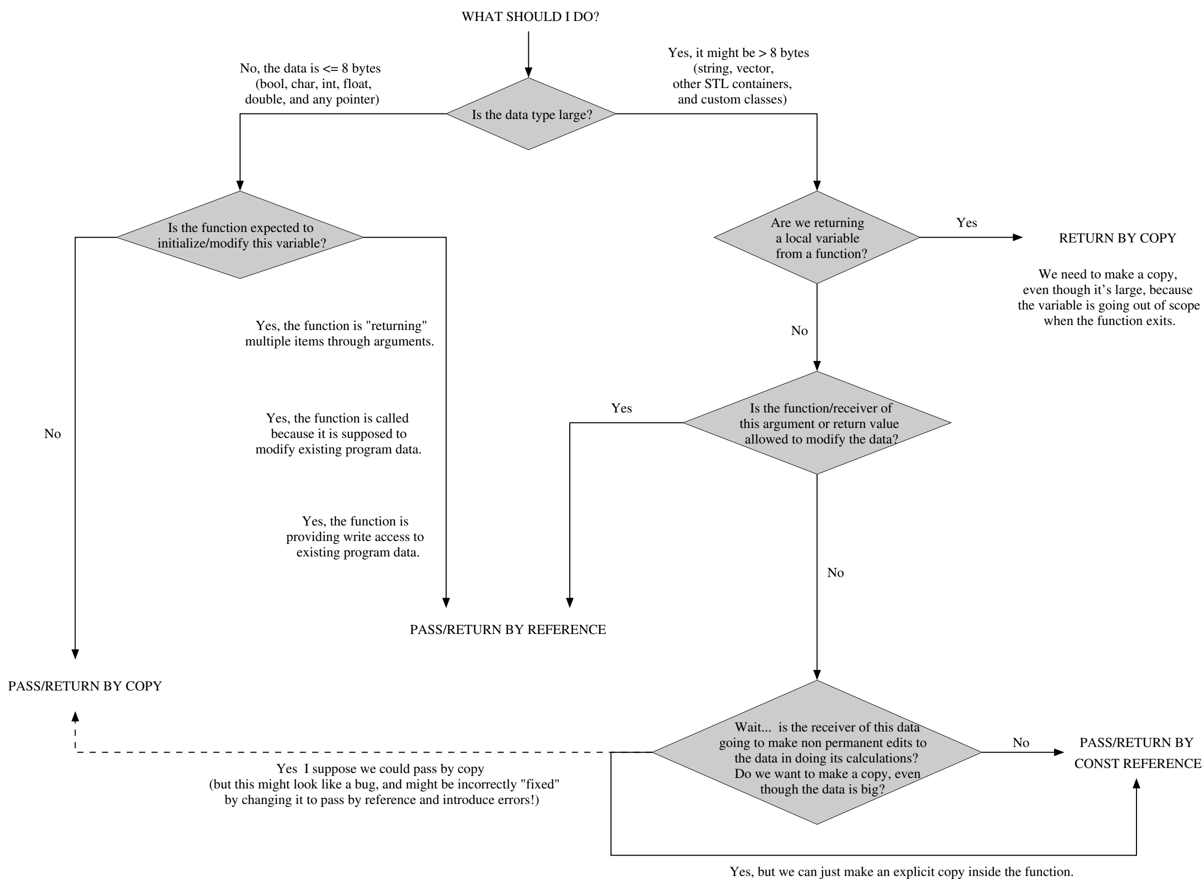
# CSCI-1200 Data Structures — Fall 2022
# Lecture 4 — Pointers, Arrays, & Pointer Arithmetic

## Review from Lectures 2 & 3

- C++ class syntax, designing classes, classes vs. structs;
- Passing comparison functions to `sort`; Non-member operators.
- More fun with `const` and reference (the '`&`')
- Lots of (Questionably Important) C++ Syntax

## FLOWCHART: Use of const and reference with functions / parameters / return types

```
___ std::string ___ FOO::my_function (___ int ___ a,

                                      ___ std::vector<int> ___ b,

                                      ___ Bar ___ c,

                                      ___ Foo* ___ d) ___ {


    ??????

}
```



## Today's Lecture — Pointers and Arrays

- Pointers store memory addresses.
- They can be used to access the values stored at their stored memory address.
- They can be incremented, decremented, added, and subtracted.
- Pointers are an alternative way to access the elements of C-Style array.
- Dynamic memory (covered in Lecture 5) is accessed through pointers.
- Pointers are also the primitive mechanism underlying `vector` iterators, which we have used with `std::sort` and will use more extensively throughout the semester.
- Today we'll start drawing pictures of computer memory and data!
  - This is an important skill in this course *and* for any future programming you do with data structures and/or low-level, detailed computer memory manipulation.
  - You will practice memory diagramming for communication with your peers and your TAs & mentors in lab and in office hours.
  - Your ability to both understand memory diagrams and to prepare new diagrams will be tested during the exams.

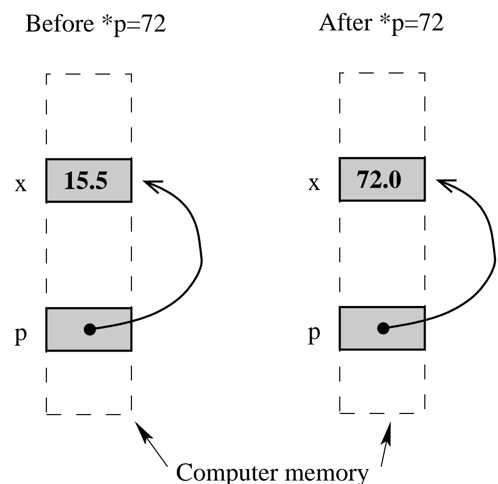
### 4.1 Pointer Example

- Consider the following code segment:

```
float x = 15.5;
float *p; /* equiv:  float* p;  or  float * p;  */
p = &x;
*p = 72;
if ( x > 20 )
  cout << "Bigger\n";
else
  cout << "Smaller\n";
```

The output is `Bigger`

because `x == 72.0`. What's going on?




### 4.2 Pointer Variables and Memory Access

- `x` is an ordinary float, but `p` is a pointer that can hold the memory address of a float variable. The difference is explained in the picture above.
- Every variable is attached to a location in memory. This is where the value of that variable is stored. Hence, we draw a picture with the variable name next to a box that represents the memory location.
- Each memory location also has an address, which is itself just an index into the giant array that is the computer memory.
- The value stored in a pointer variable is an address in memory. The statement `p = &x;` takes the address of `x`'s memory location and stores it (the address) in the memory location associated with `p`.
- Since the value of this address is much less important than the fact that the address is `x`'s memory location, we depict the address with an arrow.
- The statement: `*p = 72;` causes the computer to get the memory location stored at `p`, then go to that memory location, and store 72 there. This writes the 72 in `x`'s location.

  Note: `*p` is an *l-value* in the above expression.

## 4.3 Defining Pointer Variables

- In the example below, `p`, `s` and `t` are all pointer variables (pointers, for short), but `q` is NOT. You need the * before each variable name.

```
int * p, q;
float *s, *t;
```

- There is no initialization of pointer variables in this two-line sequence, so the statement below is dangerous, and may cause your program to crash! (It won't crash if the uninitialized value happens to be a legal address.)

```
*p = 15;
```

## 4.4 Operations on Pointers

- The unary (single argument/operand) operator * in the expression `*p` is the "dereferencing operator". It means "follow the pointer" `*p` can be either an l-value or an r-value, depending on which side of the `=` it appears on.
- The unary operator & in the expression `&x` means "take the memory address of."
- Pointers can be assigned. This just copies memory addresses as though they were values (which they are). Let's work through the example below (and draw a picture!). What are the values of `x` and `y` at the end?

```
float x=5, y=9;
float *p = &x, *q = &y;
*p = 17.0;
*q = *p;
q = p;
*q = 13.0;
```

- Assignments of integers or floats to pointers and assignments mixing pointers of different types are illegal. Continuing with the above example:

```
int *r;
r = q;      //  Illegal: different pointer types;
p = 35.1;   //  Illegal: float assigned to a pointer
```

- Comparisons between pointers of the form `if ( p == q )` or `if ( p != q )` are legal and very useful! Less than and greater than comparisons are also allowed. These are useful only when the pointers are to locations within an array.

## 4.5 Exercise

- Draw a picture for the following code sequence. What is the output to the screen?

```
int x = 10, y = 15;
int *a = &x;
cout << x << " " << y << endl;
int *b = &y;
*a = x * *b;
cout << x << " " << y << endl;
int *c = b;
*c = 25;
cout << x << " " << y << endl;
```

## 4.6 Null Pointers

- Like the `int` type, pointers are not default initialized. We should assume it's a garbage value, leftover from the previous user of that memory.
- Pointers that don't (yet) point anywhere useful are often explicitly assigned to `NULL`.
  - `NULL` is equal to the integer 0, which is a legal pointer value (you can store `NULL` in a pointer variable).
  - But `NULL` is not a valid memory location you are allowed to read or write. If you try to dereference or *follow a* `NULL` *pointer*, your program will immediately crash. You may see a segmentation fault, a bus error, or something about a null pointer dereference.
  - *NOTE:* In C++11, we are encouraged to switch to use `nullptr` instead of `NULL` or 0, to avoid some subtle situations where `NULL` is incorrectly seen as an `int` type instead of a pointer. For this course we will assume `NULL` and `nullptr` are equivalent.
  - We indicate a `NULL` or `nullptr` value in diagrams with a slash through the memory location box.
- Comparing a pointer to NULL is very useful. It can be used to indicate whether or not a pointer variable is pointing at a useable memory location. For example,

  ```
  if ( p != NULL )
    cout << *p << endl.
  ```

  tests to see if `p` is pointing somewhere that appears to be useful before accessing and printing the value stored at that location.
- But don't make the mistake of assuming pointers are automatically initialized to NULL.

## 4.7 C-Style Arrays and Subscripting

- Here's a short fragment of code using a C-Style array (not an STL `vector`).
- This example uses subscripting, `[ ]`, to access each slot of the array:

  ```
  const int n = 10;
  double a[n];
  int i;
  for ( i=0; i<n; ++i )
      a[i] = sqrt( double(i) );
  ```

- Remember: the size of array `a` is fixed at compile time. STL `vector`s act like arrays, but they can grow and shrink dynamically in response to the demands of the application.

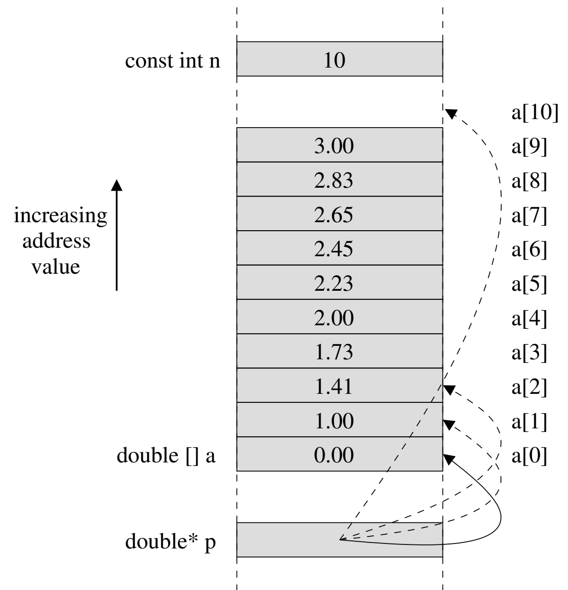
## 4.8 Stepping through Arrays with Pointers (Array *Iterators*)

- The array code above can be equivalently rewritten to instead use pointers:

  ```
  const int n = 10;
  double a[n];
  double *p;
  for ( p=a; p<a+n; ++p )
     *p = sqrt( p-a );
  ```

- The assignment: `p = a;` takes the address of the start of the array and assigns it to `p`.
- This illustrates the important fact that the name of an array is in fact **a pointer to the start of a block of memory**. We will come back to this several times! We could also write this line as: `p = &a[0];` which means "find the location of `a[0]` and take its address".

- By incrementing, `++p`, we make `p` point to the next location in the array.
  - When we increment a pointer we don't just add one byte to the address, we add the number of bytes (*sizeof*) used to store one object of the specific type of that pointer. Similarly, basic addition/subtraction of pointer variables is done in multiples of the *sizeof* the type of the pointer.
  - Since the type of `p` is `double`, and the size of `double` is 8 bytes, we are actually adding 8 bytes to the address when we execute `++p`.
- The test `p<a+n` checks to see if the value of the pointer (the address) is less than $n$ array locations beyond the start of the array.
  In this example, `a+n` is the memory location 80 bytes after the start of the array (n = 10 slots * 8 bytes per slot).
  We could equivalently have used the test `p != a+n`
- In the assignment:
  `*p = sqrt( p-a )`
  `p-a` is the number of array locations (multiples of 8 bytes) between $p$ and the start. This is an integer. The square root of this value is assigned to `*p`.
- Note that there may or may not be unused memory between your array and the other local variables. Similarly, the order that your local variables appear on the stack is not guaranteed (the compiler may rearrange things a bit in an attempt to optimize performance or memory usage). A *buffer overflow* (attempting to access an illegal array index) may or may not cause an immediate failure – depending on the layout of other critical program memory.




## 4.9 Diagramming Instances of a Custom Class & Arrays of Instances

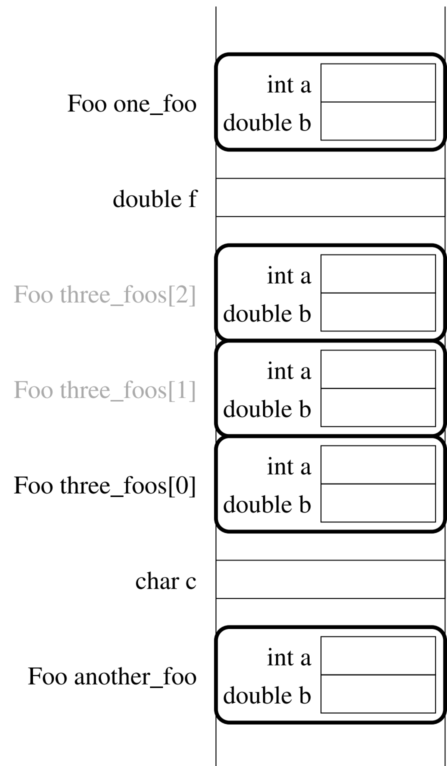
- Let's look at a more complicated array example... an array of custom objects!
- How does the compiler know how much space will be needed for each instance of user-defined class? It just adds the size of each of the class member variables (rounding up for better memory alignment).
- When we create a fixed-size array of objects, we call an appropriate constructor for every slot of that structure.
- NOTE: *static* memory allows a sort of "better" global variable. The value persists between different calls to the function, but it is only accessible from that scope (so it's less dangerous to use).
- Let's draw a detailed memory diagram for this small program in detail and study the corresponding program output.

```
class Foo {
public:
  Foo();
  double value() const { return a*b; }
private:
  int a;
  double b;
};

Foo::Foo() {
  static int counter = 1;
  a = counter;
  b = 100.0;
  counter++;
}
```

```
int main() {
  std::cout << "size of int: " << sizeof(int) << std::endl;
  std::cout << "size of double: " << sizeof(double) << std::endl;
  std::cout << "size of char: " << sizeof(char) << std::endl;
  std::cout << "size of foo object: " << sizeof(Foo) << std::endl;

  Foo one_foo;
  double f = 3.14159;
  Foo three_foos[3];
  char c = 'X';
  Foo another_foo;
  std::cout << "foo values: " << another_foo.value()
            << " & " << three_foos[1].value() << std::endl;
}
```

Program Output:

```
size of int: 4
size of double: 8
size of char: 1
size of foo object: 16
foo values: 500 & 300
```




## 4.10 Sorting an Array

- Arrays may be sorted using `std::sort`, just like `vector`s. Pointers are used in place of iterators. For example, if `a` is an array of doubles and there are `n` values in the array, then here's how to sort the values in the array into increasing order:

```
std::sort( a, a+n );
```

## 4.11 Exercises

To practice pointers and pointer arithmetic, complete the following exercises using pointers and not subscripting. In other words, don't use the square brackets (`[ ]`) to access slots of the array!

1. Write code to print the array `a` backwards, using pointers.

2. Write code to print every other value of the array `a`, again using pointers.

3. Write a function that checks whether the contents of an array of doubles are sorted into increasing order. The function should take in two arguments: a pointer (to the start of the array), and an integer indicating the size of the array.

## 4.12 C Calling Convention – *Bonus Topic!*

- We take for granted the non-trivial task of passing data to a helper function, getting data back from that function, and seamlessly continuing on with the program. *How does that work??*
- A *calling convention* is a standardized method for passing arguments between the caller and the function. Calling conventions vary between programming languages, compilers, and computer hardware.
- In C on x86 architectures here is a generalization of what happens:
  1. The caller puts all the arguments on the *stack*, in reverse order.
  2. The caller puts the address of its code on the stack (the *return address*).
  3. Control is transferred to the callee.
  4. The callee puts any local variables on the stack.
  5. The callee does its work and puts the return value in a special *register* (storage location).
  6. The callee removes its local variables from the stack.
  7. Control is transferred by removing the address of the caller from the stack and going there.
  8. The caller removes the arguments from the stack.
- On x86 architectures the addresses on the stack are in descending order. This is not true of all hardware.

## 4.13 Poking around in the Stack & Looking for the C Calling Convention

- Let's look more closely at an example of where the compiler stores our data. Specifically, let's print out the addresses and values of the local variables and function parameters:

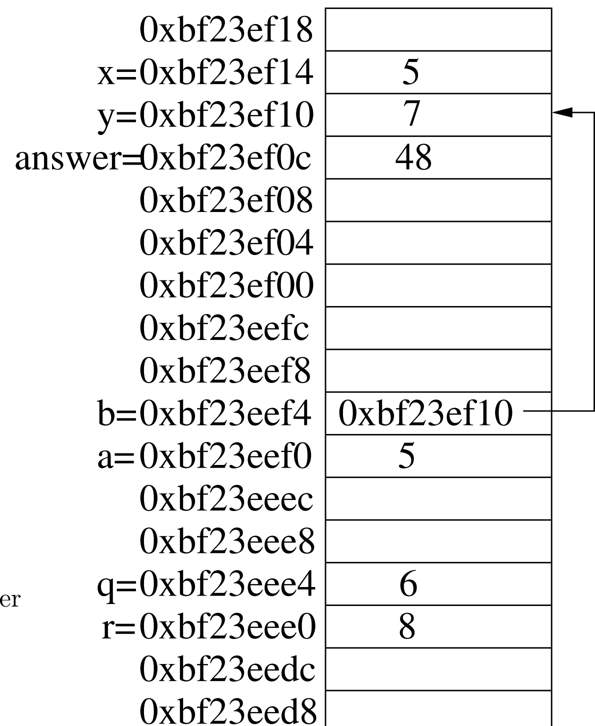
```
int foo(int a, int *b) {
  int q = a+1;
  int r = *b+1;
  std::cout << "address of a = " << &a << std::endl;
  std::cout << "address of b = " << &b << std::endl;
  std::cout << "address of q = " << &q << std::endl;
  std::cout << "address of r = " << &r << std::endl;
  std::cout << "value at " << &a << " = " << a << std::endl;
  std::cout << "value at " << &b << " = " << b << std::endl;
  std::cout << "value at " << b << " = " << *b << std::endl;
  std::cout << "value at " << &q << " = " << q << std::endl;
  std::cout << "value at " << &r << " = " << r << std::endl;
  return q*r;
}

int main() {
  int x = 5;
  int y = 7;
  int answer = foo (x, &y);
  std::cout << "address of x = " << &x << std::endl;
  std::cout << "address of y = " << &y << std::endl;
  std::cout << "address of answer = " << &answer << std::endl;
  std::cout << "value at " << &x << " = " << x << std::endl;
  std::cout << "value at " << &y << " = " << y << std::endl;
  std::cout << "value at " << &answer << " = " << answer << std::endl;
}
```

- Note that the first function parameters is regular integer, passed by copy. The second parameter is a passed in as a pointer.

- Note that we can print out data values or pointers – the address is printed as a big integer in hexadecimal format (beginning with "`0x`"). This program was compiled as 32-bit program, so our addresses are 32-bits. A 64-bit program will have longer addresses.

- Let's look at the program output and reverse engineer a drawing of the stack:

```
address of a = 0xbf23eef0
address of b = 0xbf23eef4
address of q = 0xbf23eee4
address of r = 0xbf23eee0
value at 0xbf23eef0 = 5
value at 0xbf23eef4 = 0xbf23ef10
value at 0xbf23ef10 = 7
value at 0xbf23eee4 = 6
value at 0xbf23eee0 = 8
address of x = 0xbf23ef14
address of y = 0xbf23ef10
address of answer = 0xbf23ef0c
value at 0xbf23ef14 = 5
value at 0xbf23ef10 = 7
value at 0xbf23ef0c = 48
```




- Note: The unlabeled portions in our diagram of the stack will include the frame pointer, the return address, temp variables (complex C++ expressions turn into many smaller steps of assembly), space to save registers, and padding between variables to meet alignment requirements.

- Note: Different compilers and/or different optimization levels will produce a different stack diagram.