose:
  - Initializing member variables
  - Dynamically allocate memory
- When called?
  - Object declaration / allocation (new)
  ```

# MyStr: Copy Constructor

- ```
  MyStr( const MyStr& rhs )
  {
      str_ = new char[strlen(rhs.str_)+1];
      strcpy( str_, rhs.str_ );
  }
  ```
- Like other constructors, except
  - expecting as an argument an object of same class
- When called:
  - a new object is created that is initialized with an object of the same class, e.g.
    - Objects passed by call-by-value
    - Object declaration/allocation if initializing object with an object of the same class (instead of “regular” constructor).
  ```

# MyStr: Assignment Operator

- ```
  MyStr& operator=( const MyStr& rhs ) {
    if ( this != &rhs ) {
      char *tmp = new char[rhs.size()+1];
      strcpy( tmp, rhs.str_ );
      delete [] str_;
      str_ = tmp;
    }
    return *this;
  }
  ```

Why use the **tmp** variable?

Exception safety!

Must free resources of the object being copies into.

- Purpose:
  - Same as copy constructor, except needs to "clean-up" object being assigned the value!
- When called?
  - Assignment operator (<u>except</u> initialization assignment)
  ```

# MyStr: Destructor

- ```
  ~MyStr( )
  {
    delete [] str_;
  }
  ```

Note [ ]

- Purpose:
  - Return back dynamically allocated memory
- When called:
  - object lifetime's end
    - e.g. for automatic local variables when exiting a function (block)
  - delete
  ```

# Summary: When called?

- Constructor
  - Declare objects
    - MyStr s1;
      MyStr s2("Hi"); // Same as: MyStr s2 = "Hi";
  - Creating objects with **new**
    - MyStr *p1 = new MyStr("Hi");
- Copy constructor
  - For object call-by-value parameters/return
    - void A(MyStr s); // pass-by-value
    - MyStr B(…); // return-by-value
  - Instead of regular constructor, if initializing object with an object of the same class
    - MyStr s2(s1); // Same as: MyStr s2 = s1;
    - MyStr *p2 = new MyStr(s1);

# Summary: When called?

- Assignment Operator
  - Whenever assignment operator used
    - s2 = s1;
    - except, in object initialization, e.g.
      - MyStr s2 = s1;
      - Copy constructor called instead
- Destructor
  - Life-time of object ends
    - Return from a block/function
  - Deleting a dynamically allocated object
    - **delete** p1;

# Const (EC [#3])

- **const** indicates that value cannot be changed
  - Can be confusing when used with pointers
    - Is the pointer or the value pointed to constant?

```
char greeting[] = "Hello";

char *p1 = greeting;         // non-const pointer, non-const data
const char *p2 = greeting;   // non-const pointer, const data
char * const p3 = greeting; // const pointer, non-const data
const char * const p4 = greeting; // const pointer, const data

char const *p5 = greeting;   // What do you think?
```

# Destructors and Exceptions (EC [#8])

- Do not allow destructors to emit exceptions.
  - This can lead to more than one exception being active at the same time (which may lead to and undefined behavior in C++)

```
~Object() {
   try {
      fooThatCanThrowException();
   }
   catch (  ) {
      // log error message
   }
}
```

# Inheritance

- C++ can inherit
  - public
  - protected
  - private

```
class Derived : public Base
{
};
```

- In C++ methods can be either (user specifies)
  - non-virtual
  - virtual

```
class Dog {
public:
    void bark( ) { ... }
    virtual void barkMore( ) {}
        virtual ~Dog() {}
};
```

```
class Dog {
public:
    void bark( )                 { cout << "Dog woff\n"; }
    virtual void barkMore( ) { cout << "Dog barks more\n"; }
    virtual ~Dog() {}
};
class Poodle : public Dog {
public:
    void bark( )      { cout << "Poodle woff\n"; }
    void barkMore( ) { cout << "Poodle barks more\n"; }
    virtual ~Poodle() {}
};

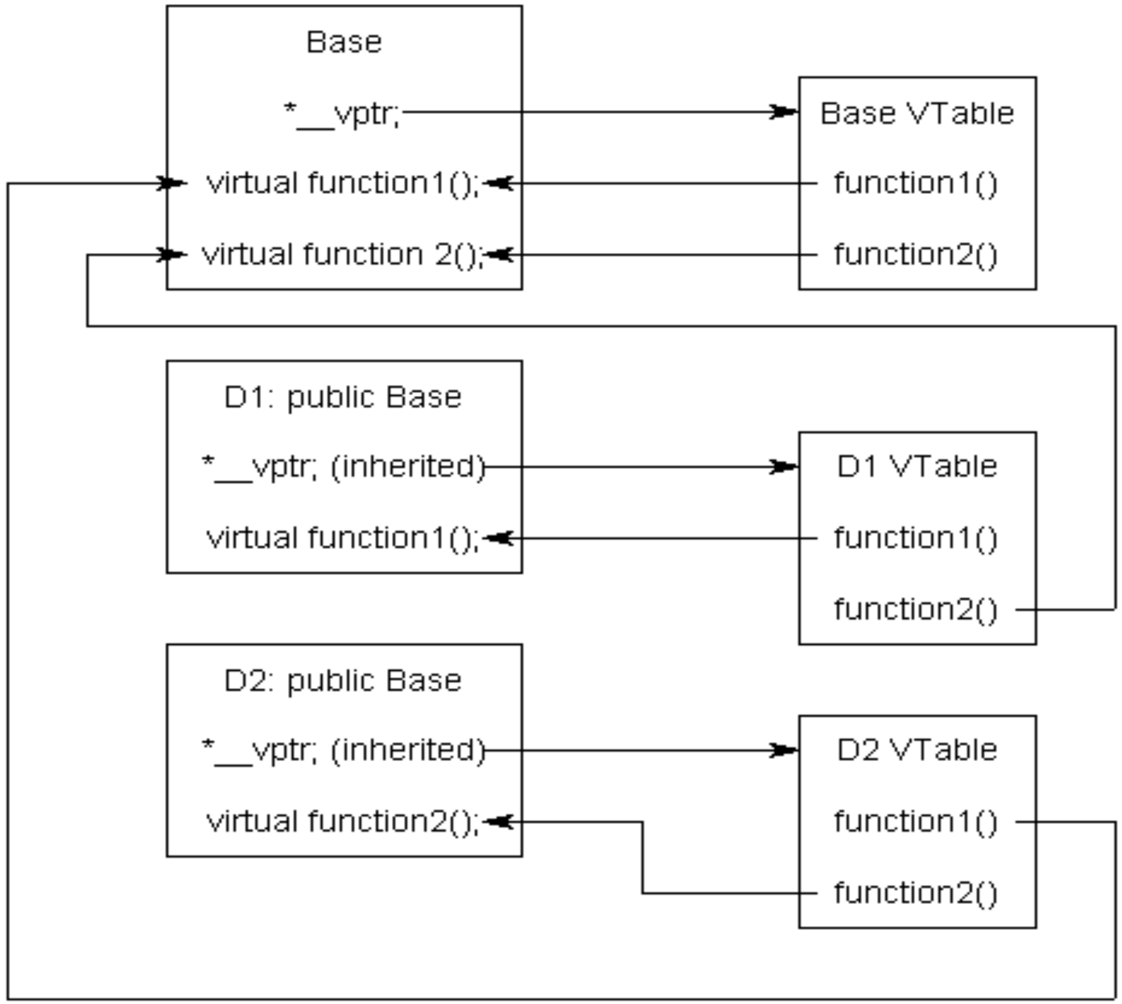
int main() {
    Dog *dog = new Dog();
    Poodle *poodle = new Poodle();
    Dog *any = dog;
    any->bark(); any->barkMore();
    any = poodle;
    any->bark(); any->barkMore();
    return 0;
}
```

If base method is virtual then derived method always virtual too (even though **virtual** not specified):

# Static vs. Dynamic Binding

- Call: dog.barkMore();
  - Where dog declared as an object of type Dog;
  - Compiler implements static binding, i.e. always calls barkMore method of class Dog.
- Call: dog->barkMore();
  - Where dog declared as a pointer to object of type Dog or Poodle;
  - Compiler implements dynamic binding, i.e. determines appropriate method call at run-time.

# Implementation



# Virtual Destructors (EC [#7])

```
class Base {
public:
    Base(  );
    virtual ~Base();
protected:
};

class Derived :
          public Base {
    Derived(  );
    virtual ~Derived();
public:
}
```

- Polymorphic base classes **should** declare virtual destructor
  - In particular, if any virtual function in the class then destructor virtual should be too.
- Classes not intended to be inherited need not have virtual destructor.

# Virtual Destructors Demo

- Particular important to get  destructors right
  - Difficult to catch errors once made.

# Derived Assignment Operators (take #1)

```
class Base {
public:
    Base( int x ) : x_(x) { }
protected:
    int x_;
};

class Derived : public Base {
public:
    Derived( int x, int y ) : Base(x), y_(y) { }

    Derived& operator=( const Derived& rhs ) {
      if ( this != &rhs ) {
            y_ = rhs.y_;
      }
      return *this;
    }
    void disp() { cout << x_ << ' ' << y_ << '\n'; }
private:
    int y_;
};
```

```
int main() {
  Derived d(1,3),e(2,4);
  d.disp();
  e = d;
  e.disp();
}
```

```
1 3
2 3
```

Note call to base destructor, needed if parameters required, **including in the copy constructor**. How about **assignment op**?

# Derived Assignment Operator

```
Derived& operator=( const Derived& rhs ) {
   if ( this != &rhs ) {
      Base::operator=( rhs );
      y_ = rhs.y_;
   }
   return *this;
}
```

EC [#12]

13
13

# Virtual Functions in C/D (EC [#9])

- Never call virtual functions during construction or destructions.
  - Does not do what you would expect it to.

```
class Transaction {
public:
    Transaction();     // calls log transaction
    virtual void logTransaction();
}

class BuyTransaction : public Transaction {
public:
    BuyTransaction();
    virtual void logTransaction();
}

BuyTransaction b;
```

BuyTransaction constr calls Transaction constructor, but in there Transaction::logTrans() is called. Not what we intended.

# Abstract Classes

- C++ does not have an explicit *Interface* declaration (like e.g., Java)
- Interfaces are declared in terms of abstract classes.
- Classes are made abstract by declaring a pure virtual function `( = 0 )`.

```
class Dog {
public:
    . . .
    virtual void barkMore( ) = 0;
     . . .
};
Dog dog;  // Wrong, cannot be instantiated.
```

Some developers like to name abstract base classes using with I prefix, e.g. IDog

# Summary

- Objects
  - Should behave as the other data-types
    - (int, …, structures)
  - OK to use default constructor/destructor/copy/assignment if no pointer/reference member variables
    - Still useful to use constructor(s) to initialize variables
  - Otherwise, need to provide
    - Constructor(s)
    - Copy constructor
    - Assignment operator
    - Destructor
  - Inheritance
    - Virtual vs. Non-virtual
    - Abstract vs. Non-abstract

# Questions?