Call by value

The functions we have seen so far use what is called **call by value** for their parameters.

- The parameter is a local variable that holds the value passed in.
- It gets a copy of the value, as though you wrote
  `parameter = argument;`
- Consider this code:

```
void myfunc(int input) {
  input++; ...
}
int x = 3; myfunc(x); cout << x << endl;
```

- What will be printed? **3**
  - `input++` changed the value of `input`.
  - But that was just a **copy** of `x`.

DIY

Challenge Activity 5.23.1 and 5.23.2 from ZyBook

Call by reference

There is another way to pass parameters, called **call by reference**.

- In C++ we denote this by putting an address of operator **&** before the parameter name.

  ```
  void reverse(string &sequence);
  ```

  - Instead of the parameter holding a copy of the argument's value, it actually refers to the argument itself.
    - ⋆ It uses the same *address* (location in memory)
  - So changing a reference parameter **does** change the original argument.
    - ⋆ Even if the argument isn't in this function's scope!
  - That does mean the argument has to be a variable: `reverse("Hello");` — Error: can't change a literal!

Call by reference examples

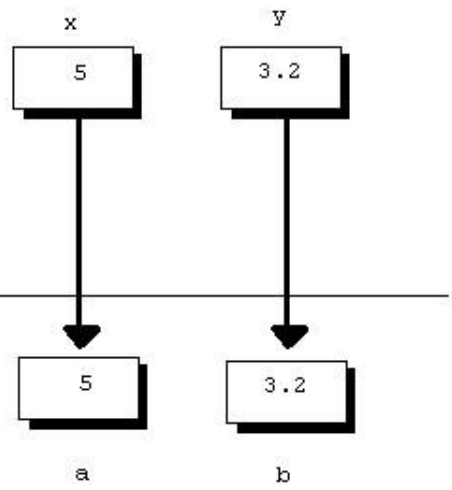
There are some things that absolutely require call by reference:

```
void swap(int &a, int &b) {
  int temp = a;
  a = b;
  b = temp;
}
int a = 2, b = 4;
swap(a, b);
cout << "a = " << a << endl;
// swap(2, 4) would be illegal!
```

- Here the whole point of the function is to change the arguments.
- Call by value just wouldn't work.

Call by value vs. call by reference

```
int x = 5;
double y = 3.2;
...
//activation
changeArgs(x, y);
```

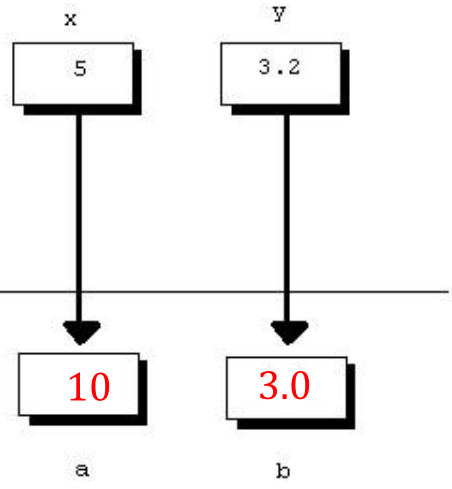



```
//function definition
void changeArgs(int a, double b)
{
        b = 3.0;
        a = 2 * a;
}
```

enter into function calling...

Call by value vs. call by reference

```
int x = 5;
double y = 3.2;
...
//activation
changeArgs(x, y);
```

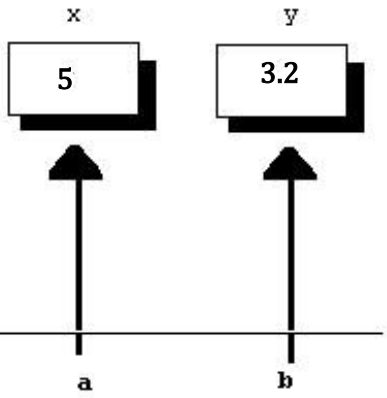



```
//function definition
void changeArgs(int a, double b)
{
        b = 3.0;
        a = 2 * a;
}
```

while calling function...

Call by value vs. **call by reference**

```
int x = 5;
double y = 3.2;
...
//activation
changeArgs(x, y);
```

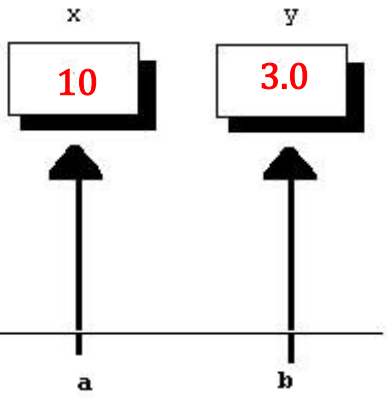



```
//function definition
void changeArgs(int& a, double& b)
{
        b = 3.0;
        a = 2 * a;
}
```

enter into function calling...

Call by value vs. **call by reference**

```
int x = 5;
double y = 3.2;
...
//activation
changeArgs(x, y);
```




```
//function definition
void changeArgs(int& a, double& b)
{
        b = 3.0;
        a = 2 * a;
}
```

while calling function...

Call by reference examples

There are some things that absolutely require call by reference:

```
void swap(int &a, int &b) {
  int temp = a;
  a = b;
  b = temp;
}
int a = 2, b = 4;
swap(a, b);
cout << "a = " << a << endl;
// swap(2, 4) would be illegal!
```

- Here the whole point of the function is to change the arguments.
- Call by value just wouldn't work.

Reference versus value

- Why would you want to use call by reference?
  - Allows getting multiple results from a function.
  - Avoids copying large variables (string, etc.).
  - Can change several things in a coordinated way.
- Why would you want to use call by value?
  - Safer for the caller (arguments won't be changed).
  - Less surprising.
  - Allows using literals etc. as arguments.
  - More efficient for small variables (int, double).

DIY

Challenge Activity 5.14.1 from ZyBook

Const references

We can get the best of both worlds with a **constant reference**:

- Put **const** at the beginning of the reference parameter type:
  `int num_codons(const string &seq) ...`
- The parameter is a reference to the argument, so no copying.
- But we're not allowed to modify it, so it's as safe as call-by-value.
- And literals are allowed as with call-by-value.
  - For strings and other potentially large types, using const references instead of call-by-value is a very common optimization.
  - In chapter 9 we'll see more uses for them.

Coding with your instructor

Please download the source file from the following link:

https://www.cs.uky.edu/~yipike/CS215/LoopExamplesKey.cpp

An we are going to define our own functions to solve the same problem.

Practice Questions???

Write the following functions:

a. `int first_digit(int n)`, returning the first digit of the argument

b. `int last_digit(int n)`, returning the last digit of the argument

c. `int digits(int n)`, returning the number of digits of the argument

Coding with your instructor

Please download the source file from the following link:

https://www.cs.uky.edu/~yipike/CS215/DemoFunctions.cpp

Practice how to define your own functions to solve the problem.

char

- The type char is used for single character.
- Character literals use single quotes.

```
char middle_initial = 'L';
char first = 'Yi';     //Error!
char second = "P";     //Error!
```

- Note the difference:
  - `'A'` is a character literal.
  - `"A"` is a string literal (that happens to contain one character)

More about Characters

- A few special characters are written differently. These are still single (individual) characters: the **escape sequence** characters.

  '\n' -- newline (ENTER)

  '\t' -- tab

  '\a' -- alarm (beeps when printed)

  '\0' -- The null terminator of a C string

  '\\' -- backslash (have to double up)

More about Characters

- The **escape sequence** characters work in strings too:

```
string message = "Hello, world\n";
string folder = "C:\\Program Files";
```

Another example from Lab1:

(the mouth of a happy face, remember?)

```
cout << "\\_/" << endl;
```

More about Characters

- Each character has an associated numerical code.
  - For simple English text with no accent marks, we use the ASCII code
  - http://en.wikipedia.org/wiki/ASCII
  - A char can be converted to or used as a number

```
int code = 'A';     // code = 65
char next = 'A' + 1; // next = 'B'
```