Pointers

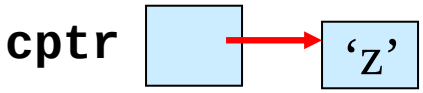
- A pointer variable stores the address of a memory location that stores the type to which it points ("a level of indirection")

```
int  *ptr;  // stores the address of an int,
            // ptr "points to" an int
char *cptr; // stores the address of a char,
            // cptr "points to" a char
```

- **`cptr`**'s type is a pointer to a char

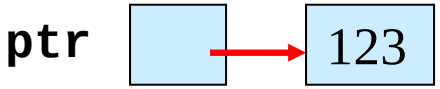
  it can point to a memory location that stores a char value

  through **`cptr`** we can indirectly access a char value



- **`ptr`**'s type is a pointer to an int

  it can point to a memory location that stores an int value

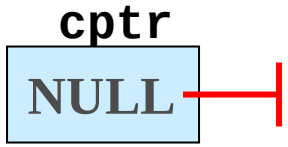


Initializing Pointer Variables

- Getting a pointer variable to "point to" a storage location
  (like any variable, must initialize a pointer before you can use it)
- Assign the pointer variable the value of a memory address that can store the type to which it points

1. **`NULL`** is a special init value for pointers, it's not a valid address

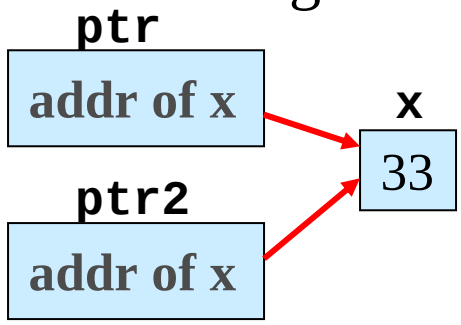
```
char *cptr = NULL;
```



2. Unary operator **&** evaluates to the address of its variable argument

```
int x = 33;
int *ptr = NULL, *ptr2 = NULL;
ptr = &x;   // ptr gets addr of x
            // ptr "points to" x
ptr2 = ptr; // ptr2 gets value of ptr
            // ptr and ptr2 point to the same location

char *cptr = &x; // ERROR! cptr can hold a char address only
```



Using Pointers

- Once a pointer is initialized to a point to a valid storage location, you can access the value to which it points using the * operator

  * : dereference a pointer variable

  (access the storage location to which it points)

```
 ptr = &x;  // ptr gets the address of x "ptr points to x"
*ptr = 10;  // store 10 in location that ptr points to
```


```
 cptr = NULL;
*cptr = 'b';  // CRASH!!! cptr doesn't point to a valid char
              // storage location, trying to dereference cptr
              // (a NULL pointer) will crash the program

if(cptr != NULL) {  // A better way is to test for NULL first.
       *cptr = 'b'; // Setting pointer to NULL, lets you test
}                   // for invalid addr. before dereference.
```

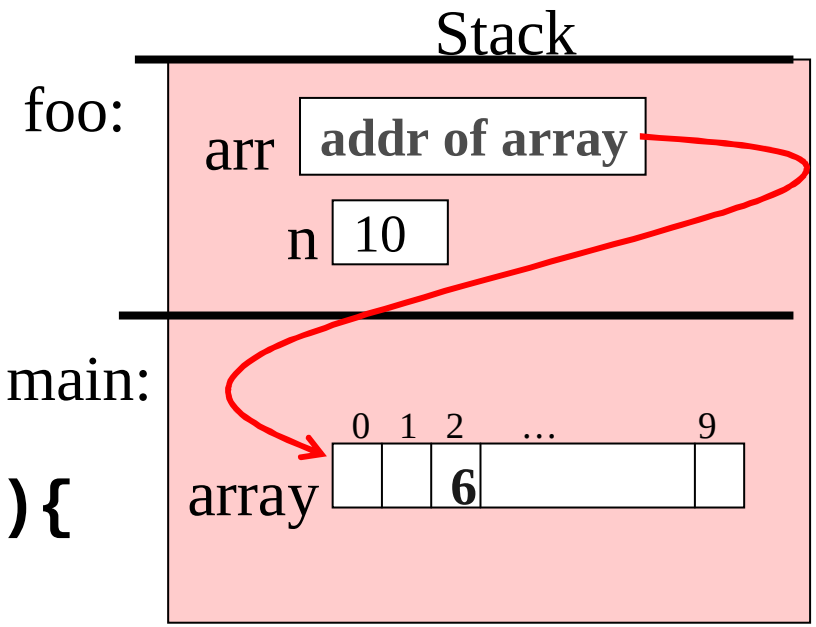
Passing Arrays

When passing an array to a function, its base address is passed (the function's parameter "points to" its array argument)

```
main(){
   int array[10];
   foo(array, 10);
}
```

pass base address of array

```
void foo(int arr[],int n){
   arr[2] = 6;
}
```



* Assigning a value to a bucket of **`arr`** in **`foo`**, modifies the corresponding bucket value of **`array`**

  `arr[2]` is **`arr+2`** is 2 **`int`** addresses beyond the the address of **`array`** (it is the address of the 2nd bucket of **`array`**)

Pass by Reference to Modify an Argument

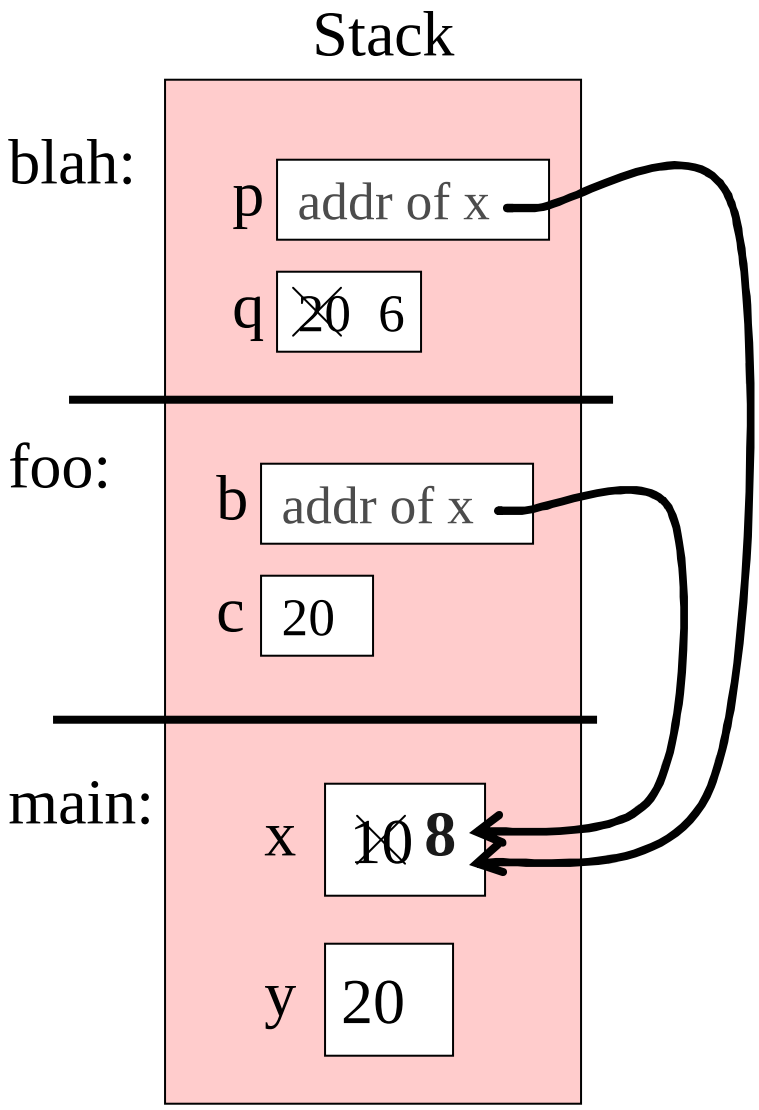
```
main(){
      int x, y;
      x = 10; y = 20;
      foo(&x, y);
}
```

pass the address of x
(x is passed by reference)

```
void foo(int *b, int c){
      *b = 8;
      blah( b , c);
}
```

the value of b (x's address)
(b is passed by value)

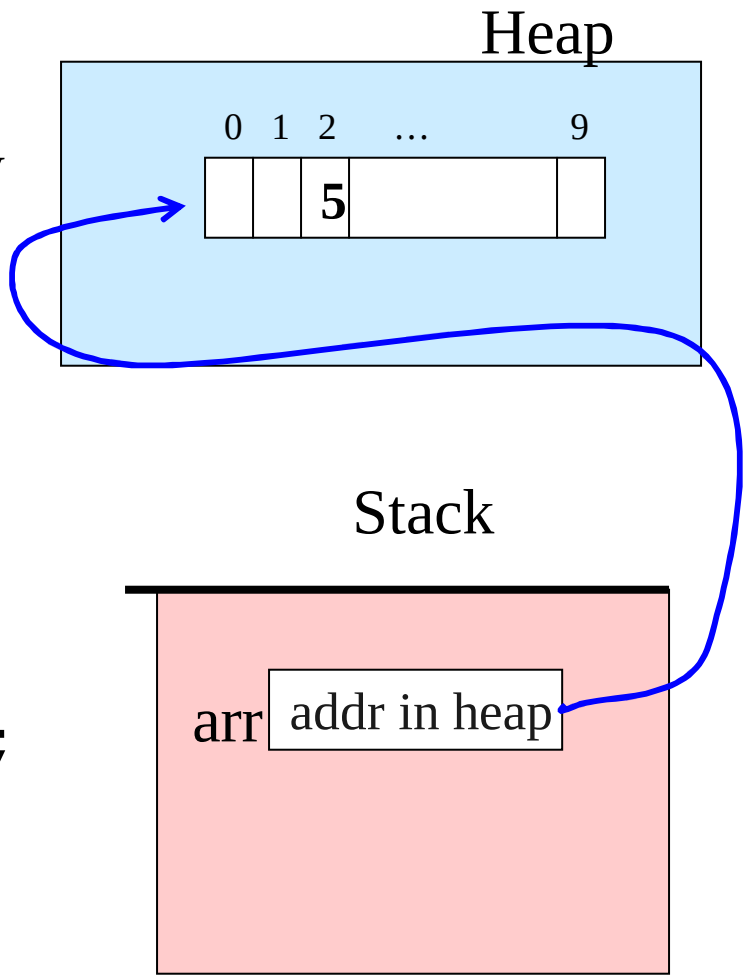
```
void blah(int *p, int q){
      q = 6;
}
```



foo and **blah** can modify the value stored in **x**

Dynamic Memory Allocation

- Can dynamically allocate memory space as your program needs it (**`malloc`**)
- Space is allocated in **Heap** memory
- Assign heap space address returned by **`malloc`** to a pointer variable
- Must free heap space when you are done using it (**`free`**)




```
main() {
  int *arr = NULL;
  // allocate heap space for
  // array of 10 ints:
  arr = malloc(sizeof(int)*10);
  if(arr != NULL) {
      arr[2]=5;
  }
  // free heap space when done
  free(arr);
```

```
main(){
  int *ar1, size=10;
  ar1 =  foo(size);

  if(ar1 != NULL) {
    ar1[1]=6;
  }
}

int *foo(int size){
  int *tmp;
  // allocate heap space:
  tmp=malloc(sizeof(int)*size);
  if(tmp != NULL) {
    tmp[2]=5;
  }
  // return malloc'ed heap address
  return tmp;
}
```

the value returned from foo is addr. of heap space foo malloc'ed