

```
leveland% CC s.c
```

```
CC s.c
```

```
leveland% a.out
```

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73,79, 83, 89, 97, 101, 103, 107,
109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
367, 373, 379, 383, leveland%
```

The contribution of the base class `Emitter`:

It allows binding of `source->out()` to change **during execution** from `Counter::out()` to `Filter::out()`.

It's the virtual function that allows this:

virtual function: `Emitter::out()`

actual functions: `Counter::out()`, `Filter::out()`.

What gets this rebinding through C++'s compile-time type-checking system?

`Counter::out()` and `Filter::out` are members of classes derived from `Emitter`, and have the same signature as `Emitter::out()`.

Private derivation

```
class List // base class
{ public: // visible throughout scope of List
  int empty();
  protected: /* visible in members of List and
               in members of classes derived
               immediately from List */
  List() { rear = new Cell(0); }
  void push(int); // add to front
  int pop(); // remove from front
  void put(int); // add to rear
private: // visible only in List's members
  Cell *rear;
};
```

Two classes derived *privately* from `List`:

```
class Queue : private List
{ // default for inherited members
public:
  Queue() {} // calls List() implicitly
  void put(int x) { List::put(x); } // in-line
  int get() { return pop(); } // in-line
  List::empty; // overrides default visibility
  // push is inherited too, but is not "publicized"
};
```

```
class Stack : private List
{
public:
  Stack() {}
  void push(int x) { List::push(x); }
  int pop() { return List::pop(); }
  List::empty;
};
```

Typical calls:

```
Stack s; int x;
```

```
... s.push(e); ...
```

```
... if( !s.empty() ) x = s.pop(); ...
```

```
Queue q; int y;
```

```
... q.put(e); ... y = q.get(); ...
```

Private derivation shares *implementations*;

public derivation shares *interfaces*.

Private-derived types are *not* subtypes.

A counterexample (simplified `List`, `Stack`, `Queue`):

```
class List
{
public:
  List();
  void push(int);
  int pop();
  void put(int);
};
List a;
a.push(3); // OK
x = a.pop(); // OK
a.put(4); // OK
```

```
class Stack : List
{
public:
  Stack();
  List::push;
  List::pop;
  // put is inherited
  // private
};
Stack b;
b.push(3); // OK
x = b.pop(); // OK
b.put(4); //error
```

```
class Queue : public List
{
public:
  Queue();
};
Queue c;
c.push(4); //OK
x = c.pop(); //OK
c.put(3); //OK
```

So `Stack` is **not** a **subtype** of `List`—a `Stack` cannot be used where a `List` is required.

`Queue`, however, **is** a subtype of `List`—a `Queue` can be used wherever a `List` is expected.

Another counterexample:

```
class Base
{
public:
    void foo(){ cout << "I'm foo.\n"; }
};
class Derived : private Base
{
public:
    Base::foo();
};
void doFoo( Base& x )
{
    x.foo();
}
main()
{ Base b; Derived d;
  doFoo(b); // OK
  doFoo(d); // error
}
```

Error message:

cast: Derived* -> base Base*; **private base class**

So an instance of Derived cannot be used where an instance of Base is expected;

∴ Derived is **not** a subtype of Base.

21 April 2005

Fault Tolerance

Handling errors by means of **Exceptions**

When an ADT's *implementors* are distinct from the ADTs' *clients*, we have a problem:

the ADT's **author** knows how to **detect** errors within the ADT, but not how to respond to them.

the ADT's **client** knows how to **respond** to errors, but cannot detect them.

Example (*sketch*): unbounded integers

```
class BigNum
{
public:
    BigNum( unsigned = 0 );
    BigNum( const char[] );
    BigNum( const BigNum& );
    ~BigNum(){ delete digits; }
    BigNum operator+( const BigNum& );
    BigNum operator*( const BigNum& );
    // etc...
private:
    short* digits;
    unsigned ndigits;
};
```

21 April 2005

Design problem:

How should the BigNum class handle a bad string passed to BigNum::BigNum(const char[])?

Unknowns:

- Is the client program interactive?
- Did the bad string come from user input?

If so, the user can be prompted for corrected data; if not, ...?

Customary solutions:

- pre-scan the input string for validity (reduces benefit from the ADT, doubles the cost)
- test program state (errno) after each call (tiresome, error-prone, inflates program)
- return some "invalid" value (which ints are "invalid" for this client?)
- call a client-supplied error-handling function (what are its arguments?)
- terminate the program (overkill)

The C++/ML/Java solution: the **exception**.

21 April 2005

Adding *exceptions* to an Array ADT with range-checking:

```
class Array
{
public:
    class Range { }; // exception class
    Array( int s = ArraySize );
    // ...
    int& operator[]( int );
    int getSize();
protected:
    int *data;
    int size;
};
inline int& Array::operator[]( int index )
{
    if( 0 <= index && index < size )
        return data[ index ];

    throw Range(); // Constructs and throws an
                  // object of type Range.
                  // (Range() is a call to the Range
                  // class's default constructor.)
}
```

21 April 2005