Function Calls

- Additionally, using the `&` operator (instead of a `*`) will make that parameter call-by-reference.
  - It will hide the obtained address, but still work with and alter the same object/variable.
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
}
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
}
`void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
} `

`void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
} `
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
}
Call By Reference (2)

```c
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
}
```

Diagram:
- Variables: a, b, temp
- Initialization: a = 2, b = 3
- Swapping: temp = a, a = b, b = temp
- Result: a = 3, b = 3
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
}
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
}
An Aside

• To some of you, we imagine that some of C++’s syntax and structure may be pretty foreign, to say the least.
  – In particular, some people have never worked (heavily) with OO before.
  – This is because there’s a whole different way of thinking about programming tasks in OO.
Object-Oriented Programming

- Object-orientation is quite different.
  - As we’ve seen already, part of its design is to enforce the organization of data into logical, conceptual units within the system.
  - Each object keeps its data private (ideally) and seeks to enforce constraints to keep itself in a proper form.
Object-Orientation

• Object-orientation is quite different.
  – Work gets done by objects interacting with other objects.
    • As such, the exact flow of execution in the program may not be easy to track.
  – Object orientation aims to avoid making anything truly global.
    • Java doesn’t even allow “truly” global variables.
    • C++ allows them.
A Fraction Object

class Fraction
{
    private:
        int numerator;
        int denominator;

    public:
        Fraction add(Fraction &f);
}

public Fraction* Fraction::add(Fraction &f) {
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
Coding in OO

• First, let’s examine this line of code.

    f1.add(f2); //Both are Fractions

• What is this setting up and modeling?
• Secondly, what is going on in add()?
f1.add(f2); //Both are Fractions

•This line is basically saying
“Call the “Fraction.add()”
method from the perspective of f1.
A Fraction Object

```
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```

So, that line of code has an **implied reference** to what was previously called “f1.”
This “implied reference” is known as this within C++. It’s understood to be implied on any “unqualified” field names in the method below.

```cpp
public Fraction* Fraction::add(Fraction &f) {
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```
The use of “numerator” and “denominator”, when not preceded by “f.” here, are with respect to this.

```cpp
public Fraction* Fraction::add(Fraction &f) {
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```
A Fraction Object

What about when we do have “f.” preceding numerator and denominator?

```cpp
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```
A Fraction Object

In such cases, the perspective *shifts* to that of the object *f*, from which it then operates for the field or method after the “.”.

```cpp
public Fraction* Fraction::add(Fraction &f)
{
    int num = numerator * f.denominator;
    num += f.numerator * denominator;
    int dnm = f.denominator * denominator;

    return new Fraction(num, dnm);
}
```
f1.add(f2); //Both are Fractions

• Even though the add() method is operating with two different Fraction class instances, the code is able to keep track of which is this and which is the parameter f.
Documentation

• Documentation is the “plain” English text accompanying code that seeks to explain its structure and use.
  – Some of this documentation is typically in comments, directly in the code.
  – Other documentation may be in external documents.
• For complex code, it can be very helpful to place inline comments on a “paragraph” level, explaining what purpose that block of code is accomplishing.
  – A line-by-line commentary may clarify what the code is doing, but rarely indicates why.
  • Note the purpose of your code – its goal.
• We’ve already noted two different ways to comment within C++:

// This is a one-line comment.

/* This is a block comment, spanning multiple lines. */
• In producing documentation for a method, it is wise to place some form of the “relationships” criterion within the description.
  – Generally, the conceptual purpose which a method, field, or class serves.
• One should also include an explanation of the method’s *pre-conditions*, if it has any.
  – Pre-conditions: the limitations a particular method imposes on its inputs.
  – If a method is called with arguments that do not match its preconditions, its behavior is considered to be undefined.
• As there exists a notion of *preconditions*, there also exist *post-conditions*.
  – Post-conditions: the effect a method has on its inputs (any unaffected/unlisted input should remain untouched), any generated exceptions, information about the return value, and effects on object state.
Benefits

• Documentation helps other programmers to understand the role of each accessible field and method for a given class.
• Documentation inside the code provides great reference material for future maintenance efforts.