Hiding and Statics

Hiding

- From Stroustrop: "*In C++, there is no overloading across scopes - derived class scopes are not an exception to this general rule*."
  - Therefore, C++ says that if a derived class overrides a function *foo*, the functions named *foo* in the base class are *hidden*.
  - Functions with the same *name* (parameters *do not* have to match) in the base class will not be visible.
- Let’s look at an example to make this clear.

class Derived extends class Base (See Example1 code)

```
Base::Base( ) { }
Base::~Base( ) { }
Virtual void Base::f(double x) {
  std::cout << "Base: " << x << std::endl;
}


Derived::Derived( ) { }
Derived::~Derived( ) { }
virtual void Derived::f(char x) {
  std::cout << "Derived: " << x << std::endl;
}
```

```
int main() {
  Derived* d = new Derived();
  Base* b = d;
  b->f(65.3); // okay: passes 65.3 to f(double x)
  d->f(65.3); // converts 65.3 to a char ('A' if ASCII)
              //  and passes it to f(char c); It does
              // NOT call f(double x)!!
}
```

```
Base: 65.3
Derived: A
```

If you are overriding base class functions

- then override all forms of the function if you need all forms
- You have extended the interface, and should provide valid implementations for the new interface.
- In the previous example, Derived should define both
  1. virtual void f(char c); // already defined
  2. virtual void f(double x); // not defined in Derived class,
     // but should be, or *using f* should
     // be used

Or, you can still call the base class

(See Example2 code for *using* example)

- Invoke Base::f(65.3) // gives 65.3
- Use a *using* declaration

```
class Derived : public Base {
public:
   using Base::f;  // this un-hides Base::f(double x).
                   // Now f(65.3) on  Derived object
                   // will call the Base class f(double x);
   void f(char c);
};
```

Why does C++ have hiding?

- When determining what functions named foo are visible, you only have to go up the inheritance chain until you find a foo declared
- You do not have to go all the way up to the least derived (i.e., most base) function
- In Java, any protected or public function in a base class is visible in a derived class, so you would have to go all the way up the chain.
- Hiding arguably improves encapsulation
- Hiding weakens the ISA relationship, because D ISA B doesn't imply that D has all of B's functionality.

Static fields and methods

- A *static field* is shared among all of the objects
- It is associated with a class, and not with an object of that class
- They become like global variables and are easy to share across all of the objects of a class (or objects of different classes)
- A static function is associated with a class
  - Static functions are not passed a *this* pointer since they are not associated with an object
  - Of course, like any code, they can allocate objects and call functions on that allocated object, and access public fields of that object.

See Example3 code

```
class Item {
public:
  Item(int, float);
  Item( );
  virtual ~Item( );
  virtual void print( );
  static int getNumberOfItems( );
private: // "private" to make explicit
  static int numberOfItems;
  int itemNum;
  float price;
};
```

int Item::numberOfItems = 0; // initialize statics like this
// in the .cpp file

```
int Item::getNumberOfItems( ) {
  return numberOfItems;
}

Item::Item(int i, float p) : itemNum(i), price(p) {
  numberOfItems++;
}

Item::Item( ) { }

Item::~Item( ) { }

void Item::print( ) {
  std::cout << "number of items: " << numberOfItems;
  std::cout << ", item number: " << itemNum;
  std::cout << ", price: " << price << std::endl;
}
```

```
class Item {
public:
  Item(int, float);
  Item( );
  virtual ~Item( );
  virtual void print( );
  static int getNumberOfItems( );
private:
  static int numberOfItems;
  int itemNum;
  float price;
};
```

```
int Item::numberOfItems = 0; // initialize statics like this


int Item::getNumberOfItems( ) {
  return numberOfItems;
}

Item::Item(int i, float p) : itemNum(i), price(p) {
  numberOfItems++;
}

Item::Item( ) {numberOfItems++;}

Item::~Item( ) { }

void Item::print( ) {
  std::cout << "number of items: " << numberOfItems;
  std::cout << ", item number: " << itemNum;
  std::cout << ", price: " << price << std::endl;
}
```

```
class Item {
public:
   Item(int, float);
   Item( );
   virtual ~Item( );
   virtual void print( );
   static int getNumberOfItems( );
private: // "private" to make explicit
   static int numberOfItems;
   int itemNum;
   float price;
};
```

```
int Item::numberOfItems = 0;

int Item::getNumberOfItems( ) {
   return numberOfItems;
}
```

```
Item::Item(int i, float p) : itemNum(i), price(p) {
   numberOfItems++;
}

Item::Item( ) { }

Item::~Item( ) { }

void Item::print( ) {
   std::cout << "number of items: " << numberOfItems;
   std::cout << ", item number: " << itemNum;
   std::cout << ", price: " << price << std::endl;
}
```

Why is numberOfItems initialized in the .cpp file?

1. C++ requires variables (including statics) be initialized exactly once

- We cannot initialize it in the .h file
  - This would imply that every time the .h file is included by a class, initialization code would be created.
  - The compiler would then have to track which .h's have executed initialization code.
    - What if the initialization is a function of a macro that is expanded at compile time and changes each time it is expanded?
    - What should the value be initialized to? The first time a .h is included? The last time?
    - Confusing behavior would result.
- static const variables of integral types can be initialized in the .h file
  - Microsoft compilers have a way around this, but don't use it – why work to make your code non-portable?

Static functions are not called polymorphically, i.e., virtually (See Example4 code)

- That is, they are not called through the VFT
- Given a call r.f( ), where f( ) is static, the class r is examined for an f( ) function
  - If it is found, it is called directly
  - It is NOT called through the VFT of the object referenced by r
  - This would also be true of p->f( )

Calling static functions

They are called directly, not through the VFT

```
class Item {
public:
  Item(int, float);
  Item( );
  virtual void print( );
  // ILLEGAL virtual static int getNumberOfItems( );
  static int getNumberOfItems( );
private: // "private" to make explicit
  static int numberOfItems;
  int itemNum;
  float priceint
};
// item.cpp
Item::getNumberOfItems( ) {
  std::cout << “base getNumberOfItems\n“;
  return numberOfItems;
}
Item::Item(int i, float p) : itemNum(i), price(p) {
  numberOfItems++;
}
Item::Item( ) {numberOfItems++;}
```

```
void Item::print( ) {
  std::cout << "number of items: " << numberOfItems;
  . . .
}
int Item::numberOfItems = 0; // initialize statics like this
// itemD.h
ItemD : public Item {
public:
  ItemD( );
  static int getNumberOfItems( );
};
ItemD::ItemD( ) { }
int ItemD::getNumberOfItems( ) {
  std::cout << "derived getNumberOfItems" << std::endl;
  return 0;
}
int main (int argc, char *argv[]) {
  Item *iP = new ItemD( );
  iP = new ItemD( );
  iP->getNumberOfItems( );
  iP->print( );
}
Output: std::cout << “base getNumberOfItems“;
number of items: 2
```

Static functions

They are called directly, not through the VFT

```
class Item {
public:
  Item(int, float);
  Item( );
  virtual void print( );
  // ILLEGAL virtual static int getNumberOfItems( );
  static int getNumberOfItems( );
private: // "private" to make explicit
  static int numberOfItems;
  int itemNum;
  float priceint
};
// item.cpp
Item::getNumberOfItems( ) {
  std::cout << “base getNumberOfItems\n“;
  // ILLEGAL ~~std::cout << itemNum << std::endl;~~
  return numberOfItems;
}
Item::Item(int i, float p) : itemNum(i), price(p) {
  numberOfItems++;
}
Item::Item( ) {numberOfItems++;}
```

```
void Item::print( ) {
  std::cout << "number of items: " << numberOfItems;
   . . .
}
int Item::numberOfItems = 0; // initialize statics like this
// itemD.h
ItemD : public Item {
public:
  ItemD( );
  static int getNumberOfItems( );
};
ItemD::ItemD( ) { }
int ItemD::getNumberOfItems( ) {
  std::cout << "derived getNumberOfItems" << std::endl;
  return 0;
}
int main (int argc, char *argv[]) {
  Item *iP = new ItemD( );
  iP = new ItemD( );
  iP->getNumberOfItems( );
  iP->print( );
}
Output: std::cout << “base getNumberOfItems“;
number of items: 2
```

Static functions

They are called directly, not through the VFT

```
class Item {
public:
  Item(int, float);
  Item( );
  virtual void print( );
  static int getNumberOfItems( );
private: // "private" to make explicit
  static int numberOfItems;
  int itemNum;
  float priceint
};
// item.cpp
Item::getNumberOfItems( ) {
  std::cout << “base getNumberOfItems\n“;
  return numberOfItems;
}
Item::Item(int i, float p) : itemNum(i), price(p) {
  numberOfItems++;
}
Item::Item( ) {numberOfItems++;}
```

```
void Item::print( ) {
  std::cout << "number of items: " << numberOfItems;
   . . .
}
int Item::numberOfItems = 0; // initialize statics like this
// itemD.h
ItemD : public Item {
public:
  ItemD( );
  static int getNumberOfItems( );
};
ItemD::ItemD( ) { }
int ItemD::getNumberOfItems( ) {
  std::cout << "derived getNumberOfItems" << std::endl;
  return 0;
}
int main (int argc, char *argv[]) {
  Item *iP = new ItemD( );
  iP = new ItemD( );
  Item::getNumberOfItems( ); // another way to call
  iP->print( );
}
Output: std::cout << “base getNumberOfItems“;
number of items: 2
```