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…
Communication Involving Objects

Kinds:

- **by name**: implicit communication that can occur when one object is in a scope where its name is visible to other objects.
- **by parameter passing**: a method of a class takes 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.
Inter-Object Communication

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.
Passing an Object as a Parameter

An object may be passed as a function parameter:

```cpp
DisplayableNumber D(42, &cout);
ofstream oFile("output.text");

D.ShowIn(&oFile);  // D receives oFile as a parameter
```

```cpp
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.
Returning an Object

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 Array::Retrieve(int 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.
Different Ways to Communicate

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 -> )?
### Characteristics of Communicated Objects

<table>
<thead>
<tr>
<th>Technique</th>
<th>Copied</th>
<th>Changeable</th>
<th>Visible</th>
<th>C++ Access</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>by copy</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>by reference</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>by pointer</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td>-&gt;</td>
</tr>
<tr>
<td>by const reference</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>by pointer to const</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td></td>
<td>-&gt;</td>
</tr>
</tbody>
</table>

**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
Example: Person Class

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;
    ...  
};
```
Example: an Object as a Parameter

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:

```cpp
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):

```cpp
void Person::setSpouse(Person* const Sp) {
    Spouse = Sp;
}
```
Example: an Object as a return Value

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

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

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

```cpp
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()`:

```cpp
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:

```cpp
Address MovedTo("3221 Bob Petit Blvd", "Apt 6", "Baton Rouge", "Louisiana", "78703");

Person JT(. . .);

JBH.setAddress(MovedTo).setSpouse(&JT);
```
Anonymous Objects

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.
Example: Anonymous Objects as Parameters

Without anonymous objects, we have a mild mess:

```java
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:

```java
Person JBH(Name("Joe", "Bob", "Hokie"),
    Address("Oak Bridge Apts", "#13", "Blacksburg",
        "Virginia", "24060"),
    MALE);
.
```
Example: Anonymous Objects as Defaults

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

```cpp
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;
}
```