CMPSC 297 - Introduction to C Programming

PennState

Systems and Internet Infrastructure Security Laboratory

Week #4

Professor Patrick McDaniel

Memory allocation

- So far, we have seen two kinds of memory allocation:

```
int counter = 0; // a global variable


int foo(int a) {
  int x = a + 1;  // local variables
  return x;
}

int main() {
  int y = foo(10);  // local variable
  return 0;
}
```

`counter` is *statically* allocated

- allocated when program is loaded
- deallocated when program exits

`a,x,y` are *automatically* allocated

- allocated when function is called
- deallocated when function returns

We need more flexibility

- Sometimes we want to allocate memory that:
  - is too big to fit on the stack
  - its size is not known in advance to the caller (this is called *dynamic* memory)

```
int foo(int size) {
   char *bank_account = AllocateMemory(size);
   ...// do something with buffer

   return 0;
}
```

Memory allocation

- `malloc` allocates a block of memory of the given size and returns a pointer:

  `malloc(size in bytes)`

- Note: you should assume the memory initially contains garbage values
- Note: you'll typically use `sizeof` to calculate the size you need

```c
// allocate a 10-element float array
float *arr = (float *)malloc(10 * sizeof(float));

if (arr == NULL)
  return errcode;

arr[0] = 5.1;  // etc.
```

Memory deallocation

- Releases the memory pointed-to by the pointer:

```
free(pointer)
```

- after free( )ing a block of memory, that block of memory may be allocated again later in some future malloc( ) / calloc( )
- Note: it’s good form to set a pointer to NULL after freeing it

```
long *arr = (long *) calloc(10 * sizeof(long));
if (arr == NULL)
  return errcode;

// .. do something ..

free(arr);
arr = NULL;
```

Copying memory

- `memcpy` copies data from one memory region (pointer) to another
  - Copies from “source” to “destination” buffer

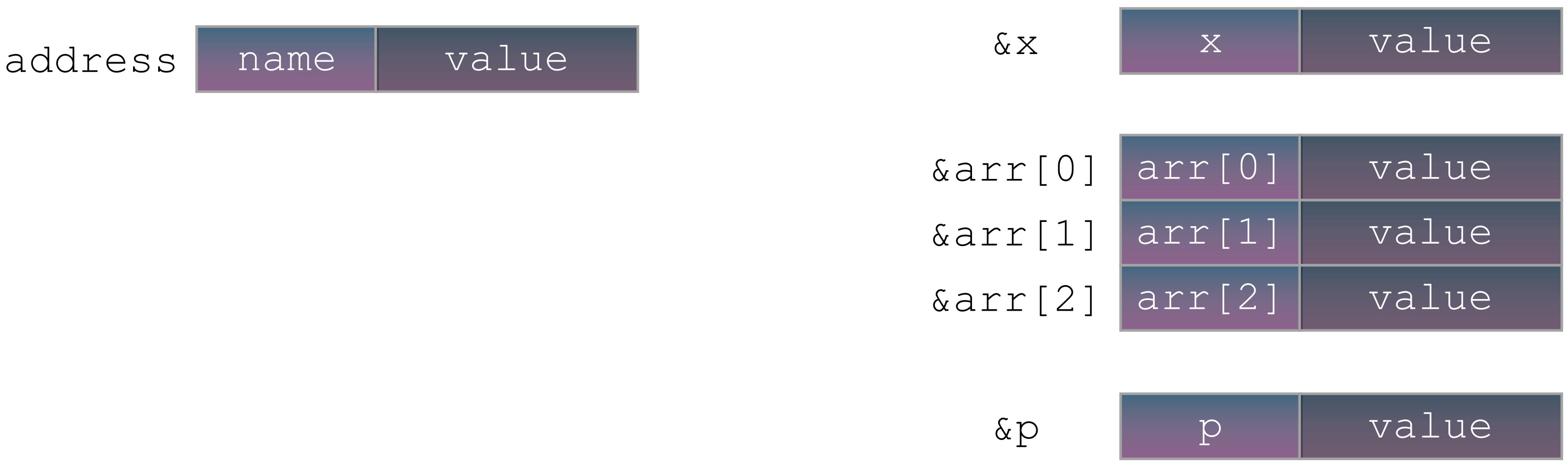
`memcpy(dest, src, n)`
is kinda like `dest = src`
for pointers

```
char buf1[4] = { 0, 1, 2, 3 };
char buf2[4] = { 0, 0, 0, 0 };

printf("Before\n");
for (int i = 0; i < 4; i++) {
  printf("buf1[i] = %d, buf2[i] = %d\n",
           (int)buf1[i], (int)buf2[i]);
}

// Copy the first 4 elements
memcpy(buf2, buf1, 4);

printf("\nAfter\n");
for (int i = 0; i < 4; i++) {
  printf("buf1[i] = %d, buf2[i] = %d\n",
           (int) buf1[i], (int)buf2[i]);
}
```

```
Output:

Before
buf1[i] = 0, buf2[i] = 0
buf1[i] = 1, buf2[i] = 0
buf1[i] = 2, buf2[i] = 0
buf1[i] = 3, buf2[i] = 0

After
buf1[i] = 0, buf2[i] = 0
buf1[i] = 1, buf2[i] = 1
buf1[i] = 2, buf2[i] = 2
buf1[i] = 3, buf2[i] = 3
```

Box and arrow diagrams

```
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

return 0;
}
```



address | name | value

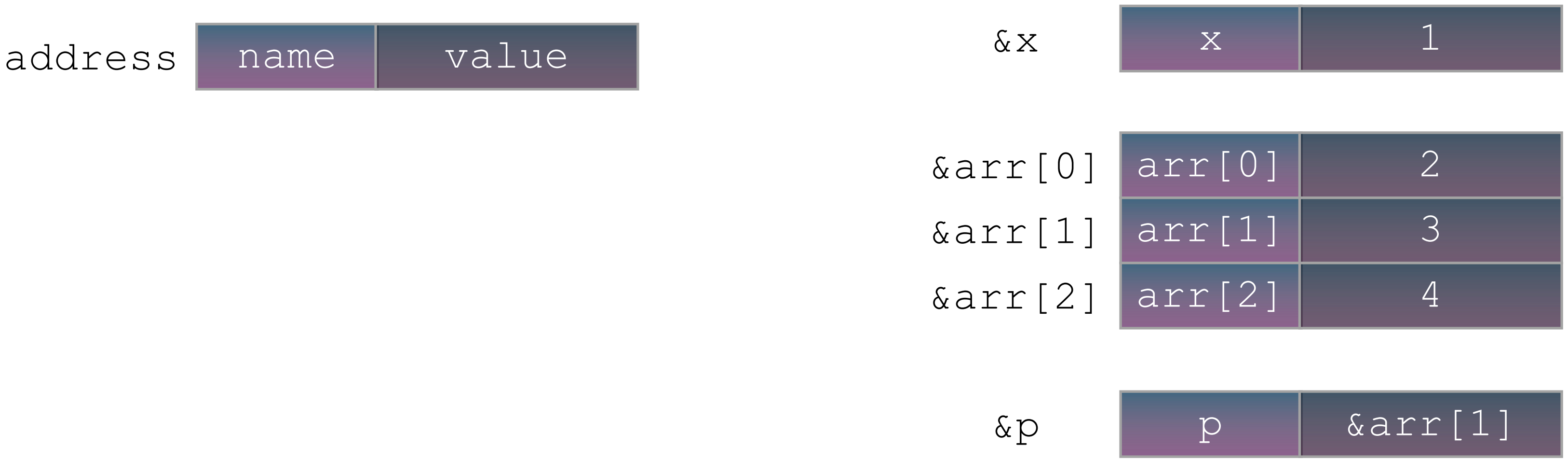
&x | x | value

&arr[0] | arr[0] | value
&arr[1] | arr[1] | value
&arr[2] | arr[2] | value

&p | p | value

Box and arrow diagrams

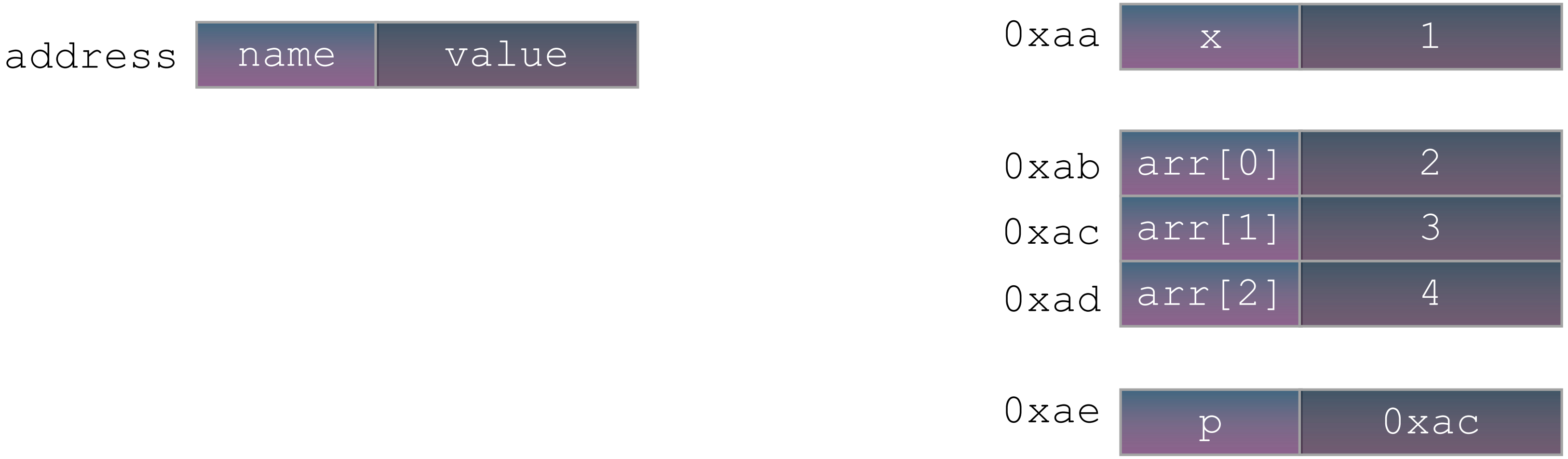
```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

return 0;
}
```



address	name	value
&x	x	1
&arr[0]	arr[0]	2
&arr[1]	arr[1]	3
&arr[2]	arr[2]	4
&p	p	&arr[1]

Box and arrow diagrams

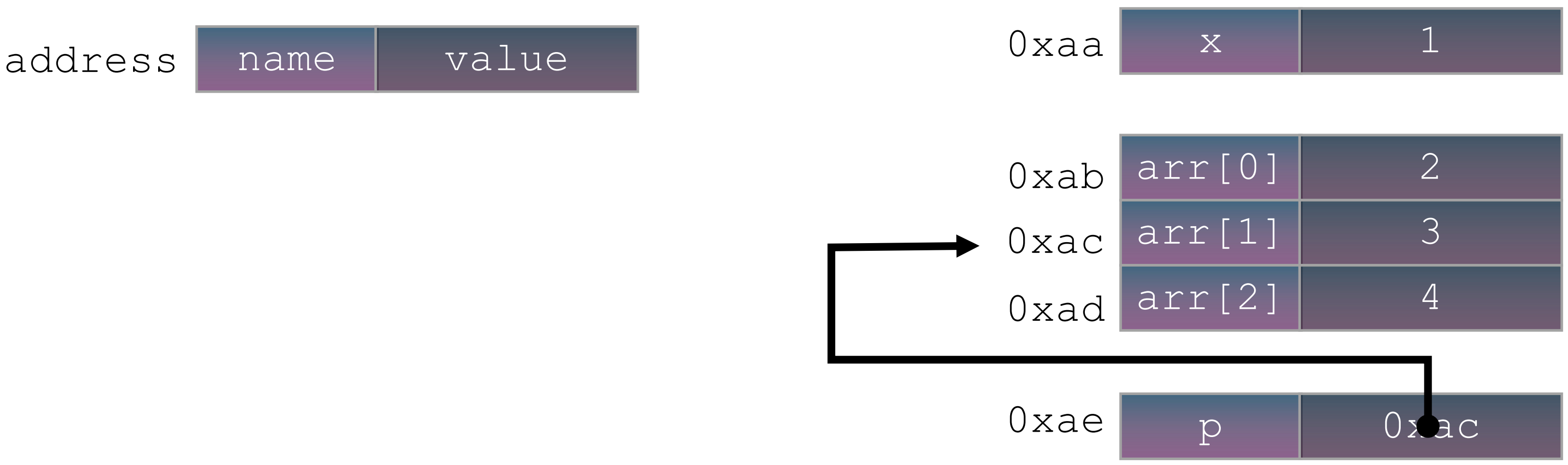
```
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

return 0;
}
```



address	name	value
0xaa	x	1
0xab	arr[0]	2
0xac	arr[1]	3
0xad	arr[2]	4
0xae	p	0xac

Box and arrow diagrams

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

return 0;
}
```



address	name	value
0xaa	x	1
0xab	arr[0]	2
0xac	arr[1]	3
0xad	arr[2]	4
0xae	p	0xac

Box and arrow diagrams

What if we added these lines? →

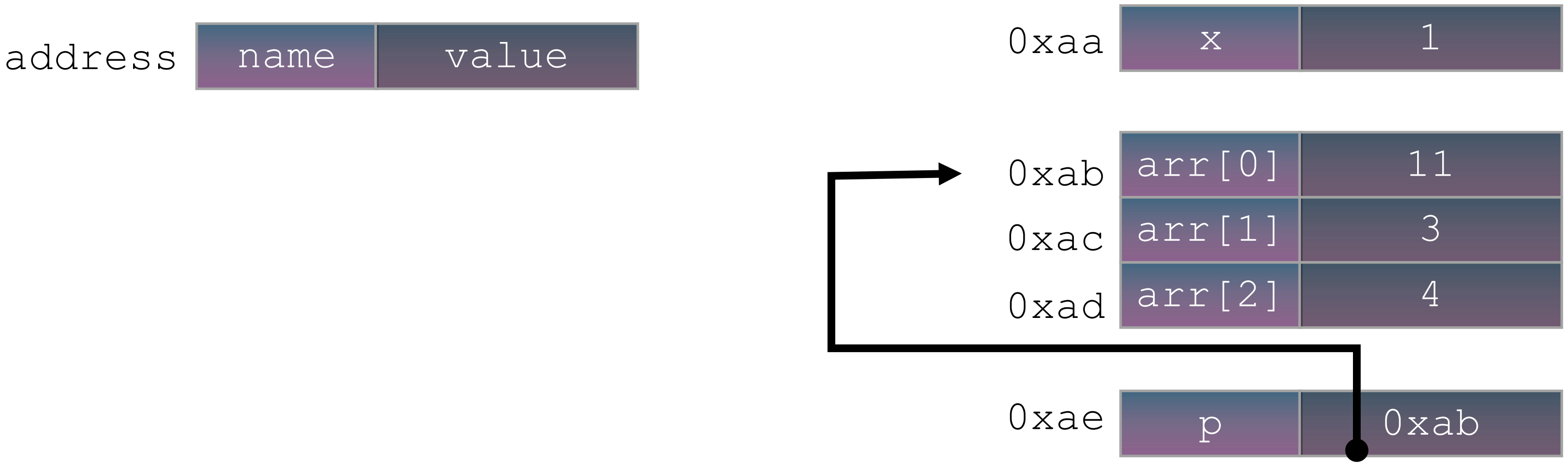
```
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  p  = &arr[0];
  *p = 11;
return 0;
}
```

address	name	value
0xaa	x	1
0xab	arr[0]	2
0xac	arr[1]	3
0xad	arr[2]	4
0xae	p	0xac

How will the diagram change?

Box and arrow diagrams

```c
int main(int argc, char **argv) {
  int x = 1;
  int arr[3] = {2, 3, 4};
  int *p = &arr[1];

  p  = &arr[0];
  *p = 11;
return 0;
}
```



address	name	value
0xaa	x	1
0xab	arr[0]	11
0xac	arr[1]	3
0xad	arr[2]	4
0xae	p	0xab

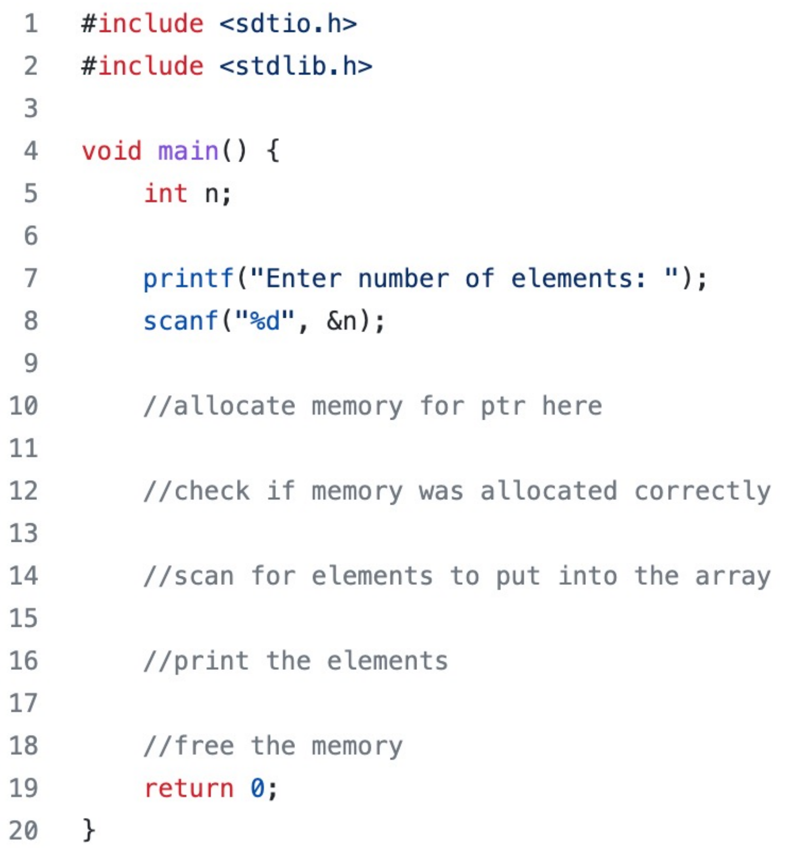
Practice Exercise #1

Draw 2 ***box and arrow diagrams*** for this code, one at each checkpoints.

```
int i;
int b[5] = {1, 3, 5, 7, 11};
int* a;

a = (int *) calloc(sizeof(b));

//// Checkpoint 1 ////

for (i = 0; i < sizeof(b); i++){
   a[i] =  b[i];
}

//// Checkpoint 2 ////
```

Practice Exercise #2

- Find the exercise on Canvas
- You will:
  - Dynamically allocate and free an array
  - Dynamically allocate a struct
  - Copy memory
  - Free memory



```
#include <sdtio.h>
#include <stdlib.h>

void main() {
    int n;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    //allocate memory for ptr here

    //check if memory was allocated correctly

    //scan for elements to put into the array

    //print the elements

    //free the memory
    return 0;
}
```

Syntax for Malloc/Calloc/Realloc:

```
type *var = (type *)malloc(<size>);
```