Week 6, Lecture 2

July 27, 2022

C++ Primer 2.3

Multiple File Streams to Same File

- Once you open an output stream to a file, the contents are deleted
  - If you had an input stream open before that, but try to read after the file is opened for writing, you will get nothing
- When you print to a file, the contents are not always put there immediately
  - Reading from an output file after writing to it may not yet get the new content, if the output file stream is still open

Pointers

First: Stack and Heap

- Memory for any program is divided into two sections: the Stack and the Heap
- Everything I have said so far about memory is for the Stack
    - Must know size of variables to be stored at compile time
    - Cannot change the size of variables already stored (arrays)
    - Variables are local to the curly braces where they are defined
        - Deleted afterwards
- When we create a variable in any of the ways we've learned so far, it goes on the Stack
    - Except for strings, but those are special

First: Stack and Heap

- Memory for any program is divided into two sections: the Stack and the Heap
- We've seen a few memory addresses on the stack
  - Call by reference uses these to allow a function to change the value of a variable passed to it outside of the function as well
  - If we accidentally print a whole array rather than looping and printing each element, the address of the array is printed
- On the stack, we can use addresses but mostly we use variable names to access variables
- If we want to access the heap, we have to do it by address

First: Stack and Heap

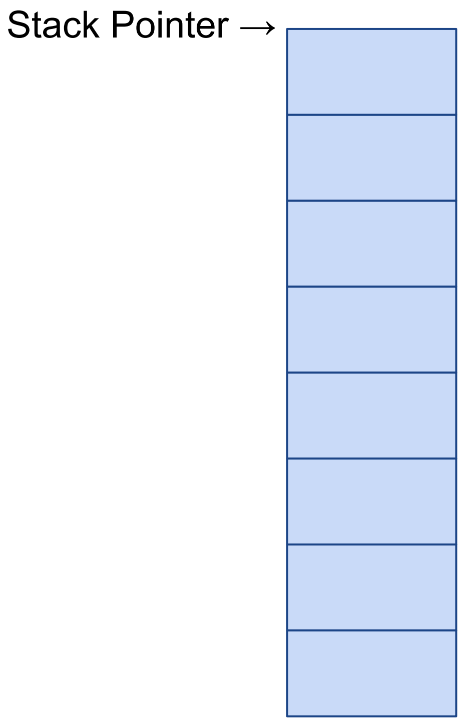
- Memory for any program is divided into two sections: the Stack and the Heap
- The size of variables stored on the heap does not have to be known at compile time
    - We can, for instance, take in the length of an array from user input
- We have to keep track of the address of variables on the heap ourselves, in the form of **pointers**
- A variable on the heap is not deleted at the end of the curly braces where the variable was declared

First: Stack and Heap

- A variable on the heap is not deleted at the end of the curly braces where the variable was declared
  - However, the pointer that is storing that address is deleted
    - We need to make sure that we don't lose access to the variables we have kept on the heap
    - If we delete all pointers to a variable on the heap without deleting the variable itself, we get a **memory leak**
      - The computer will still believe that memory is being used, and prevent your program and other programs from using it
      - Since we do not have the address, we cannot retrieve or delete that variable

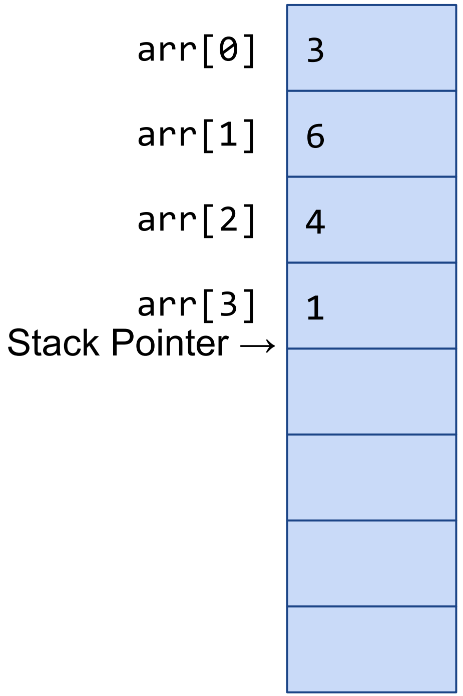
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



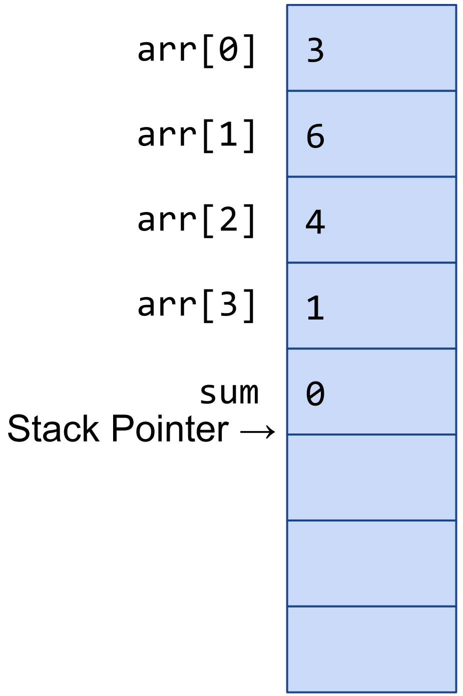
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



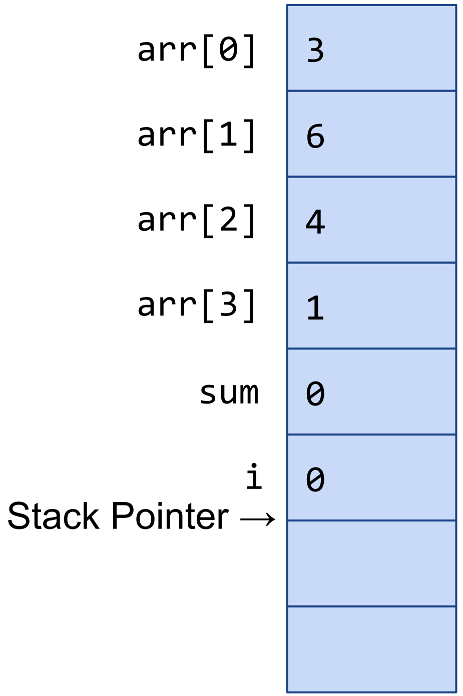
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



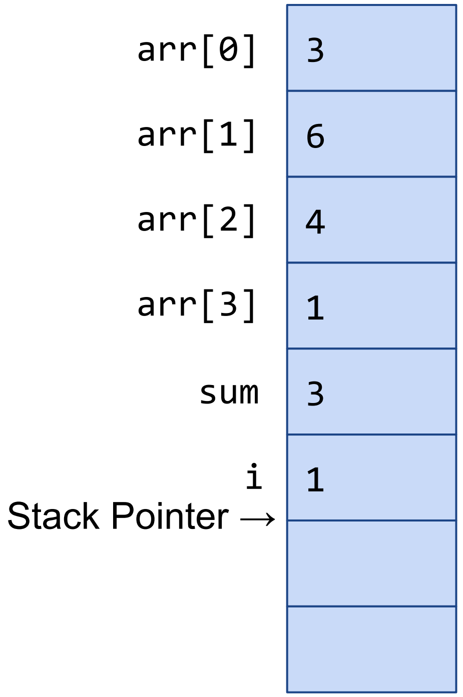
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



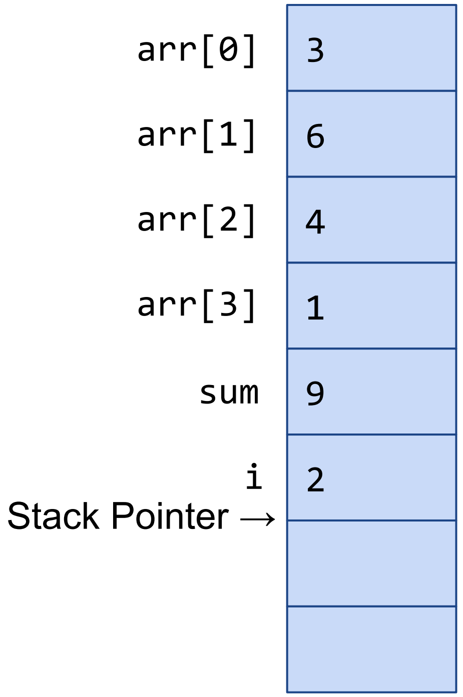
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



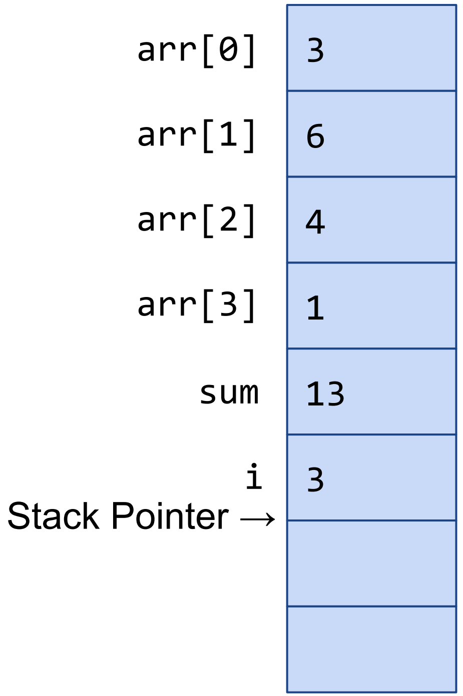
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



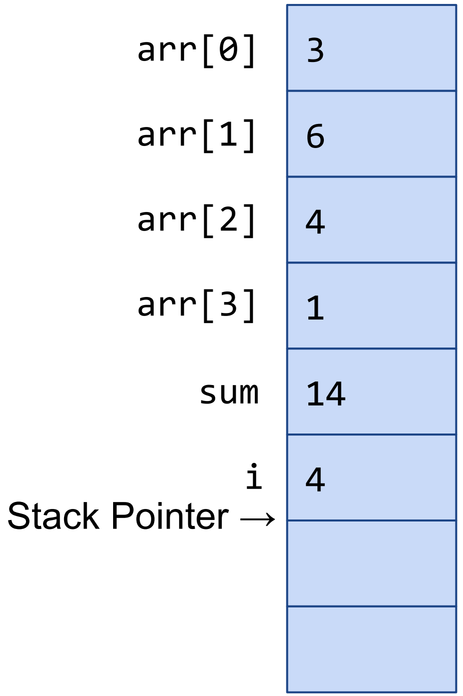
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



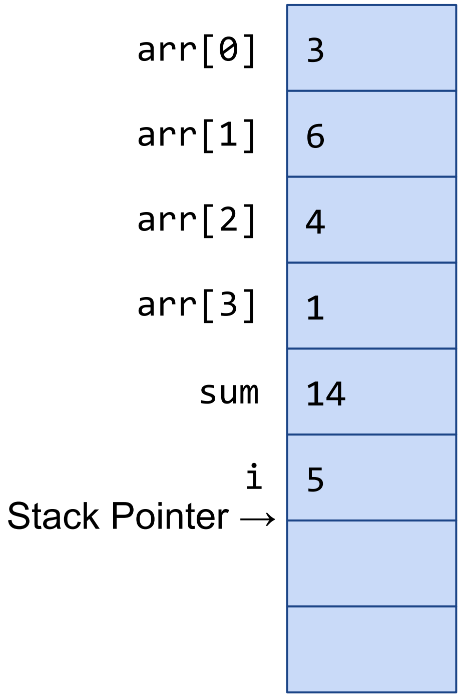
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



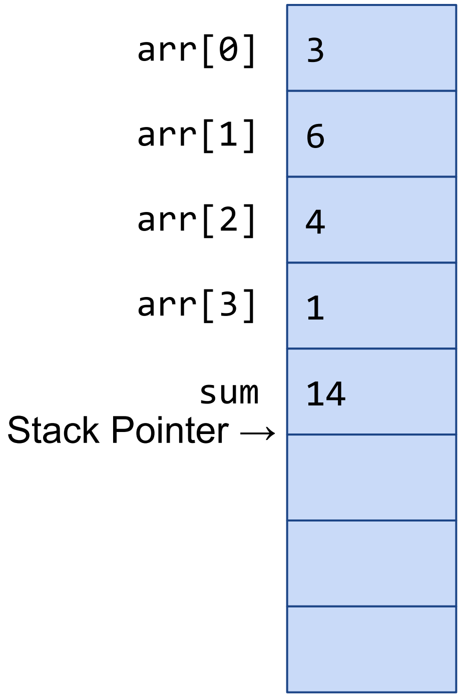
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



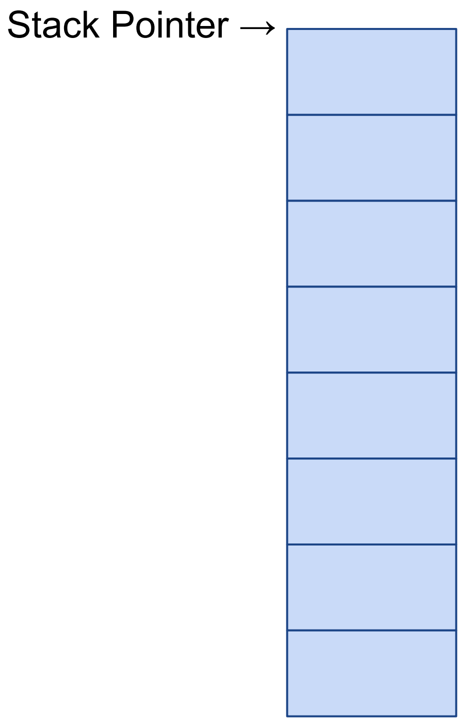
Stack

```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



Stack

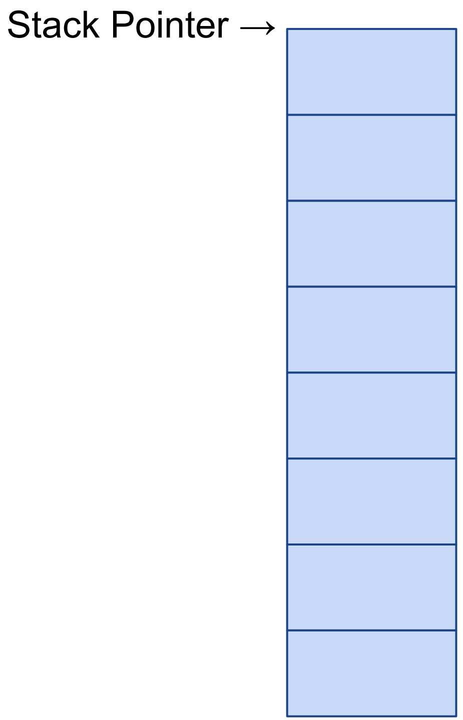
```
int arr[4] = {3, 6, 4, 1};
int sum = 0;
for(int i = 0; i < 4; ++i){
    sum += arr[i];
}
```



Stack

```
int sum(int a, int b, int c){
    int result = a + b + c;
    return result;
}

int main(){
    int x = 10;
    int y = 20;
    int z = 30;
    int res = sum(x, y, z);
}
```

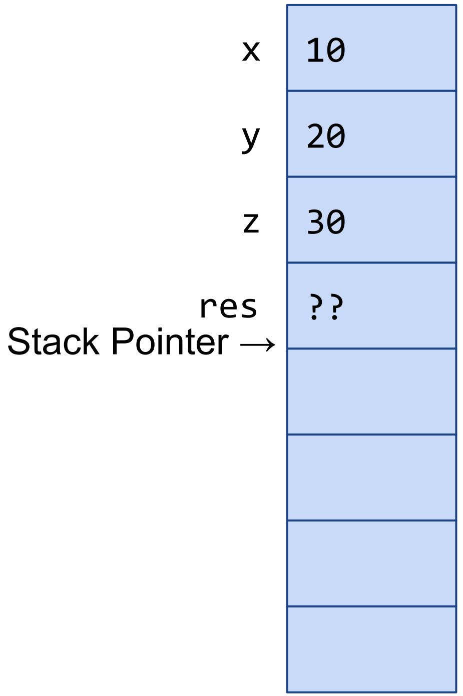


Stack

```
int sum(int a, int b, int c){
    int result = a + b + c;
    return result;
}
```

Within main:

```
int x = 10;
int y = 20;
int z = 30;
int res = sum(x, y, z);
```

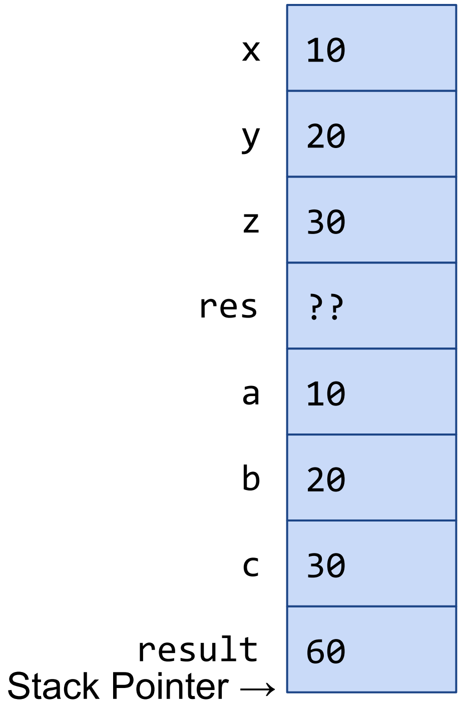


Stack

```
int sum(int a, int b, int c){
    int result = a + b + c;
    return result;
}
```

Within main:

```
int x = 10;
int y = 20;
int z = 30;
int res = sum(x, y, z);
```

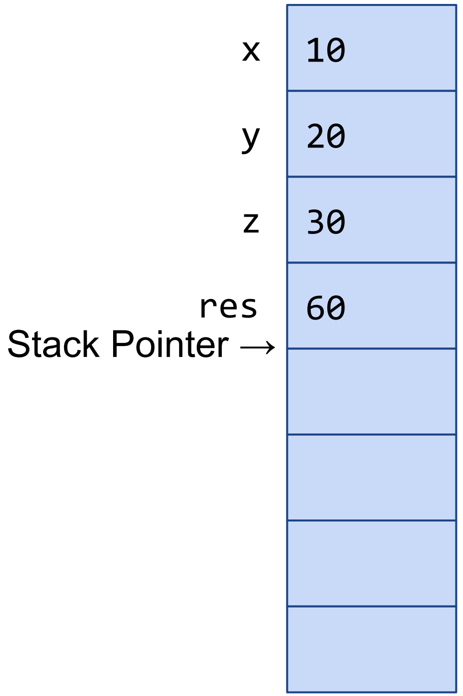


Stack

```
int sum(int a, int b, int c){
    int result = a + b + c;
    return result;
}
```

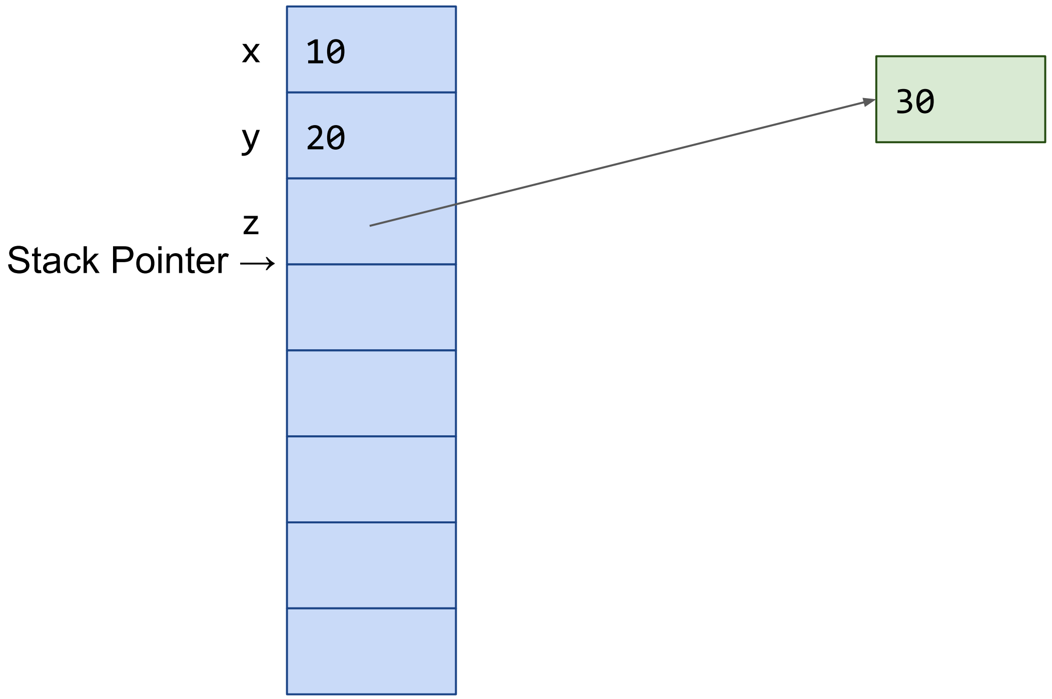
Within main:

```
int x = 10;
int y = 20;
int z = 30;
int res = sum(x, y, z);
```



Pointers from the Stack to the Heap

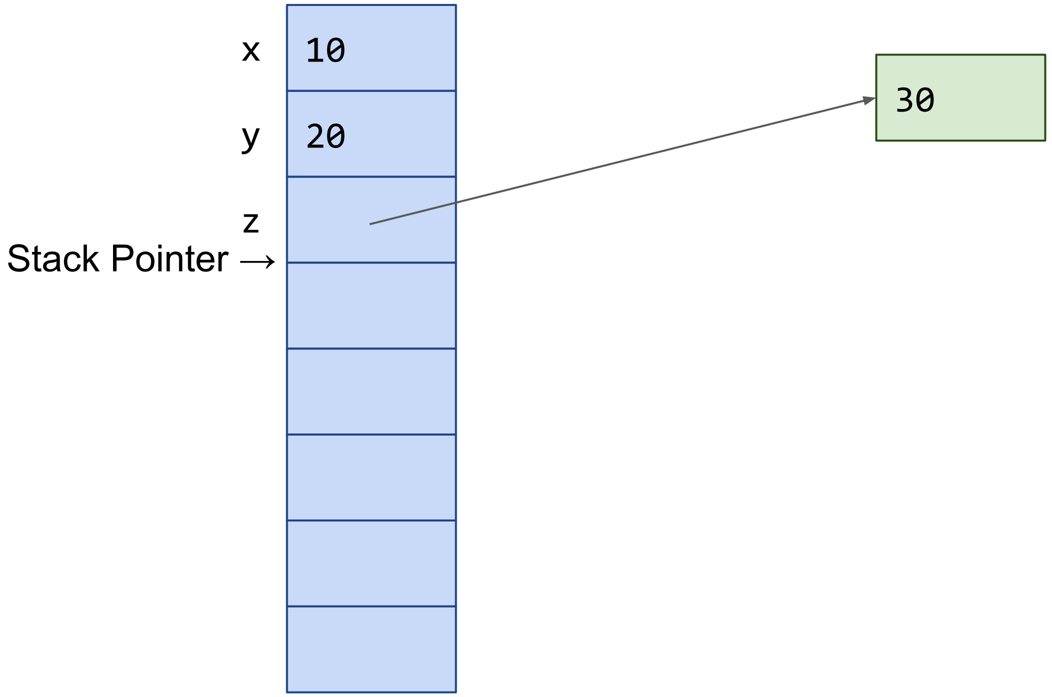
```
int x = 10;
int y = 20;
int* z = new int(30);
```



Pointers from the Stack to the Heap

```
int x = 10;
int y = 20;
int* z = new int(30);
```

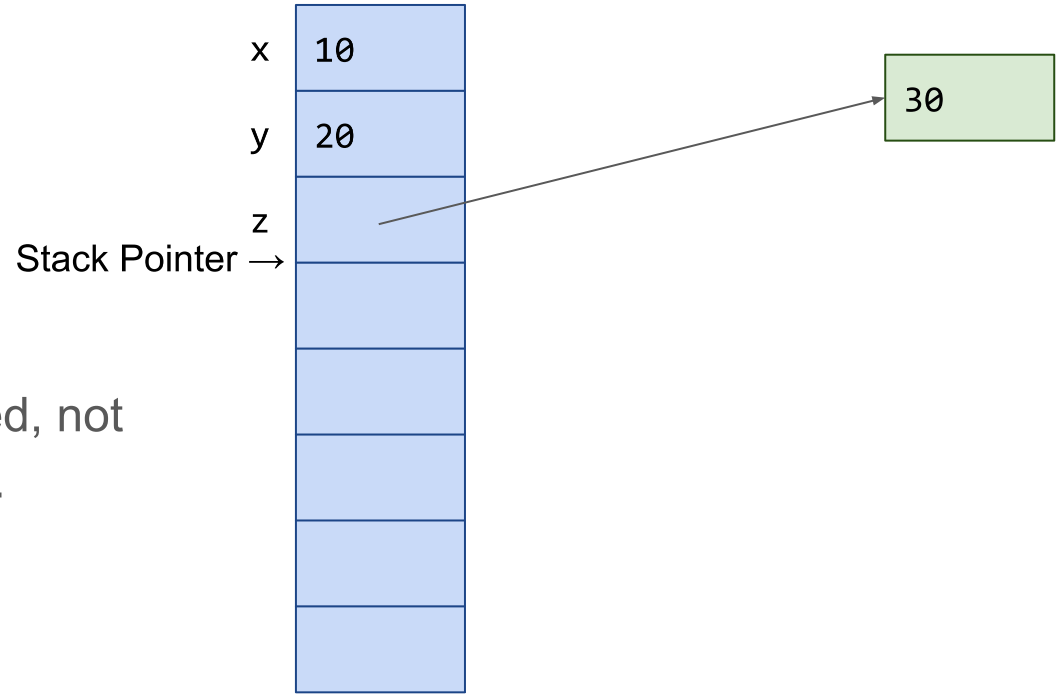
The thing we are pointing to is an integer



Pointers from the Stack to the Heap

```
int x = 10;
int y = 20;
int* z = new int(30);
```

The asterisk indicates that this is a pointer to a variable of the type specified, not a variable of that type itself.

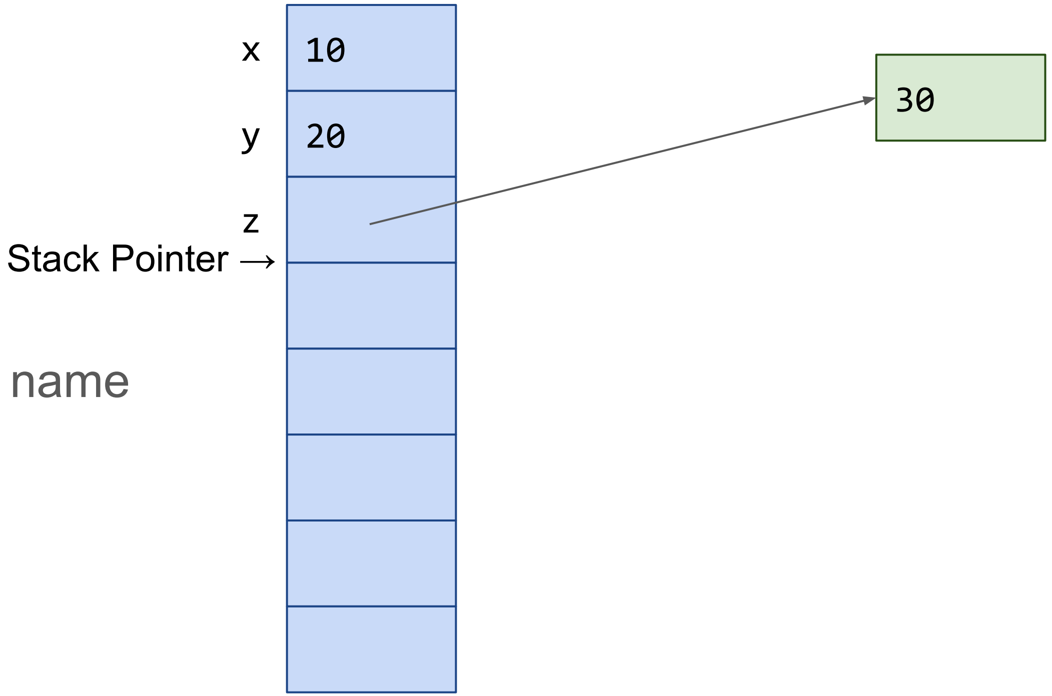


Pointers from the Stack to the Heap

```
int x = 10;
int y = 20;
int* z = new int(30);
```

As usual, the name.

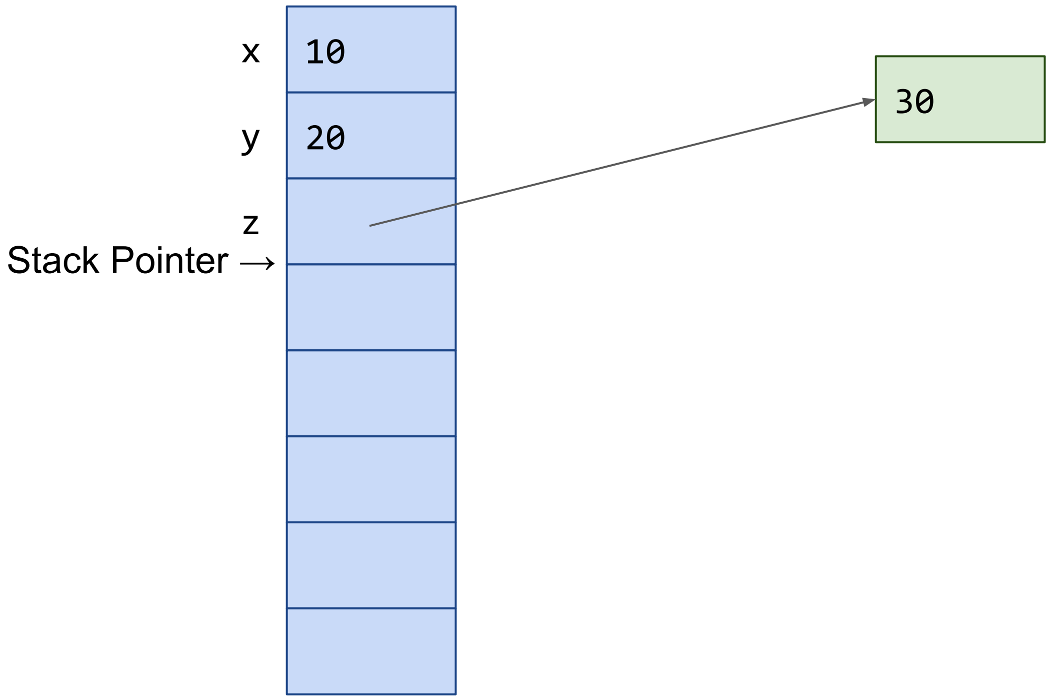
This time though, this is the name of the pointer variable.



Pointers from the Stack to the Heap

```
int x = 10;
int y = 20;
int* z = new int(30);
```

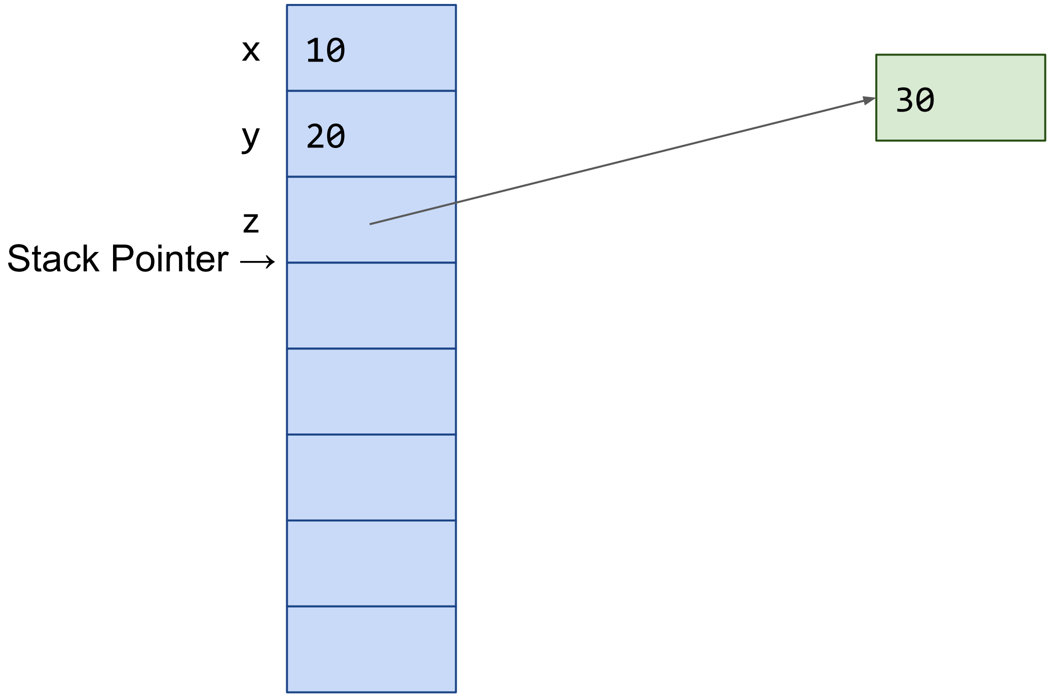
"new" tells the compiler that this is a variable on the Heap, not the Stack.



Pointers from the Stack to the Heap

```
int x = 10;
int y = 20;
int* z = new int(30);
```

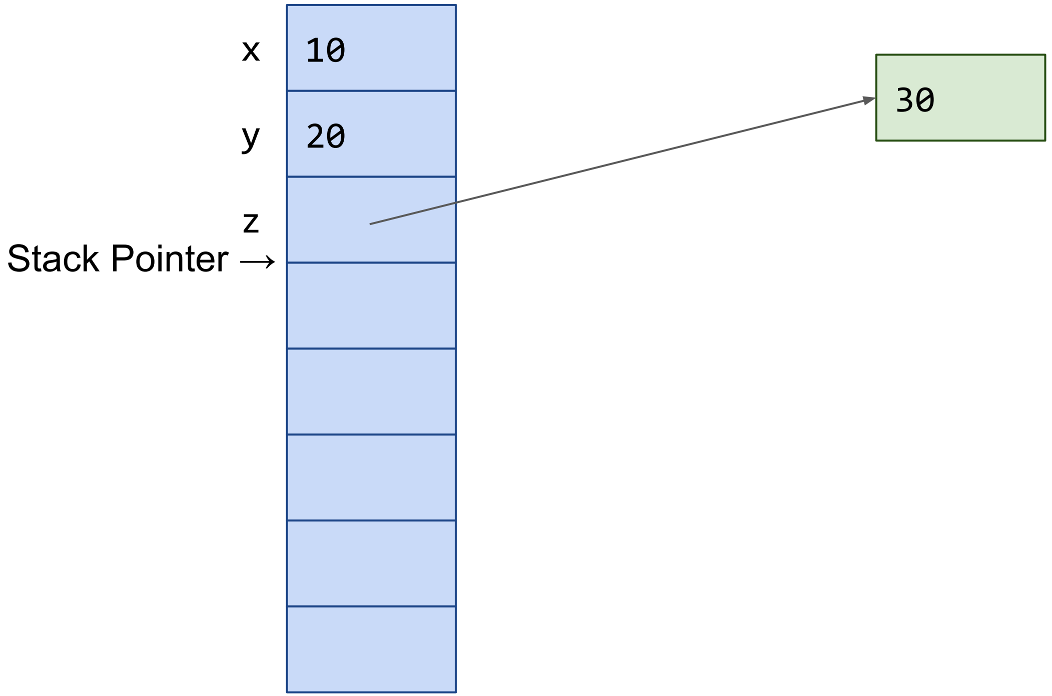
This is necessary to tell the compiler which type of new variable to make.



Pointers from the Stack to the Heap

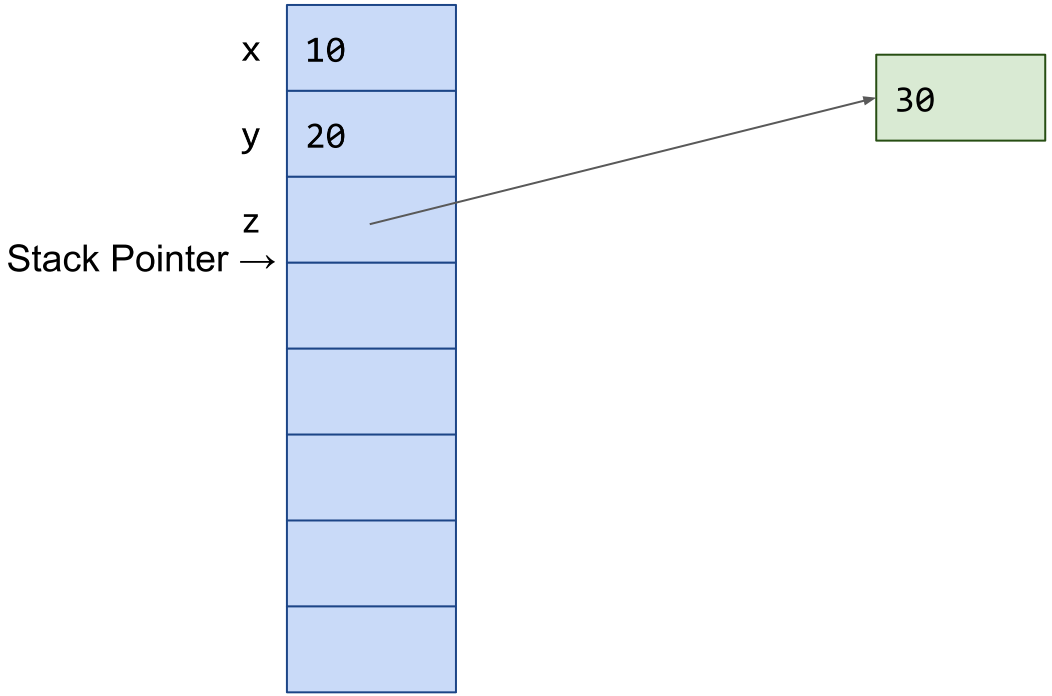
```
int x = 10;
int y = 20;
int* z = new int(30);
```

Just slightly different syntax to initialize an int with a value.



Another Option

```
int x = 10;
int y = 20;
int* z = new int;
*z = 30;
```

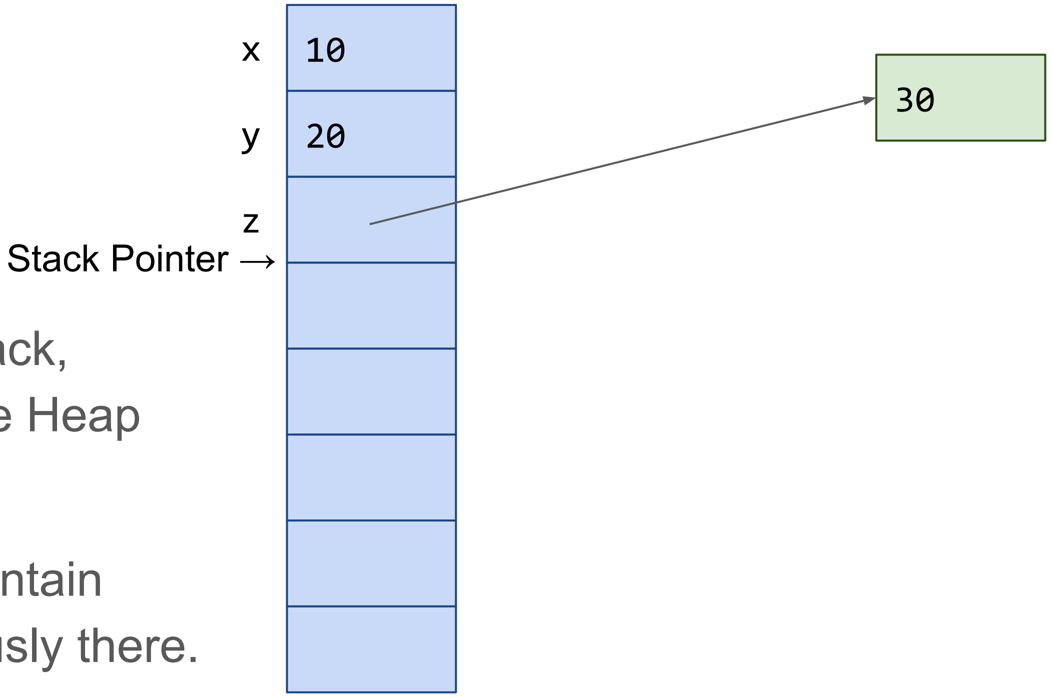


Another Option

```
int x = 10;
int y = 20;
int* z = new int;
*z = 30;
```

Just the same as on the Stack,
we can declare an int on the Heap
and initialize it later.

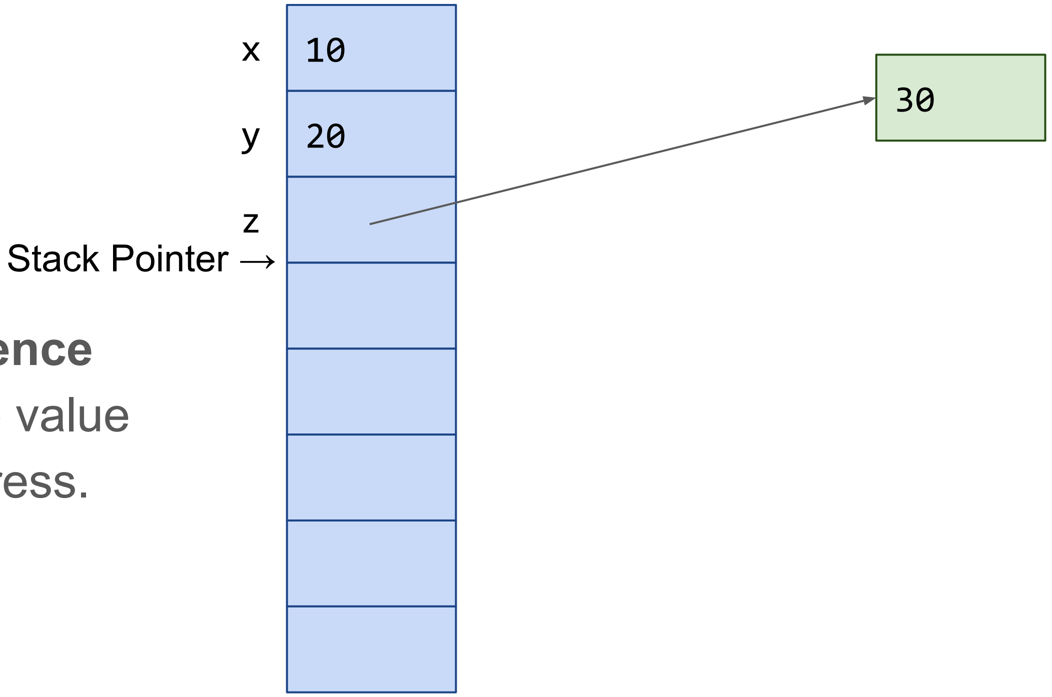
Until it is initialized, it will contain
whichever bits were previously there.



Another Option

```
int x = 10;
int y = 20;
int* z = new int;
*z = 30;
```

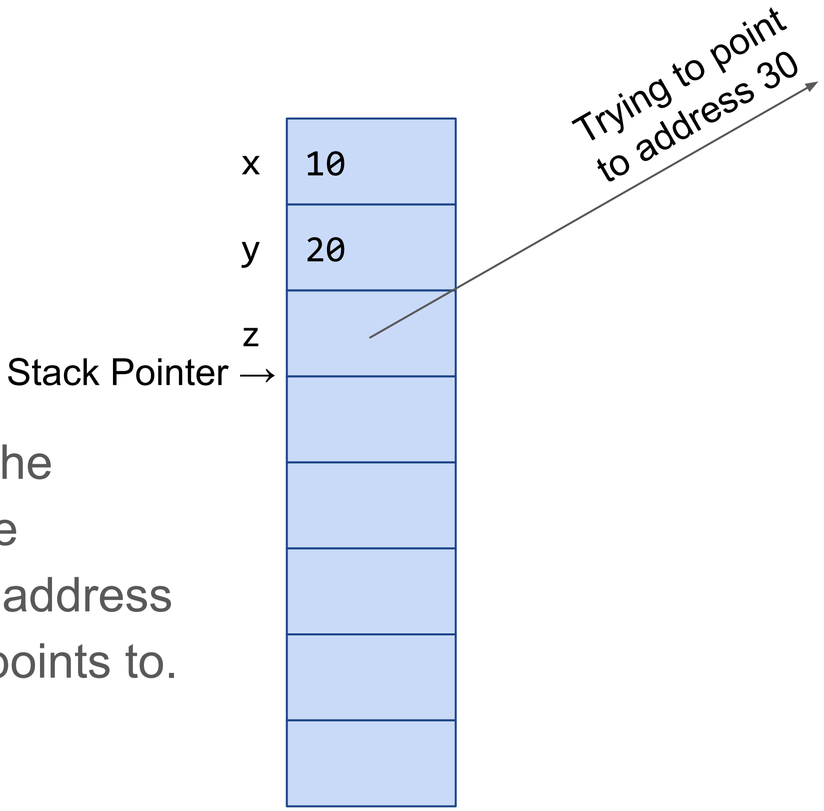
The star is used to **dereference** the pointer: interact with the value it is pointing to, not the address.



Another Option

```
int x = 10;
int y = 20;
int* z = new int;
z = 30;
```

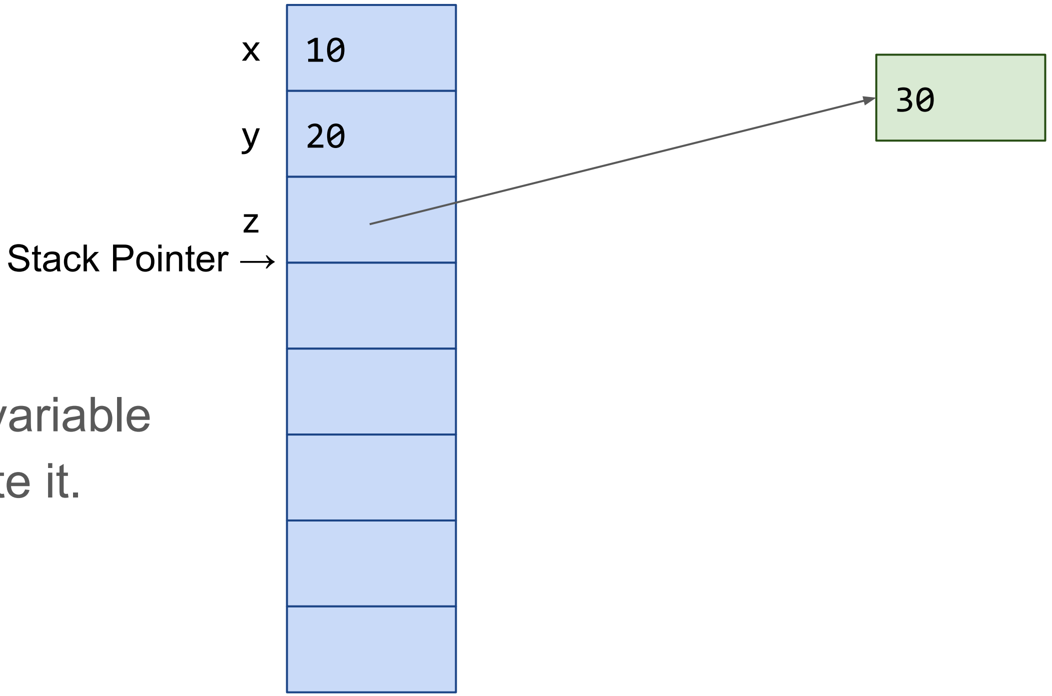
If we forget to dereference the pointer while using it, we are actually interacting with the address in memory that the pointer points to.



Another Option

```
int x = 10;
int y = 20;
int* z = new int;
*z = 30;
delete z;
```

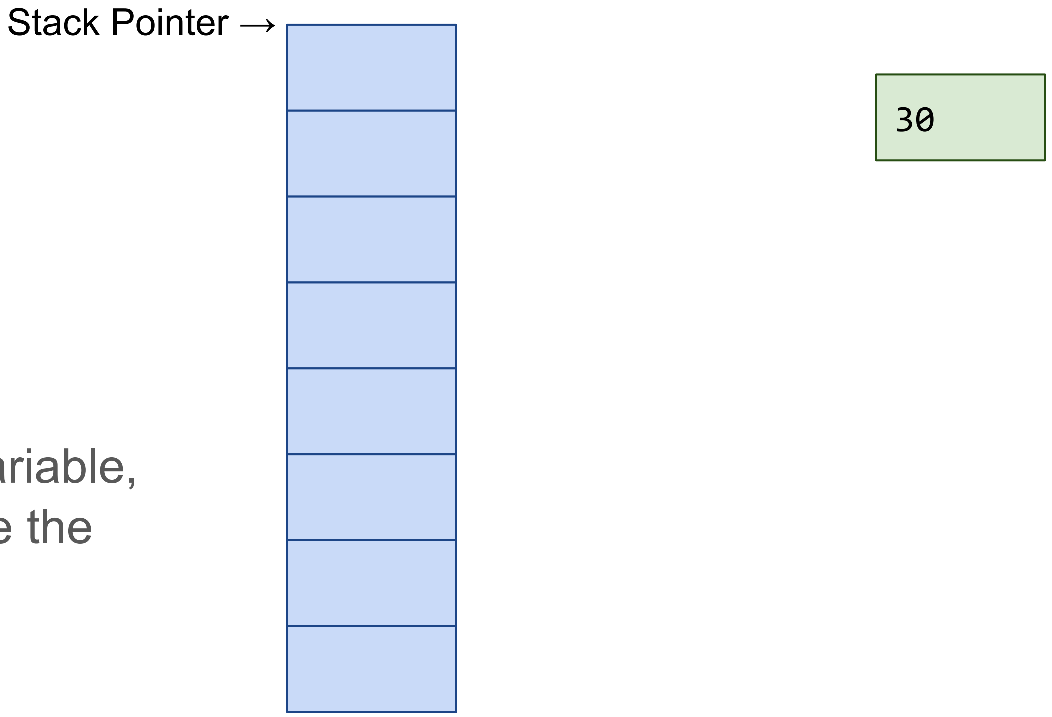
When you are done with a variable on the heap, you must delete it.



Another Option

```
{
    int x = 10;
    int y = 20;
    int* z = new int;
    *z = 30;
}
```

If you forget to delete the variable, and the block of code where the pointer was declared ends, you have a memory leak.



Daily “Quiz”

- Please open up your device, go to Gradescope, and we’ll spend the next couple of minutes doing the quiz
  - Feel free to discuss with your neighbors
  - Reminder: the quizzes are not graded on correctness, just completion
- Put your device down or close the lid when you are done
  - We’ll spend the last couple of minutes discussing the quiz as a class