Passing values to functions(see 4_pointers_refs_functions project)

Why not just use globals?

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

```
--includes.h       //any app including this file will have access to
                   //count
extern int count;

--includes.cpp
#include "includes.h"
int count = 4;              //includes.cpp 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 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.

```
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;
}
```

Or you can pass the data structures around via functions;

For most cases pass to functions- 3 ways

By value, reference or pointer

What is a Reference?

A reference is an alias to another object;

Cannot be changed

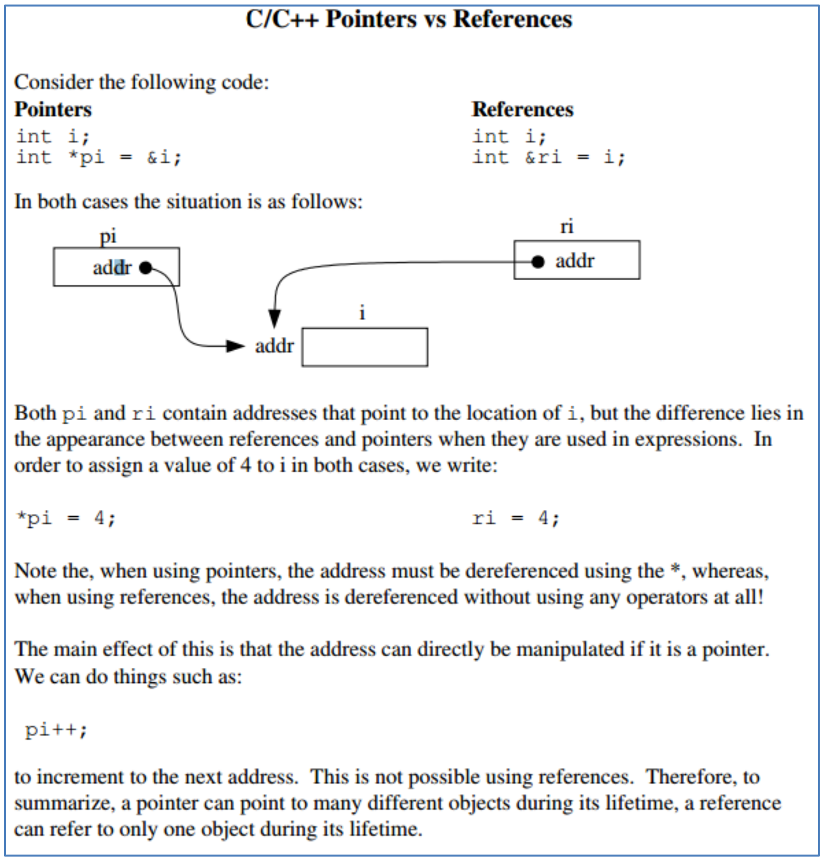
Must be set on initialization

Cannot be null

Once initialized you can access a variables value using either the reference or original var

Example:

```
        int myint1 = 3;
        int &myint2 = myint1;
        myint1++;
        myint2++;
```

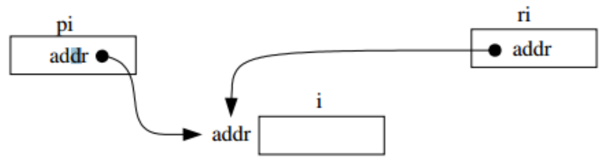


C/C++ Pointers vs References

Consider the following code:

Pointers	**References**
`int i;`	`int i;`
`int *pi = &i;`	`int &ri = i;`

In both cases the situation is as follows:



Both `pi` and `ri` contain addresses that point to the location of `i`, but the difference lies in the appearance between references and pointers when they are used in expressions. In order to assign a value of 4 to i in both cases, we write:

```
*pi = 4;                                ri = 4;
```

Note the, when using pointers, the address must be dereferenced using the *, whereas, when using references, the address is dereferenced without using any operators at all!

The main effect of this is that the address can directly be manipulated if it is a pointer. We can do things such as:

```
 pi++;
```

to increment to the next address. This is not possible using references. Therefore, to summarize, a pointer can point to many different objects during its lifetime, a reference can refer to only one object during its lifetime.

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

Pass by Reference

First what is & address of?

It gives the memory address of a var or pointer

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

Not used very much outside of passing objects to functions

IS used in debugger to find address of variable and then look it up in memory

Using & in functions

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

Demo

Pass by 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

<u>Can be null;</u>

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 *p = 0;         //init to 0
If (p){
            //if not null then use
}
//initializing pointers
Int myint =1;
Int *pInt = myInt;          //p points to myInt
pInt = &myInt               // p points to 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

Summary

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. Must verify that pointer is not null before dereference

Use pointers if you want to do pointer arithmetic with them (e.g. incrementing the pointer address to step through an array) or if you ever have to pass a pointer.

Use references otherwise (it’s a little safer)

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:

```java
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:

```java
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");
}
```