

CSc 352

C Programming
Arrays

Benjamin Dicken

Arrays

In C, an array can be thought of as a pointer to a chunk of memory. More specifically, a chunk of memory that is sequential, with a size that is $N * T$ where N is the number of slots in the array (length) and T is the size of the type of the array.

If **char** = 1 byte and **int** = 4 bytes, then:

```
// name is a ptr to 5 contiguous bytes
char name [5] = "zach";
// numbers is a ptr to 28 contiguous bytes
int numbers[7] = {1, 2, 3, 4, 5, 6, 7};
```

Arrays

// name is a "ptr" to 5 contiguous bytes

```
char name [5] = "zach";
```

// numbers is a "ptr" to 28 contiguous bytes

// if an int is 4 bytes

```
int numbers[7] = {1, 2, 3, 4, 5, 6, 7};
```

```
int * numbers_2 = numbers;
```

```
printf("%p %p %p\n", name, numbers, numbers_2);
```

```
printf("%ld %ld\n", sizeof(name), sizeof(numbers));
```

```
int numbers[7] = {50, 100, 150, 200, 250, 200, 250};  
// Code A  
for (int i = 0 ; i < 7; i++) {  
    printf("element %d is: %d\n", i, numbers[i]);  
}  
// Code B  
for (int i = 0 ; i < 7; i++) {  
    printf("element %d is: %d\n", i, *(numbers + i) );  
}  
// Code C  
for (int i = 0 ; i < 7; i++) {  
    printf("element %d is: %d\n", i, *numbers++ );  
}
```

What is the difference between these codes?

Arrays

- Can do math on pointers!
- Especially useful when dealing with arrays, iterations, offsets

```
char x[4] = {'z', 'r', 't', 'v'};
long y[4] = {100, 200, 300, 400};
char * x2 = x + 1;
long * y2 = y + 2;
printf("%c and %ld", *x2, *y2);
```

What will print?

```
char char_var = '\0';  
int int_var = 0;  
long long_var = 0;  
char * char_ptr = &char_var;  
int * int_ptr = &int_var;  
long * long_ptr = &long_var;
```

```
long one, two;
```

```
one = (long) char_ptr; two = (long) (char_ptr+1);  
printf("two minus one: %ld\n", two - one);
```

```
one = (long) int_ptr; two = (long) (int_ptr+1);  
printf("two minus one: %ld\n", two - one);
```

```
one = (long) long_ptr; two = (long) (long_ptr+1);  
printf("two minus one: %ld\n", two - one);
```

```
int values[10] = {0, 50, 100, 150, 200,
                 250, 300, 350, 400, 450};

int i = 0;
while (i < 5) {
    int a = *(values+i) + *(values+9-i);
    printf("%d ", a);
    i += 1;
}
printf("\n");
```

What will print?

```
void another (int array[]) {  
    long size = sizeof(array);  
    printf("size: %ld\n", size);  
}
```

```
int main() {  
    int values[10] = {0, 50, 100, 150, 200,  
                     250, 300, 350, 400, 450};  
    long size = sizeof(values);  
    printf("size: %ld\n", size);  
    another(values);  
    return 0;  
}
```

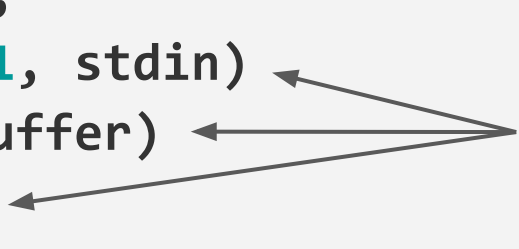
What will happen?
(without -Wall -Werror)

scanf and fgets with pointers

Remember lines such as these?

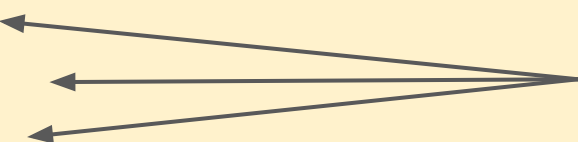
```
int x = 0;  
char buffer[32];  
fgets(buffer, 31, stdin)  
scanf("%31s", buffer)  
scanf("%d", &x)
```

Providing the fgets / scanf functions a *pointer* to the memory location to store the input value

The diagram consists of three arrows originating from a single point on the right side of the text block and pointing to the left. The top arrow points to the 'buffer' argument in the 'fgets' function call. The middle arrow points to the 'buffer' argument in the first 'scanf' function call. The bottom arrow points to the '&x' argument in the second 'scanf' function call.

What is wrong with this code?
What might happen?

```
char last[10];  
char middle[10];  
char first[10];  
printf("first name:\n");  
scanf("%s", first);  
printf("middle name:\n");  
scanf("%s", middle);  
printf("last name:\n");  
scanf("%s", last);  
printf("%s %s %s", first, middle, last);
```



Assume that these three arrays are allocated in sequence in space on the stack (10 bytes, then the next 10, then the next 10)

```
void main() {
    char last[10];
    char middle[10];
    char first[10];
    printf("first name:\n");
    scanf("%s", first);
    printf("middle name:\n");
    scanf("%s", middle);
    printf("last name:\n");
    scanf("%s", last);
    printf("%s %s %s", first, middle, last);
    return 0;
}
```

Memory Addresses

0x0...200 sp
// other vars

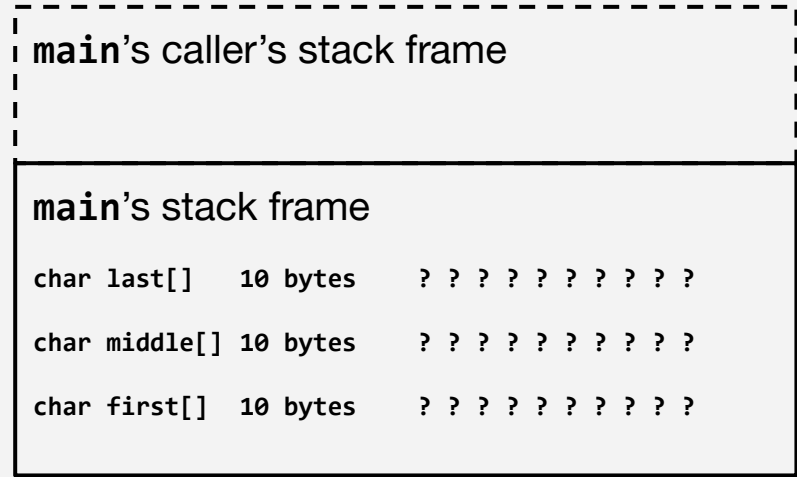
0x00...230 sp

0x00...238 last

0x00...244 middle

0x00...24e first

Stack Example



Stack growth direction



```

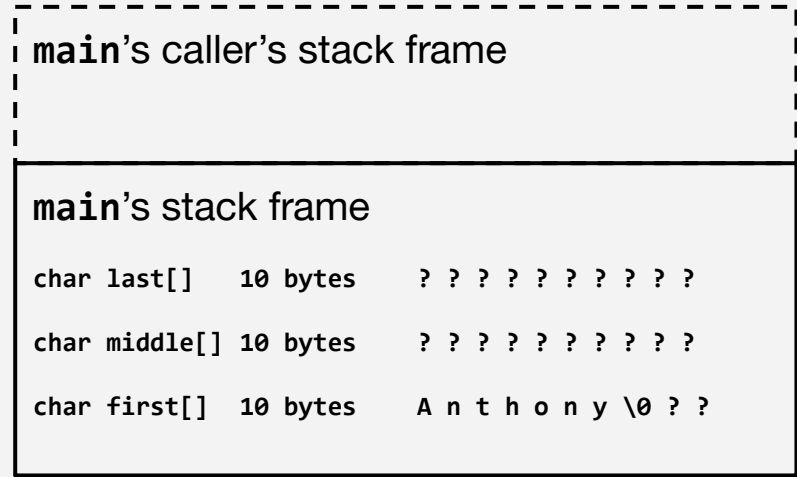
void main() {
    char last[10];
    char middle[10];
    char first[10];
    printf("first name:\n");
    scanf("%s", first);
    printf("middle name:\n");
    scanf("%s", middle);
    printf("last name:\n");
    scanf("%s", last);
    printf("%s %s %s", first, middle, last);
    return 0;
}

```

Memory Addresses

0x0...200 sp
 // other vars
 0x00...230 sp
 0x00...238 last
 0x00...244 middle
 0x00...24e first

Stack Example



Stack growth direction



Input provided: Anthony

```

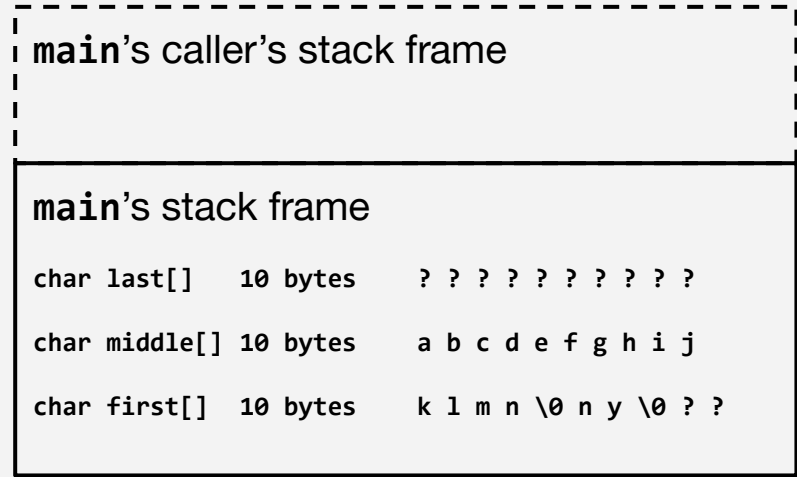
void main() {
    char last[10];
    char middle[10];
    char first[10];
    printf("first name:\n");
    scanf("%s", first);
    printf("middle name:\n");
    scanf("%s", middle);
    printf("last name:\n");
    scanf("%s", last);
    printf("%s %s %s", first, middle, last);
    return 0;
}

```

Memory Addresses

0x0...200 sp
 // other vars
 0x00...230 sp
 0x00...238 last
 0x00...244 middle
 0x00...24e first

Stack Example



Stack growth direction



Input provided: abcdefghijklmn

```

void main() {
    char last[10];
    char middle[10];
    char first[10];
    printf("first name:\n");
    scanf("%s", first);
    printf("middle name:\n");
    scanf("%s", middle);
    printf("last name:\n");
    scanf("%s", last);
    printf("%s %s %s", first, middle, last);
    return 0;
}

```

Memory Addresses

Stack Example

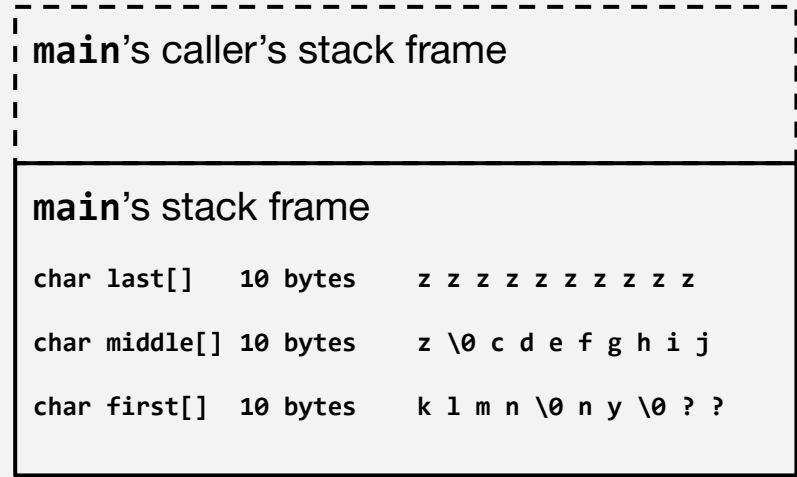
0x0...200 sp
 // other vars

 0x00...230 sp

 0x00...238 last

 0x00...244 middle

 0x00...24e first



Stack growth direction

Input provided: zzzzzzzzzzz

```

void main() {
    char last[10];
    char middle[10];
    char first[10];
    printf("first name:\n");
    scanf("%s", first);
    printf("middle name:\n");
    scanf("%s", middle);
    printf("last name:\n");
    scanf("%s", last);
    printf("%s %s %s", first, middle, last);
    return 0;
}

```

Memory Addresses

Stack Example

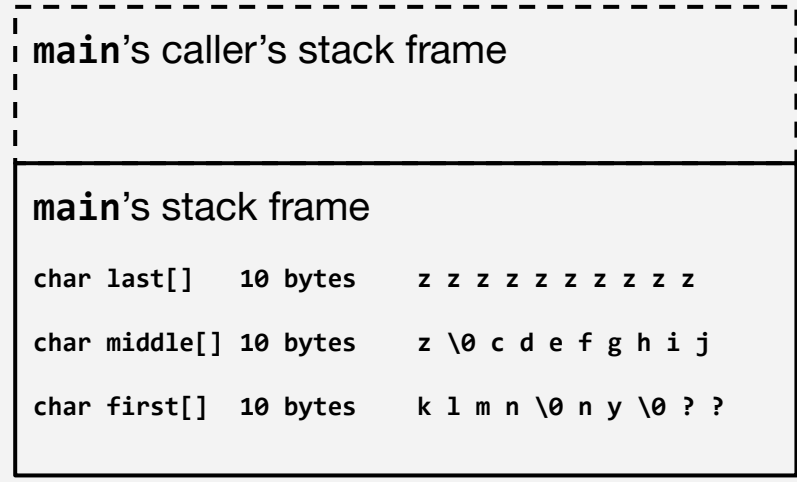
0x0...200 sp
 // other vars

0x00...230 sp

0x00...238 last

0x00...244 middle

0x00...24e first



Stack growth direction

Prints: klmn z zzzzzzzzzz

Uninitialized and Dangling Pointers

Uninitialized Pointer: A pointer that does not get assigned a value

- What happens when you look up a “random” address?

Dangling Pointer: Points to a location that is no longer valid

- Think: Points to a value that **was** on the stack but has been deallocated
- Think: Points to dynamically-allocated memory that has been freed

What do you think of this code?

```
char * get_name(char* prompt) {
    char buffer[32];
    printf("%s", prompt);
    scanf("%31s", buffer);
    return buffer;
}

int main() {
    char* name = get_name("Enter your name:\n");
    printf("Your name is %s\n", name);
    return 0;
}
```

What do you think of this code?

```
void get_name(char* prompt, char** name) {  
    char buffer[32];  
    printf("%s", prompt);  
    scanf("%31s", buffer);  
    *name = buffer;  
}
```

```
int main() {  
    char* name;  
    get_name("Enter your name:\n", &name);  
    printf("Your name is %s\n", name);  
    return 0;  
}
```

What do you think of this code?

```
void get_name(char* prompt, char* name) {  
    printf("%s", prompt);  
    scanf("%31s", name);  
}
```

```
int main() {  
    char name[32];  
    get_name("Enter your name:\n", name);  
    printf("Your name is %s\n", name);  
    return 0;  
}
```

The args array

```
int main(●) {  
    . . .  
}
```



```
int main(int argc, char * argv[]) {  
    . . .  
}
```



This can be thought of as:
“A pointer to an array of
pointers, each of which points
to a sequence of characters”

`$./a.out -v hi there`

