Pass By Reference vs. Pass By Value

Reference Variables

- A **reference** variable is a nickname, or alias, for some other variable
- To delare a reference variable, we use the unary operator &

```
int n = 5;          // this declares a variable, n
int & r = n;        // this declares r as a reference to n
```

  In this example, r is now a reference to n. (They are both referring to the SAME storage location in memory).
- To declare a reference variable, add the & operator after the type
- Here is a small example program that illustrates a reference variable
- Note: The notation can become confusing when different sources place the & differently. The following three declarations are equivalent:

```
int &r = n;
int& r = n;
int & r = n;
```

  The spacing between the "int" and the "r" is irrelevant. All three of these declare r as a reference variable that refers to n.

WHY???!

- While the above code example shows what a reference variable is, you will not likely use it this way!
- In this example, the regular variable and the reference are in the *same scope*, so it seems silly. ("Why do I need to call it r when I can call it x ?")
- So when are references useful? When the two variables are in *different scopes* (this means functions!)

Pass By Value

- Recall that the variables in the formal parameter list are always *local variables* of a function
- Consider this example program
  - The `Twice` function takes two integer parameters, and multiplies each by 2.
  - Note that the original variables passed into the function from `main()` are **not** affected by the function call
  - The local parameters `a` and `b` are *copies* of the original data sent in on the call
- This is known as **Pass By Value** - function parameters receive copies of the data sent in.

```
void Func1(int x, double y)
{
  x = 12;              // these lines will not affect the caller
  y = 20.5;            //  they change LOCAL variables x and y
}
```

- In the function above, any `int` and `double` *r-values* may be sent in

```
int num = 5;
double avg = 10.7;
Func1(num, avg);              // legal
Func1(4, 10.6);               // legal
Func1(num + 6, avg - 10.6);   // legal
```

Pass By Reference

- In this version of the example, the `Twice` function looks like this:

```
void Twice(int& a, int& b)
{
  a *= 2;
  b *= 2;
}
```

  - Note that when it is run, the variables passed into `Twice` from the `main()` function *DO* get changed by the function
  - The parameters `a` and `b` are still local to the function, but they are *reference* variables (i.e. nicknames to the original variables passed in (x and y))

- When reference variables are used as formal parameters, this is known as **Pass By Reference**

```
void Func2(int& x, double& y)
{
  x = 12;             // these WILL affect the original arguments
  y = 20.5;
}
```

- When a function expects strict reference types in the parameter list, an *L-value* (i.e. a variable, or storage location) must be passed in

```
int num;
double avg;
Func2(num, avg);              // legal
Func2(4, 10.6);               // NOT legal
Func2(num + 6, avg - 10.6);   // NOT legal
```

- Note: This also works the same for return types. A return by value means a copy will be made. A reference return type sends back a reference to the original.

```
int Task1(int x, double y);   // uses return by value
int& Task2(int x, double y);  // uses return by reference
```

  This is a trickier situation than reference parameters (which we will not see in detail right now).

Comparing: Value vs. Reference

- **Pass By Value**
  - The local parameters are copies of the original arguments passed in
  - Changes made in the function to these variables do not affect originals
- **Pass By Reference**
  - The local parameters are references to the storage locations of the original arguments passed in.
  - Changes to these variables in the function **will** affect the originals
  - No copy is made, so overhead of copying (time, storage) is saved

`const` Reference Parameters:

- The keyword **const** can be used on reference parameters.

  ```
  void Func3(const int& x);     // pass by const reference
  ```

  This will prevent x from being changed in the function body

- General format

  ```
  const typeName & variableName
  ```

  This establishes *variableName* as a reference to a location that cannot be changed through the use of *variableName*.
- This would be used to avoid the overhead of making a copy (especially of a large item), but still prevent the data from being changed
- Since the compiler will guarantee that the parameter value cannot change, it IS legal to pass in any R-value in this case:

  ```
  int num = 5;
  Func3(num);           // legal
  Func3(10);            // legal
  Func3(num + 50);      // legal
  ```

- Code example illustrating a function with a const reference parameter