

Think of the “Borg” on Star Trek.

Borg crew member = 1 object

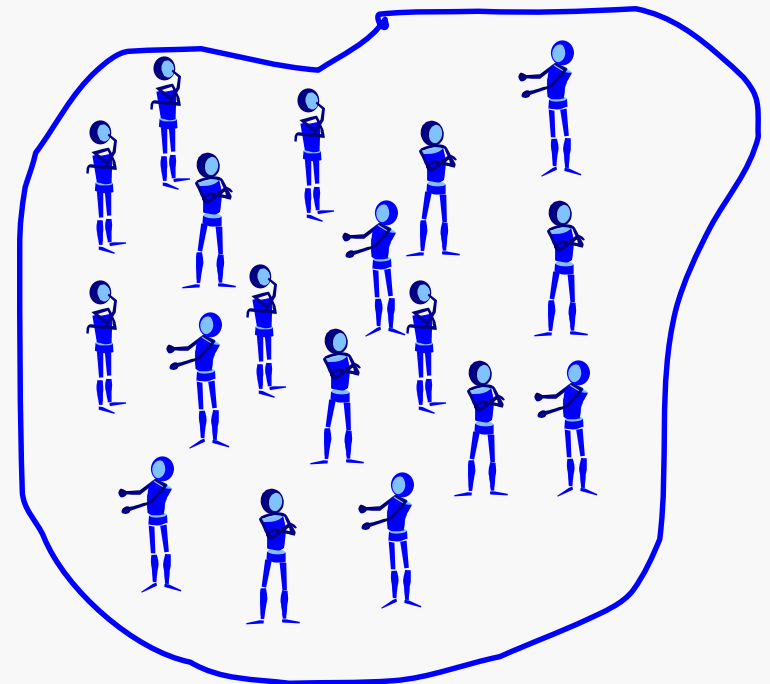
Borg Collective = composition of objects

To achieve the purpose of the Collective, each Borg must continually communicate with other Borg. A Borg can be a “sender” or a “receiver”, and may play both roles at different times.

Similarly, objects can be senders or receivers. Objects can also serve as messages!

The resulting software system is viewed as a collection of collaborating objects.

Collaboration requires communication...



## Kinds:

**By name** – implicit communication that can occur when one object is in a scope where its name is visible to other object

**By parameter passing** – a method of a class take an object as a parameter

**By return value** – a method returns an object

Parameters and return values allow two-way interaction

Object may be communicated by:

Copying

Identity

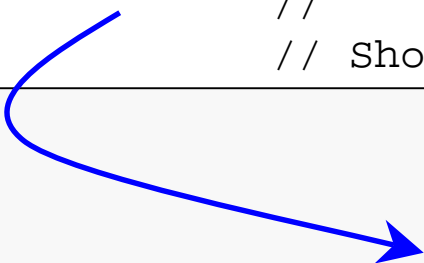
Reference

Pointer

May want to control whether receiver can modify the object, and if so, whether the sender sees any changes made by the receiver.

By name: sender “knows the name” of the receiver and uses the name to access the public interface of the receiver.

```
DisplayableNumber D(42, &cout);  
  
D.Show(); // The function accesses D by name,  
          //      passing the object cout by address,  
          // Show()accesses cout by a pointer member
```

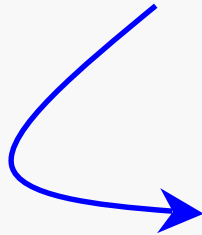


```
void DisplayableNumber::Show() const {  
    *Out << Count << endl;  
}
```

The "name" may be the identifier associated with the object, or a pointer to the object.

An object may be passed as a function parameter:

```
DisplayableNumber D(42, &cout);  
ofstream oFile("output.text");  
  
D.ShowIn(&oFile); // D receives oFile as a parameter
```



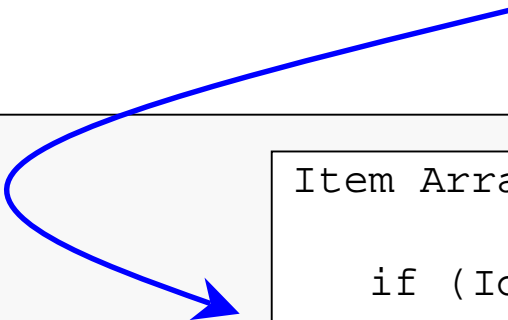
```
void DisplayableNumber::ShowIn(ostream* setOut) {  
    Out = setOut; // store address of oFile  
}
```

As is always the case in C++, by default an object parameter is passed by value to the called function.

An object may be the return value from a function:

```
typedef DisplayableNumber Item;           // define an alias
const int Digits = 10;
Array LCD(Digits, Item(0, &cout));      // array of DNs
. . .

DisplayableNumber Digit4 = LCD.Retrieve(4); // shallow copy
. . .
```



```
Item ArrayClass::Retrieve(Item Idx) const {

    if (Idx >= Usage)
        return -1;
    else
        return List[Idx];
}
```

Using an object as the return value provides a mechanism for encapsulating a body of related heterogeneous data.

Is object communicated by:

- copying
- reference
- pointer

Can the receiver modify the object?

If the receiver does modify the object, does the sender see the changes?

What language syntax is used in receiver to access (. or ->)?

<u>Technique</u>	<u>Copied</u>	<u>Changeable</u>	<u>Visible</u>	<u>C++</u>	<u>Access Syntax</u>
by copy	yes	yes	no		.
by reference	no	yes	yes		.
by pointer	no	yes	yes		->
by const reference	no	no	no		.
by pointer to const	no	no	no		->

## By Copy:

- ✓ Sender is “isolated” from changes by receiver
- ✗ No good if sender/receiver want to share object
- ✗ Bad if object is large (why?)

## By Identity (pointer or reference):

- ✗ No isolation
- ✓ Permits sharing of objects
- ✓ Improves memory cost for large objects

Person class represents basic characteristics of a person.

A variety of object communications take place in the Person interface.

```
enum Gender {MALE, FEMALE, GENDERUNKNOWN};
class Person {
private:
    Name      Nom;        // sub-object
    Address  Addr;        // sub-object
    Person*  Spouse;      // association link
    Gender   Gen;         // simple data member
public:
    . . .
    void changeAddress(const Address& newAddr);
    Address getAddress() const;
    . . .
    void setSpouse(Person* const Sp);
    Person* getSpouse() const;
    . . .
};
```

The `Person` member `changeAddr ( )` receives an `Address` object as a parameter; pass by constant reference is used to avoid copying while safeguarding the actual parameter:

```
void Person::changeAddr(const Address& newAddr) {  
    Addr = newAddr;  
}
```

The `Person` member `setSpouse ( )` receives a `Person` object as a parameter; pass by constant pointer is used to avoid copying while safeguarding the actual pointer (but not its target):

```
void Person::setSpouse(Person* const Sp) {  
    Spouse = Sp;  
}
```

## Example: an Object as a return Value

The Person member getAddress ( ) returns a sub-object:

```
Address Person::getAddress() const {  
    return Addr;  
}
```

As we've seen, one thing this allows is "chaining" of member function calls:

```
Name JBHName("Joe", "Bob", "Hokie");  
Address JBHAddr("Oak Bridge Apts", "#13", "Blacksburg",  
               "Virginia", "24060");  
Person JBH(JBHName, JBHAddr, MALE);  
  
cout << JBH.changeAddress().getZip() << endl;
```

## Example: returning a Reference

Changing the Person member setAddress ( ):

```
Person& Person::setAddress(const Address& newAddr) {  
    Addr = newAddr;  
    return (*this);  
}
```

**When the return value is a reference,  
no copying is performed.**

Returning a reference to the "implicit" object allows carrying out multiple operations in a single statement:

```
Address MovedTo("3221 Bob Petit Blvd", "Apt 6", "Baton Rouge",  
                "Louisiana", "78703");  
  
Person JT(. . .);  
  
JBH.changeAddress(MovedTo).setSpouse(&JT);
```

An nameless (i.e., unnamed) object.

Useful:

- for temporary use (parameter in a method call, return, expression term)
- as default value for an object parameter

Anonymous objects are created by a direct invocation of a class constructor.

There's an example of this in the Aggregation notes (slide C08.16).

Anonymous objects are frequently used in conjunction with mutators and constructors when aggregation is involved, providing a cleaner interface to the aggregating class.

Without anonymous objects, we have a mild mess:

```
Name      JBHName("Joe", "Bob", "Hokie");
Address   JBHAddr("Oak Bridge Apts", "#13", "Blacksburg",
                 "Virginia", "24060");
Person    JBH(JBHName, JBHAddr, MALE);

. . .
```

With anonymous objects we reduce pollution of the local namespace:

```
Person    JBH(Name("Joe", "Bob", "Hokie"),
              Address("Oak Bridge Apts", "#13", "Blacksburg",
                     "Virginia", "24060"),
              MALE);

. . .
```

Used as default parameter values, anonymous objects provide a relatively simple way to control initialization and reduce class interface clutter:

```
Person::Person(Name      N = Name("I", "M", "Nobody"),
                Address  A = Address("No Street", "No Number",
                                     "No City", "No State",
                                     "00000"),
                Gender    G = GENDERUNKNOWN) {
    Nom      = N;
    Addr     = A;
    Spouse   = NULL;
    Gen      = G;
}
```