CS 3460

Introduction to Dynamic Memory

Stack & Heap Review

- Stack
  - The language/compiler provides this concept
  - Growable/shrinkable # of items can be stored on it
  - Used for temporary storage
    - Local variables
    - function parameters
    - other items
  - Upon scope open, a new stack frame is reserved
  - Upon scope close, stack frame is removed

Stack & Heap Review

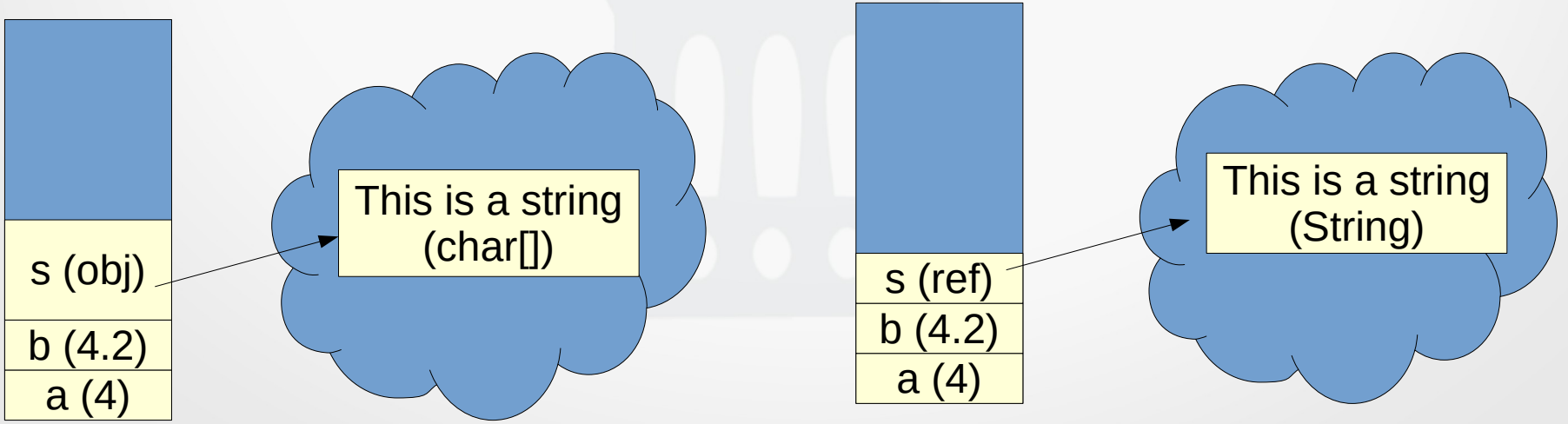
- Heap
  - OS manages all memory on the computer
  - Large section is reserved for the ***heap***
  - When a process requests memory, it is provided from the heap; a heap allocation
  - Lifetime of heap memory is independent of scope
  - When finished, program tells OS and the memory is marked as available

C++ vs Java Stack/Heap

- Variable definitions are stack allocations
- But let's look at how C++ and Java compare

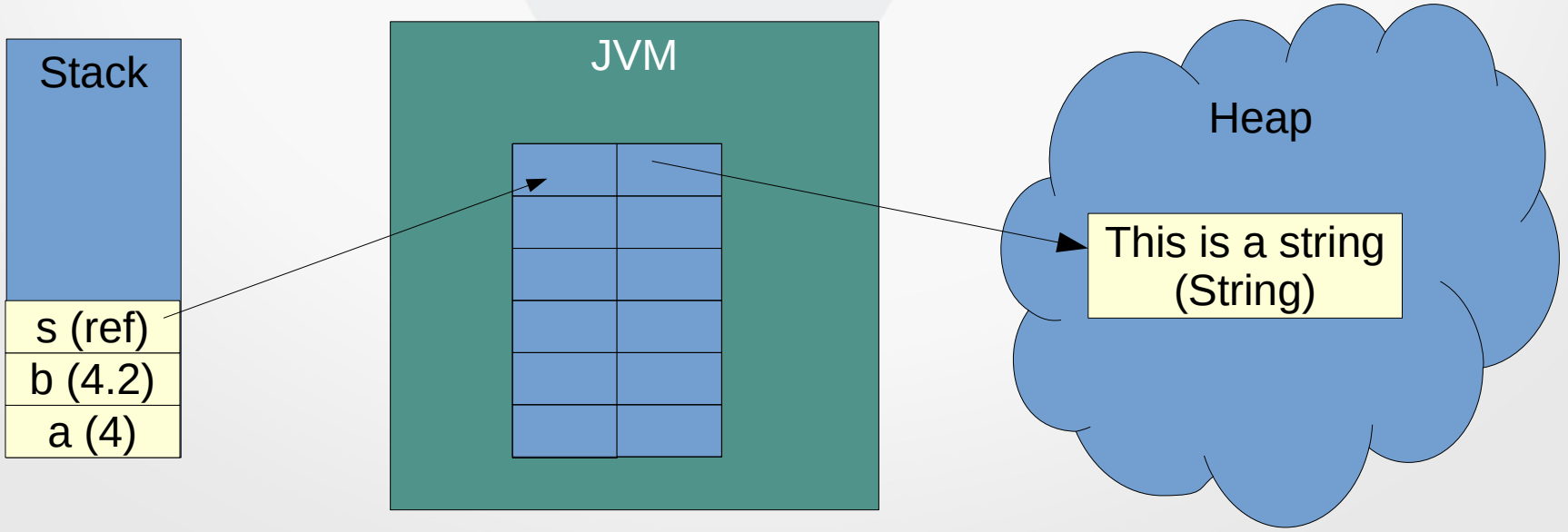
```
int a = 4;
double b = 4.2;
std::string s = "This is a string";
```

```
int a = 4;
double b = 4.2;
String s = "This is a string";
```



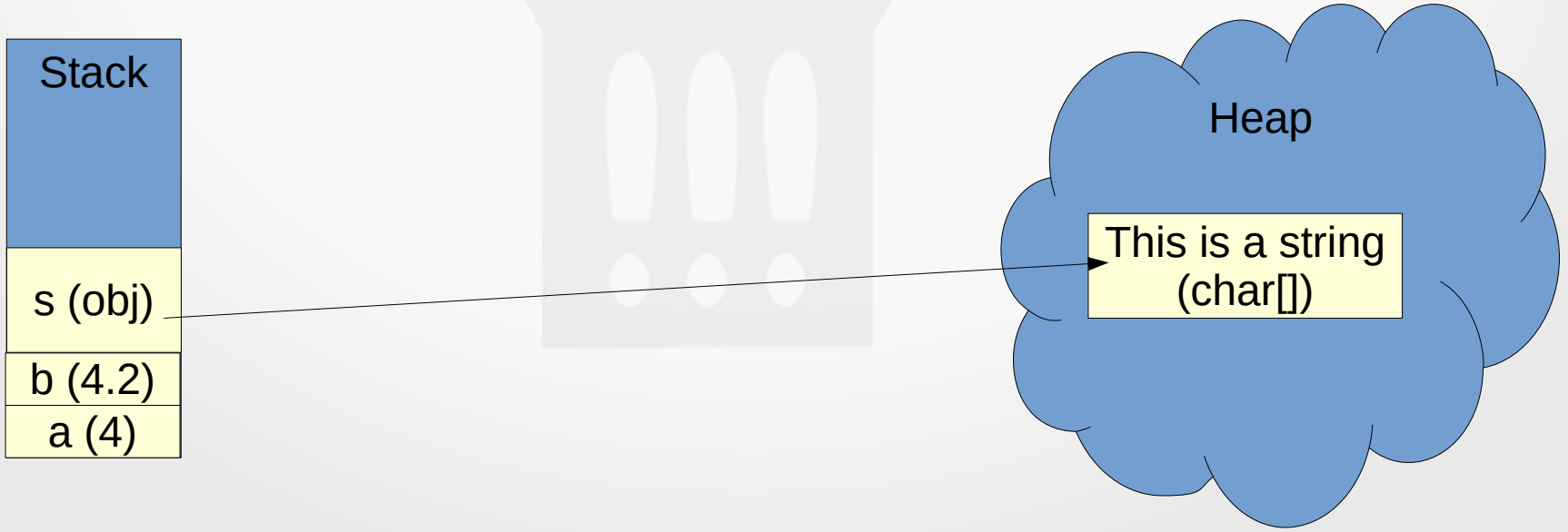
C++ Pointers and Java References

- A C++ ***reference*** has nothing to do with a Java ***reference***
- C++ reference is an alias for a variable
- Java reference stores a value the JVM uses to look up the memory location of heap allocated memory (keyword `new`)
- When reference is no longer used GC reclaims and informs the OS



C++ Pointers and Java References

- C++ uses same `new` keyword for memory allocation
- `new` operator is translated into call to OS, which eventually returns the actual memory location for the storage
- C++ calls the memory location a ***pointer***



C++ Raw Pointers

- Pointer data type is formed by adding an `*`

```
int* a;
```

- Any data type can be turned into a pointer this way
- What about a pointer, can you have a pointer to a pointer?

```
int** a;
```

- Yes!! (stay tuned)

```
int* a = new int(1);
double* b = new double(3.14159);
std::string* c = new std::string("This string is dynamically allocated");
// don't ever dynamically allocate an std::string, this is for
// demonstration purposes only; I'm a highly trained professional
// and know what I'm doing
```

Arrays & Pointers

- The name of a raw array ***is*** a pointer to the first element location

```
int primes[] { 2, 3, 5, 7 };
```

- The value stored in `primes` is the memory location of the first element of the array; but the array is not a heap allocation, it is on the stack

```
int primes[] { 2, 3, 5, 7 };
int* pointerToPrimes = primes;

std::cout << primes[0] << " : " << pointerToPrimes[0] << std::endl;
std::cout << primes[1] << " : " << pointerToPrimes[1] << std::endl;
std::cout << primes[2] << " : " << pointerToPrimes[2] << std::endl;
std::cout << primes[3] << " : " << pointerToPrimes[3] << std::endl;
```

Let's look at this in a code demonstration

Dynamic Allocation of Raw Arrays

- Knowing the name of a raw array is a pointer to the array data location, let's talk about dynamic array allocation

```
int* primes = new int[4];

primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
```

Dynamic Allocation of Raw Arrays

- It is possible to allocate and initialize in one statement

```
int* primes = new int[4]{ 2, 3, 5, 7 };
```

Raw Pointers & Memory Responsibility

- In Java, the JVM periodically invokes the GC to reclaim memory
  - Java program runs in context of the JVM; an additional execution environment, on top of the OS
- No similar thing in C++
  - no additional execution environment, executes directly from the OS
  - Therefore, raw pointer memory is complete responsibility of the C++ program

Raw Pointers & Memory Responsibility

- In a C++ program, if a program loses track of allocated memory, it isn't returned to the OS, instead, it is lost and is what we call a ***memory leak***
  - Reclaimed only when process is terminated
- Returning memory is C++ program responsibility
  - use the `delete` keyword

Raw Pointers & Memory Responsibility

```
int* primes = new int[4];

primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
... do something with the prime numbers ...
delete []primes
```

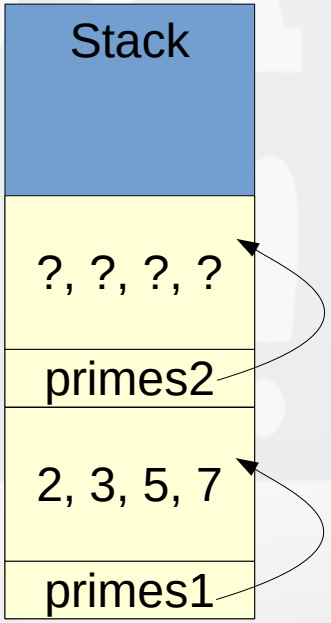
- A couple of items
  - The `[]` are required when deleting an array
  - After `delete`, the value in primes stays the same, it is just invalid now. Can set to `nullptr` to note is doesn't point to anything…

```
primes = nullptr;
```

Arrays: Stack or Heap

- These arrays are stack allocated

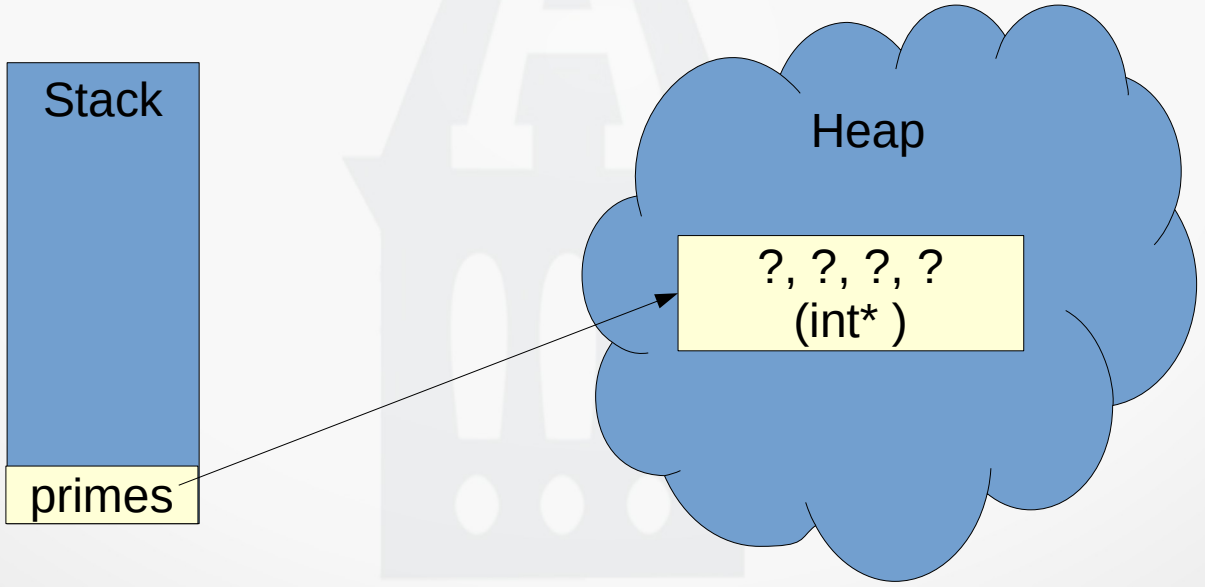
```
int primes1[] { 2, 3, 5, 7 };
int primes2[4];
```



Arrays: Stack or Heap

- This array is heap allocated

```
int* primes = new int[4];
```



Code Demo of Stack/Heap Array Allocations (generate primes function examples)

Using Pointer Variables

- Can obtain the memory address of a variable
  - `&` : *address-of* operator
  - Easily confused with a reference type
    - `&` operator looks like the reference type decorator
- Obtain the value of a memory location
  - `*` : *dereference* operator
    - Looks like the pointer type decorator; I know

Code Demo – Using Pointer Variables

Raw Pointer Arithmetic

- Can perform arithmetic on pointers. Consider the following code...

```
int primes[] { 2, 3, 5, 7 };

std::cout << *(primes + 0) << std::endl;
std::cout << *(primes + 1) << std::endl;
std::cout << *(primes + 2) << std::endl;
std::cout << *(primes + 3) << std::endl;
```

Raw Pointer Arithmetic

- Can display the memory location. Consider the following code…

```
int primes[] { 2, 3, 5, 7 };

std::cout << (primes + 0) << std::endl;
std::cout << (primes + 1) << std::endl;
std::cout << (primes + 2) << std::endl;
std::cout << (primes + 3) << std::endl;
```

- Note the values differ by 4, in each location
  - an `int` is 32 bits (4 bytes)
  - pointer arithmetic of `primes + 1` is
    `(primes + 1 * sizeof(int))`
  - If we change the array type to `short`, the arithmetic is
    `(primes + 1 * sizeof(short))`

Pointers as Function Parameters

- Have already seen how to use references for passing function parameters
- C++ is *pass-by-value*, by default, unless *pass-by-reference* is used
- Pointers can be used as parameters
  - It is still *pass-by-value*, making a copy of the pointer only, not a copy of the memory pointed to
  - Pointers can by passed *by-reference*

Code Demo – Pointers as Function Parameters