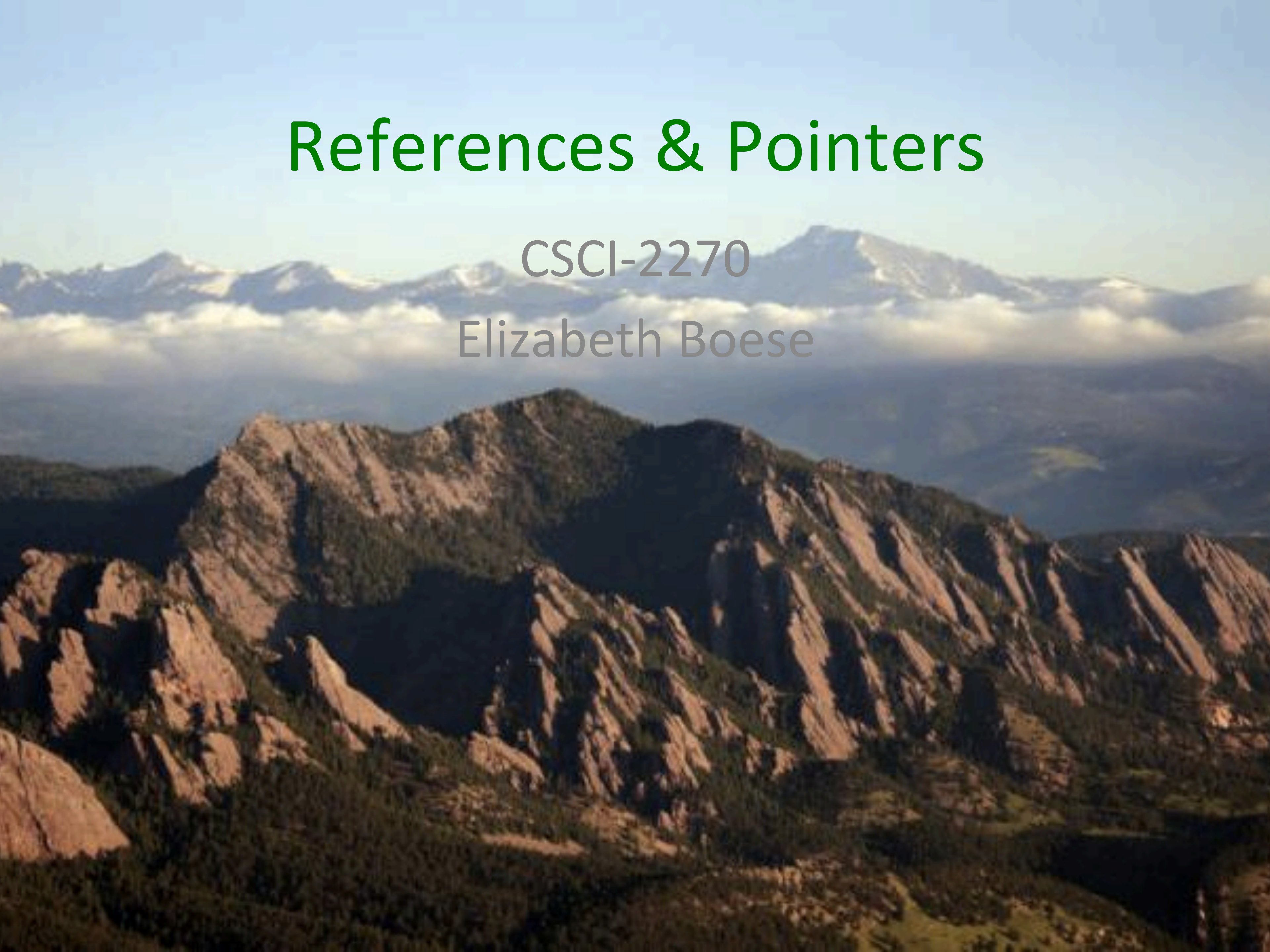
References & Pointers

CSCI-2270

Elizabeth Boese



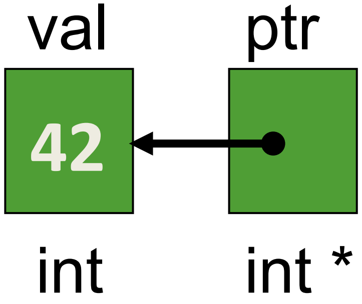
MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A
0x6339392C,
0x7363682E.
I HATE YOU.

http://xkcd.com/138/

POINTERS

What is a Pointer?

- a variable that holds the memory address of (points to) another variable



```
int val = 42;   // regular int
int *ptr;       // pointer to an int
ptr = &val;     // points to val
```

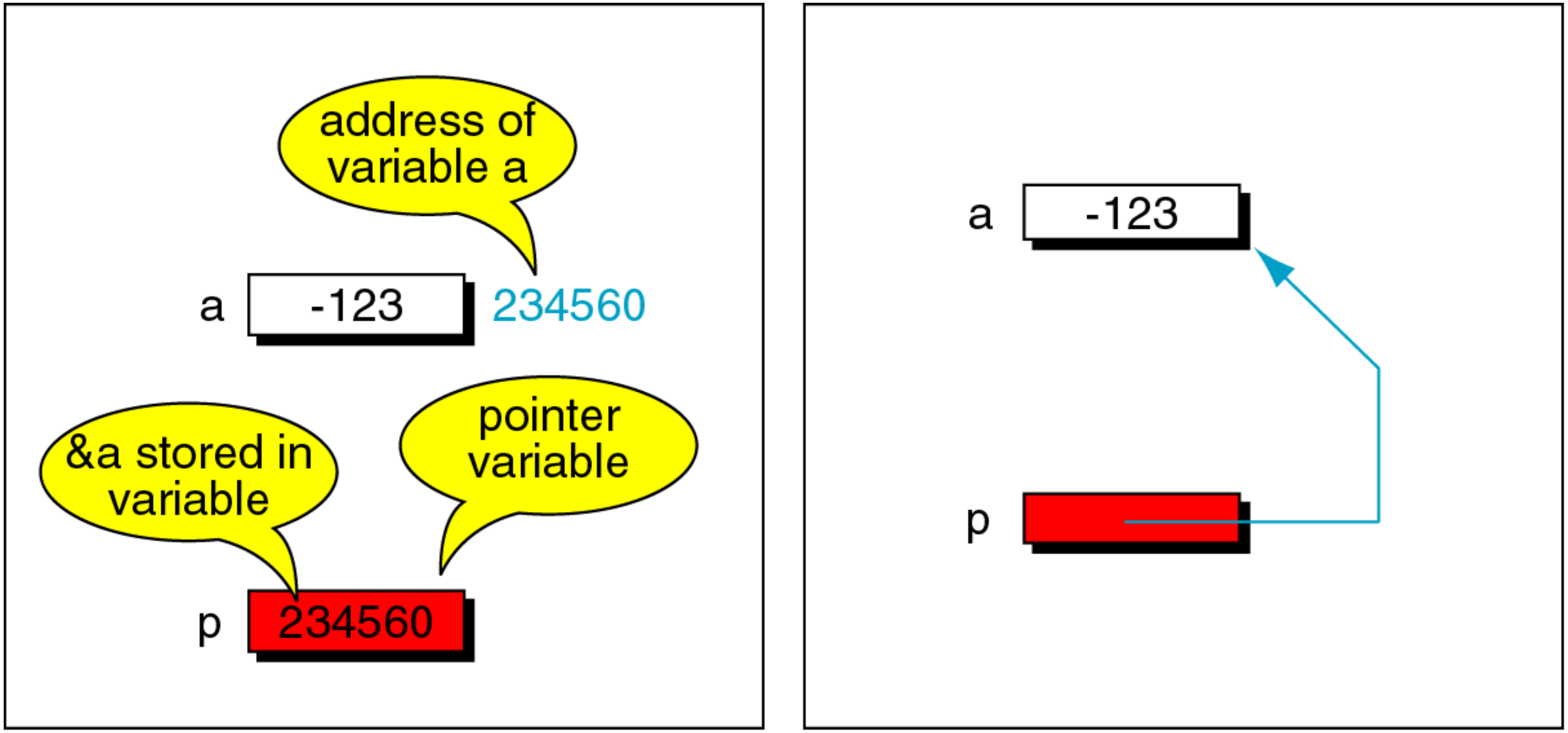
What is a Pointer?

& operator - address of a variable

```
int a = -123;
int             // pointer to an int
p =             // assign to p the address of a
```

Figure 9-5

What is a Pointer?



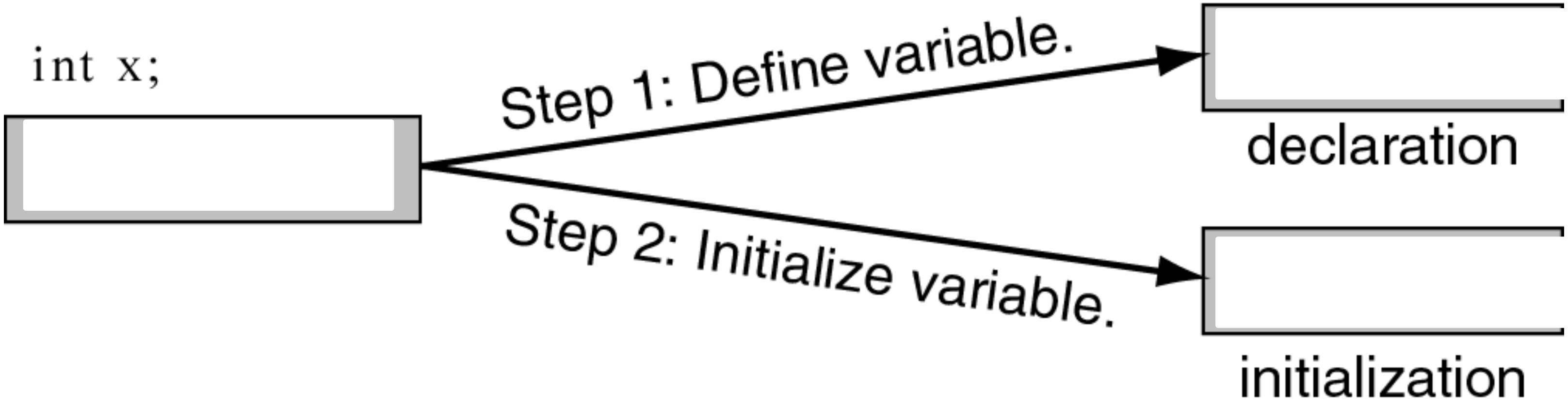
Physical representation

Logical representation

```
int a = -123;
int *p = &a;
```

Figure 9-12

Declaring and Initializing



in 1 step

in 2 steps

Pointers

Reference

De-reference

Pointers

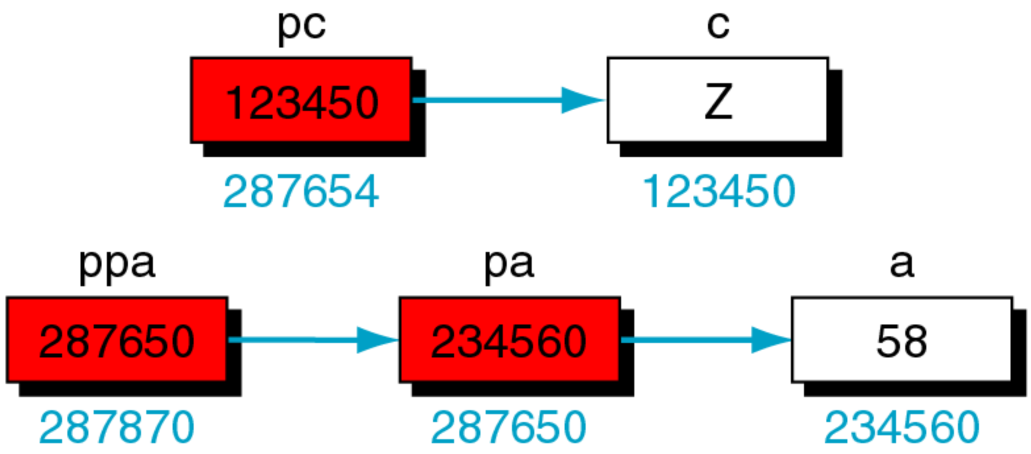
- Very Powerful
  -
  -
  -
  - Creating **arbitrarily-sized lists** of values in memory
  - Working with strings and arrays

- If used incorrectly it can be bad.
  - Pointers + carelessness = Core Dump
  - Segmentation faults -common when you misuse pointers.
  - But, errors don't always crash…
    sometimes they mess up your program later

What is a pointer ?

- A pointer can contain the memory address of any variable type
    - A primitive (int, char, double)
    - An array
    - A struct or union
    - Dynamically allocated memory
    - Another pointer
    - A function

Figure 9-21



```
char c;
char *pc;

int   a;
int  *pa;
int **ppa;

pc  = &c;
pa  = &a;
ppa = &pa;


pc  = &a;
ppa = &a;
```

PARAMETER PASSING (REVIEW)

Parameter Passing (Review)

```cpp
#include <iostream>
using namespace std;

int add(              )
{
   x = 9;
   return x+y;
}

int main()
{
   int a=3, b=5, sum;
   sum = add(a, b);
   cout << sum << endl;
   return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int add(                )
{
   x = 9;
   return x + y;
}

int main()
{
   int a=3, b=5, sum;
   sum = add(a, b);
   cout << sum << endl;
   return 0;
}
```

Parameter Passing (Review)

Arrays

```
#include <iostream>
using namespace std;

int add(int values[])
{
   values[0] = 9;
   return  values[0] + values[1];
}

int main()
{
   int a=3, sum;
   int list[] = {4,5};
   sum = add(list);
   cout << sum << endl;
   cout << list[0] << endl;
   return 0;
}
```

Parameter Passing (Review)

struct

```
struct Point
{
   int x, y;
} ;

void func( Point *p );

int main( )
{
    Point  mypoint = { 1, 2 };
    cout <<  “BEFORE x = “ << mypoint.x << “ y = ” << mypoint.y << endl;
    func( &mypoint );
    cout <<  “AFTER x = “ << mypoint.x << “ y = ” << mypoint.y << endl;
    return 0;
}

void func( Point *p )
{
     (*p).x = 9;
     (*p).y = 11;
}
```

Must wrap *p inside ( )

Pointers and Structs

- Given a pointer to a struct, its members can be accessed using the

  operator.

- The notation avoids the hassle of dereferencing the pointer to access the members of the struct.

```
struct Person{
  string  name;
  int     age;
  string  phone;
  float   height;
};

int main()
{
  Person bob;
  Person* p;
  p = &bob;
  (*p).age = 7;
  p->age = 6;
  cout << p->age << endl;
  return 0;
}
```

RETURNING VALUES

Returning values

- How can you return TWO (or more) values from a function?

- Example: Write a function that takes two numbers and returns the quotient AND the remainder!

Returning values

Example: Write a function that takes two numbers and returns the quotient AND the remainder!

Dividend and divisor are input only

Values can be *returned* to calling function using pointers

```
void  long_division( int  dividend, int  divisor, int  *quotientp,  int  *remainderp)
{
   *quotientp = dividend / divisor;
   *remainderp = dividend % divisor;
}
...
int quot, rem;
long_division(40, 3, &quot, &rem);
```

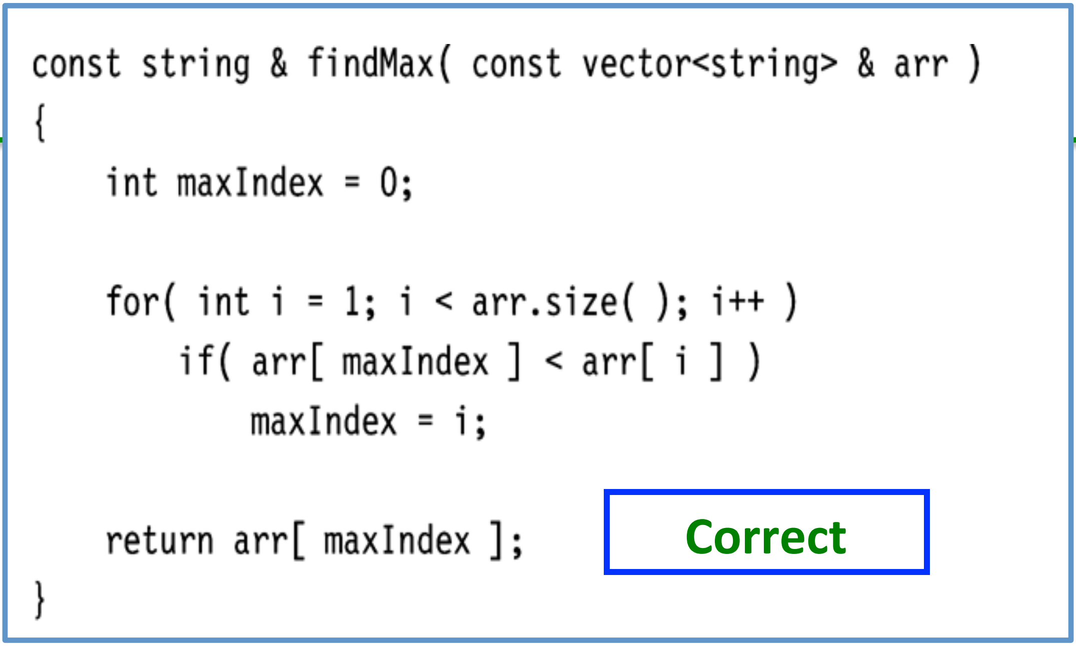
Swaps

- Swap the values of x and y
  - Using “Pass by ” instead of “Pass by ”

```
...
int x = 5;
int y = 6;
swap (          );

void swap(int *a, int *b)
{





}
```

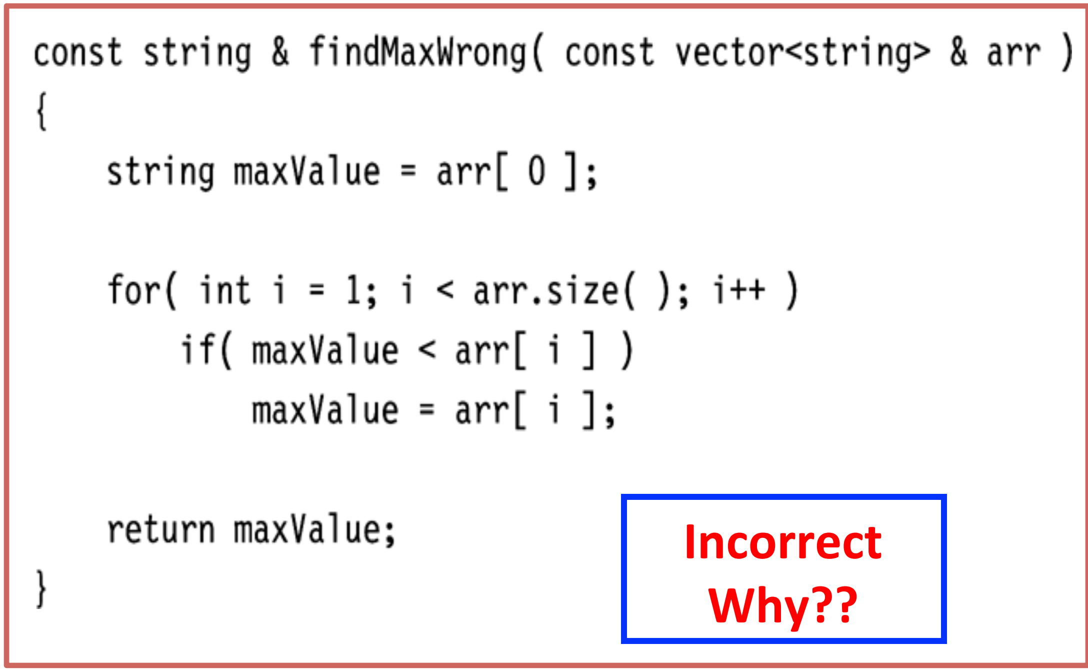
Return Passing

- Return by
  - Copy returned
- Return by
  - Address returned
- Return by
  - Address returned
  - Return value cannot be modified by caller.
- Last two techniques
  - Lifetime of returned value should extend beyond the function called!



```
const string & findMax( const vector<string> & arr )
{
    int maxIndex = 0;

    for( int i = 1; i < arr.size( ); i++ )
        if( arr[ maxIndex ] < arr[ i ] )
            maxIndex = i;

    return arr[ maxIndex ];
}
```

Correct



```
const string & findMaxWrong( const vector<string> & arr )
{
    string maxValue = arr[ 0 ];

    for( int i = 1; i < arr.size( ); i++ )
        if( maxValue < arr[ i ] )
            maxValue = arr[ i ];

    return maxValue;
}
```

Incorrect
Why??

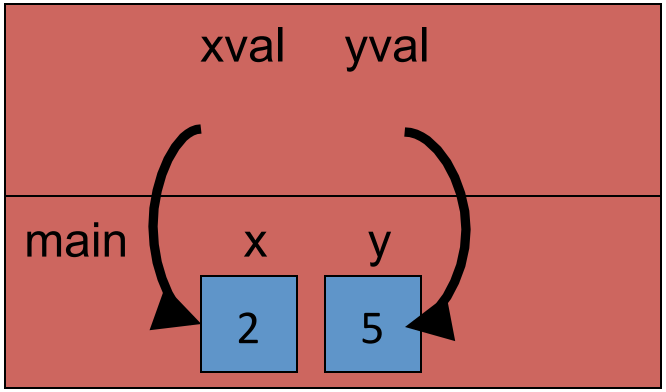
DIFFERENCE BETWEEN
PASS BY REFERENCE
PASS BY POINTER

Parameter Passing

Pass values

```
int main()
{
   methodCall(x, y);
}
void methodCall(int& xval, int& yval)
{
}
```

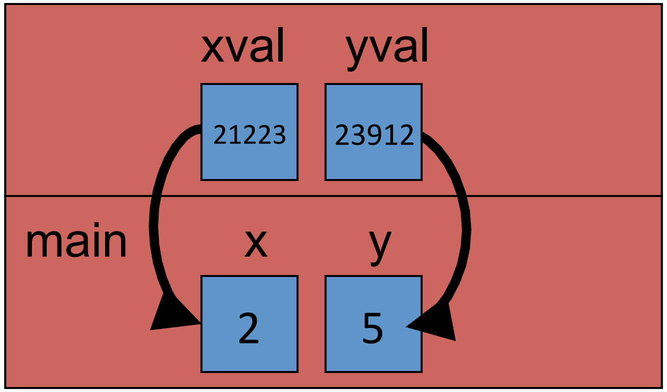
xval and yval are references of x and y. xval and yval don't have their own memory space – an alias for x and y



Pass values

```
int main()
{
   methodCall(&x, &y);
}
void methodCall(int* xval, int* yval)
{
}
```

xval and yval are pointers. They store the address of x and y in a memory location. xval and yval do have their own memory space



Pointers vs References

1. A pointer can be re-assigned any number of times while a reference can not be re-seated after binding.
2. Pointers can point nowhere (NULL), whereas reference always refer to an object.
3. You can't take the address of a reference like you can with pointers.
4. There's no "reference arithmetics" (but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).

As a general rule,

- Use references in function parameters and return types to define useful and self-documenting interfaces.
- Use pointers to implement algorithms and data structures.

http://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-and-a-reference-variable-in

Pointers vs References

A pointer can be re-assigned:

```
int x = 5;
int y = 6;
int *p;
p =  &x;
p = &y;
*p = 10;
assert(x == 5);
assert(y == 10);
```

A reference cannot, and must be assigned at initialization:

```
int x = 5;
int y = 6;
int &r = x;
```

http://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-and-a-reference-variable-in

Pointers vs References

You can have pointers to pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.

```
int x = 0;
int y = 0;
int *p = &x;
int *q = &y;
int **pp = &p;
pp = &q;//*pp = q
**pp = 4;
assert(y == 4);
assert(x == 0);
```

Pointer can be assigned NULL directly, whereas reference cannot.

```
int *p = NULL;
int &r = NULL; <--- compiling error
```

http://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-and-a-reference-variable-in

Pointers vs References

A pointer needs to be dereferenced with * to access the memory location it points to, whereas a reference can be used directly. A pointer to a class/struct uses -> to access it's members whereas a reference uses a .

References cannot be stuffed into an array, whereas pointers can be

http://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-and-a-reference-variable-in

NUTHIN’ MUCH ABOUT NULL

Dereferencing null pointers

- Caution!

```
int *ptr;
cout << *ptr << endl;
```

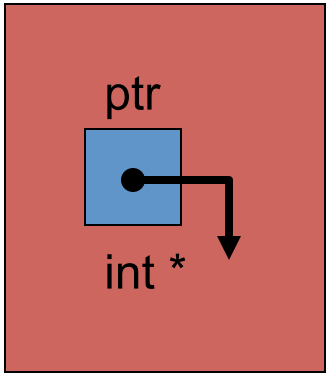
NULL

- NULL is a pointer to **NOTHING**!

```
ptr = NULL; // NULL is a pointer to
      // address 0 (on most compilers)
```

- What is in *ptr?

  nothing. Cannot assign/print/access.



- To reference NULL you need:

  #include <cstddef>

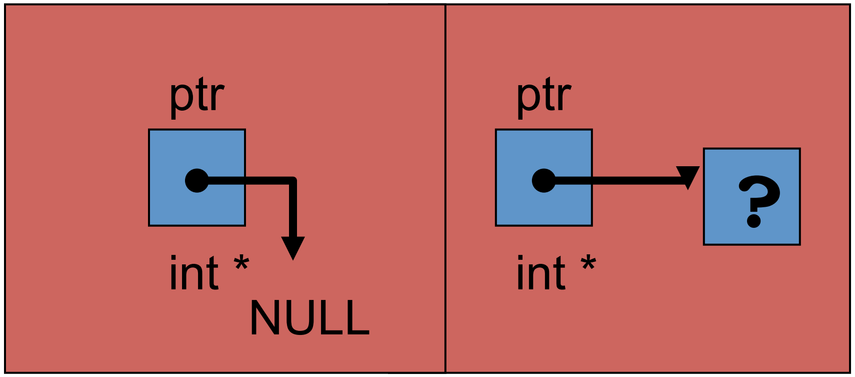
- Assign primitive data types to NULL

  int* x = NULL;   double* x = NULL;

NULL

- Difference between
  - uninitialized
  - initialized to NULL

```
int* ptr = NULL    int* ptr;
```



An uninitialized pointer could have any value in the allocated spot in memory… could point anywhere.

Initialize to NULL so you can check in your program:

COMPARING POINTERS

Comparing Pointers

- Relational operators can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2)     // compares
                      // addresses

if (*ptr1 == *ptr2)   // compares
                      // contents
```

POINTER MATH

Pointer Math

- pointer + int *gives you a* pointer
  - It moves the pointer forward
    - How much depends on the type of the pointer
- Example

```
int array[5], *ptr;
ptr = &array[2];
ptr = ptr + 2;
```

The thing ptr points to is 4 bytes… So ptr + 1 will move you forward 4 bytes in memory.

So, ptr + 2 will give you a pointer to the int two cells after ptr.

Pointer Math

- pointer - int *gives you a* pointer
  - It moves the pointer backward
- pointer - pointer *gives you a* int
  - Gives you the difference between the memory cells
- pointer + pointer … not valid
- same with * / % etc…

Pointer Math

- int i, a[100];
- Arrays
  - Points to the beginning of the cells
  - All three of these are the ***same***:
    - a &a &a[0]

- a[i] is equivalent to *(a+i)

Pointers

Pointers to objects must, similarly be dereferenced:

```
Complex z( 3, 4 );
Complex *pz;
pz = &z;
cout << z.abs() << endl;
cout << (*pz).abs() << endl;
```

Exercise

- Create an array of structs
  - Shallow copy, and
  - Deep copy
- How is it different if it is an array of pointers to structs?

QUESTIONS TO PONDER

Reference Variables

- Avoid the cost of copying
  E.g.

```
string x = findMax(a);
string &y = x;
cout << y << endl;
```

What does memory look like?

How to change to avoid the copy?

Questions

1. What happens when you de-reference a pointer that is pointing to NULL?
2. What happens when you de-reference a pointer that has not yet been initialized?
3. What happens if you de-reference a variable that is not a pointer?
4. Which of the following are valid? Assume pt is a pointer.
   a) pt = &45
   b) pt = &(miles+10)
   c) pt = &miles + 10
5. Which of the following are valid?

   ```
   int nums[25];
   int *pt;
   ```

   a) pt = &nums
   b) pt = nums;
   c) pt = *nums;