Passing values to functions

Either global variables
parameters

Problem: Global variables – accessible to the whole file (or even other files in the app)

```
--includes.h
extern int count;

--main.cpp
#include "includes.h"
int count = 4;              //main does not know if any
                            //other app has externed this global

--other.cpp
#include "includes.h"
cout<<count; // will output 4
```

You don't know when they are accessed, or by what. They are available to everything in program including other cpp files that include your header. If above count is incorrect how do you find what accessed it?

This problem becomes more and more difficult as the application grows in code size. And when you introduce threads. They all have access to your global (STATIC too?) variables. You cannot easily tell when they are accessed.

There are ways to debug both single and multithreaded versions of this but they are a pain.

There are however times when global variables are appropriate (multithreaded database situation, singleton pattern) . But they are all too advanced for this class.
You can wrap your globals in an object and have accessor methods with error reporting.
Or you can pass the data structures around via functions;

```
class myGlobalVar{
public:
        int get();
        void set(int newi);
private:
        int i;
};

int myGlobalVar::get(){
        //log data here
        return i;
}

void myGlobalVar::set(int newi){
        //log data here
        i=newi;
}
```

Pass by Value

Pass a copy. Changes made in a function are lost.

Show program, show passing data, show what happens to data on return;

Kind of error prone, you think you are changing things but no

Reference and address of

A reference is a pointer to something that;

Cannot be changed

Must be set on initialization

Cannot be null

Once initialized you ccan access a variables value using either the ref or original var

Demo Show what a reference is; show how memory is managed

Not used very much outside of passing objects to functions

Address of operator

You can pass something called a reference, which pass a function the memory address of the object itself, Changes made in a function are preserved on return.

Way to pass the actual data to a function so you see functional modifications on return

Show it in action

Pointer

Like a souped up reference

Can reassign;

Use dereference to get access to what it points to

Should set to 0 on creation otherwise inited to garbage

Can be null;

So you must check for null before you dereference it or get access violation (dereference unallocated memory), if you don't initialize it to null and dereference it without setting it you are peeking into some random mem you don't own , access violation again.

Int myint =1;

Int *pInt = myInt;

Get the memory address of myint; p

Get access to the contents of myint *p (called dereferencing)

Demo show memory access. Show change of int

Finally

pass by value = copy of

pass by ref = by address the object itself (changes stick)

pass by pointer = address of the object but can be null (changes stick), also can change what pointer points to.

Demo in functions

Java is always pass-by-value. The difficult thing can be to understand that Java passes objects as references and those references are passed by value.

It goes like this:

```
Dog aDog = new Dog("Max");
foo(aDog);
aDog.getName().equals("Max"); // true

public void foo(Dog d) {
  d.getName().equals("Max"); // true
  d = new Dog("Fifi");
  d.getName().equals("Fifi"); // true
}
```

In this example `aDog.getName()` will still return `"Max"`. `d` is not overwritten in the function as the object reference is passed by value.
Likewise:

```
Dog aDog = new Dog("Max");
foo(aDog);
aDog.getName().equals("Fifi"); // true

public void foo(Dog d) {
  d.getName().equals("Max"); // true
  d.setName("Fifi");
}
```