Pointer Arithmetic

2024 Winter APS 105: Computer Fundamentals
Jon Eyolfson

Lecture 14
1.0.0

Let's Move `sum` to a Function

```c
#include <stdio.h>

#define ARRAY_LENGTH(arr) (sizeof(arr) / sizeof((arr)[0]))

int sum(int array[]) {
    int arrayLength = ARRAY_LENGTH(array);
    int accumulator = 0;
    for (int i = 0; i < arrayLength; ++i) { accumulator += array[i]; }
    return accumulator;
}

int main(void) {
    int grades[] = {75, 83, 99, 64, 72};
    int gradesLength = ARRAY_LENGTH(grades);
    int average = sum(grades) / gradesLength;
    printf("Average: %d\n", average);
    return 0;
}
```

The Program Looks the Same, But the Output is Wrong

Previous, we saw: `Average: 78`
Now, `Average: 31`

What happened?

It Seems We Only Use the First Two Elements in `sum`

Let's just check what the size of the array is...

```
int sum(int array[]) {
    printf("sizeof(array): %d\n", (int) sizeof(array));
    int arrayLength = ARRAY_LENGTH(array);
    int accumulator = 0;
    for (int i = 0; i < arrayLength; ++i) {
        accumulator += array[i];
    }
    return accumulator;
}
```

If we run this we see: `sizeof(array): 8`

It Seems We Only Use the First Two Elements in `sum`

Let's just check what the size of the array is...

```
int sum(int array[]) {
    printf("sizeof(array): %d\n", (int) sizeof(array));
    int arrayLength = ARRAY_LENGTH(array);
    int accumulator = 0;
    for (int i = 0; i < arrayLength; ++i) {
        accumulator += array[i];
    }
    return accumulator;
}
```

If we run this we see: `sizeof(array): 8`

We need to see how C stores arrays in memory to explain this...

C Picks a Random Address, `a`, to Start the Array

The elements of the array are beside each other in memory
(the size of the element depends on the type, this slide assumes `int`)

	75	83	99	64	72
Index	0	1	2	3	4
Address	a+0x00	a+0x04	a+0x08	a+0x0c	a+0x10

To keep track of the array, C just uses a pointer to the start of the array!

Arrays Cannot Be Passed by Value

C will not create a copy of the entire array
Instead, C copies the address to the start of the array

A type like `int []` gets replaced by `int *`
You may find this referred to as array decay

Arrays Cannot Be Passed by Value

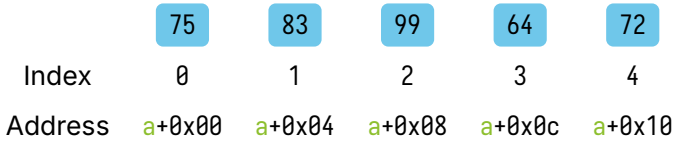
C will not create a copy of the entire array
  Instead, C copies the address to the start of the array

A type like `int []` gets replaced by `int *`
  You may find this referred to as array decay

So, `sizeof(array): 8` (in the `sum` function) because `array` is a pointer
  On 64-bit machines pointers, which are just addresses, are 8 bytes
    (any type that ends with a * is an address, and assume its 8 bytes)

If We Know `a`, We Can Compute the Address of Any Element

Again, assume we have our array of `int`:



Given `a`, the starting address of the array, and the index, `i`,
it seems like the address of an element should be: `a + i * sizeof(int)`

However, C automatically uses `sizeof` on the type pointed to for computing pointer arithmetic, so `a + i` in C results in the address above

We Can Only Perform Addition and Subtraction with Pointers

Assume we have:

```
int *a;
double *b;
char **c;
```

When C computes addresses:

`a + i` results in the address `a + i * 4`
`b + i` results in the address `b + i * 8`
`c + i` results in the address `b + i * 8`
`*c + i` results in the address `*c + i * 1`

C Guarantees the Order of Elements in Memory

	75	83	99	64	72
Index	0	1	2	3	4
Address	a+0x00	a+0x04	a+0x08	a+0x0c	a+0x10

We say array elements are contiguous (they are all together in memory)

There is also no wasted space, we use addresses [a+0x00, a+0x14)

Note: in a range [means inclusive and) means exclusive

In order words, addresses a + 0, a + 1, ..., a + 19 (20 bytes)

Using Pointer Arithmetic We Can Access Any Element

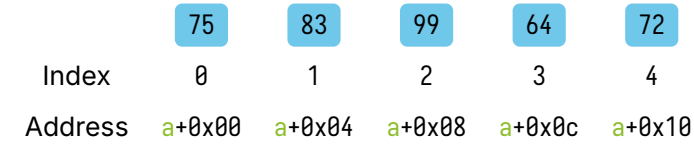
Assume we have our `int` array called `grades`:

	75	83	99	64	72
Index	0	1	2	3	4
Address	a+0x00	a+0x04	a+0x08	a+0x0c	a+0x10

`grades + 1` is a pointer to the second element (83)
We can dereference the pointer, `*(grades + 1)`, to access the value 83

Using Pointer Arithmetic We Can Access Any Element

Assume we have our `int` array called `grades`:



`grades + 1` is a pointer to the second element (`83`)
We can dereference the pointer, `*(grades + 1)`, to access the value `83`

The syntax to access an array element is just for your convenience
`grades[index]` is the same as `*(grades + index)`
You should always prefer the array syntax

`sizeof` an Array is Only Valid in Scope of Its Declaration

Otherwise, there's pointer decay, and it just becomes another pointer
Therefore, we need to tell the `sum` function the length of the array

```
int sum(int array[], int arrayLength) {
    int accumulator = 0;
    for (int i = 0; i < arrayLength; ++i) {
        accumulator += array[i];
    }
    return accumulator;
}
```

There are Rules Where Variables Are in Memory

Addresses for variables in functions (called local variables)
start at a randomly picked address, sp (short for stack pointer)

Variables will be put, in order, at lower addresses from the initial sp
Sometimes there will be space between variables, sometimes not

These details are not needed for this course,
and the rules may vary been CPUs and OSs

You Should Not Actually Do Pointer Arithmetic!

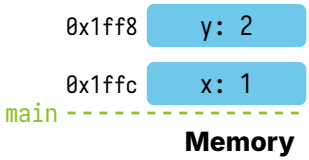
```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 1;
    int y = 2;
    int z = 3;
    int *p = &z;
    *(p + 2) = 4;
    printf("x: %d, y: %d, z: %d\n", x, y, z);
    return EXIT_SUCCESS;
}
```

You Should Not Actually Do Pointer Arithmetic!

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 1;
    int y = 2;
    int z = 3;
    int *p = &z;
    *(p + 2) = 4;
    printf("x: %d, y: %d, z: %d\n", x, y, z);
    return EXIT_SUCCESS;
}
```

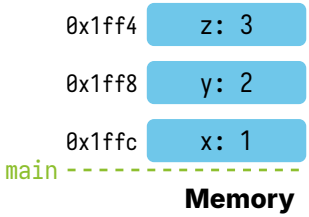
You Should Not Actually Do Pointer Arithmetic!

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 1;
    int y = 2;
    int z = 3;
    int *p = &z;
    *(p + 2) = 4;
    printf("x: %d, y: %d, z: %d\n", x, y, z);
    return EXIT_SUCCESS;
}
```



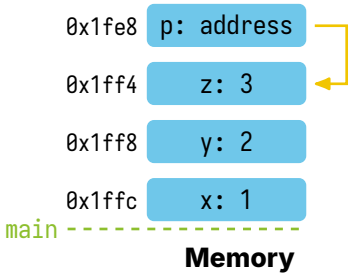
You Should Not Actually Do Pointer Arithmetic!

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 1;
    int y = 2;
    int z = 3;
    int *p = &z;
    *(p + 2) = 4;
    printf("x: %d, y: %d, z: %d\n", x, y, z);
    return EXIT_SUCCESS;
}
```



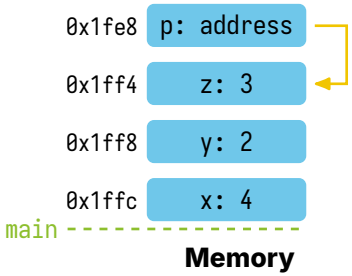
You Should Not Actually Do Pointer Arithmetic!

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 1;
    int y = 2;
    int z = 3;
    int *p = &z;
    *(p + 2) = 4;
    printf("x: %d, y: %d, z: %d\n", x, y, z);
    return EXIT_SUCCESS;
}
```



You Should Not Actually Do Pointer Arithmetic!

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 1;
    int y = 2;
    int z = 3;
    int *p = &z;
    *(p + 2) = 4;
    printf("x: %d, y: %d, z: %d\n", x, y, z);
    return EXIT_SUCCESS;
}
```



The Data Structure That Stores Values is Called a Stack

Notice in all our examples with memory in functions,
we add new variables to the top,
and we remove variables from the top

The real life analogy is a spring-loaded stack of plates
The last plate in, is the first plate out

The Data Structure That Stores Values is Called a Stack

Notice in all our examples with memory in functions,
we add new variables to the top,
and we remove variables from the top

The real life analogy is a spring-loaded stack of plates
The last plate in, is the first plate out

Again, you do not have to know these rules!
You may notice there's some unused addresses between `z` and `p`

Why Don’t Just Use Pointers Instead in Functions?

Instead of:

```
int sum(int array[], int arrayLength);
```

Why not:

```
int sum(int *array, int arrayLength);
```

Why Don't Just Use Pointers Instead in Functions?

Instead of:

```
int sum(int array[], int arrayLength);
```

Why not:

```
int sum(int *array, int arrayLength);
```

They do mean the same thing to the compiler, but as humans:

- `int array[]` is a pointer to multiple contiguous values
- `int *array` is a pointer to a single value

Beware: Variable Declarations are Strange with Pointers

Assume we make the following declaration:

```
int *x, y;
```

C does not carry any * across the commas in declarations, so:

The type of `x` is `int *`

The type of `y` is `int`

Beware: Variable Declarations are Strange with Pointers

Assume we make the following declaration:

```
int *x, y;
```

C does not carry any * across the commas in declarations, so:

The type of `x` is `int *`

The type of `y` is `int`

Instead of remembering this rule, always declare one variable at a time

If We Can't Set A Pointer Immediately, Initialize it to a "Safe" Address

If we don't know the value of an `int`, `x` then, we should declare it as `int x = 0;`

Otherwise, we may get a "random" value

If We Can't Set A Pointer Immediately, Initialize it to a "Safe" Address

If we don't know the value of an `int`, `x` then,
we should declare it as `int x = 0;`

Otherwise, we may get a "random" value

In this course, you should likely only set pointers to a value from &
If you don't know the value, you should declare it as `int *p = NULL;`

Dereferencing `NULL` Always Generates an Error

It's a special address (it's actually address `0`) that's always invalid

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p = NULL;
    *p = 1;
    return EXIT_SUCCESS;
}
```

When we run this we'll see: **segmentation fault** (with a bunch of numbers)
Our program immediately crashes and stops

Segmentation Faults are a Good Thing

They're easier to debug, because you know how far your program gets

It's much much **much** harder to debug if a random value changes

There's a tool called Valgrind that may help (you're not required to use it)
If you run your program on the command line,
you can write `valgrind` before your executable

Local Variables Only Exist While the Function Runs

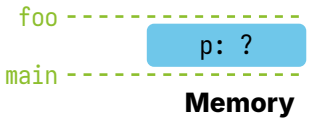
```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int x = 1;
    return &x;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```

main ----------

Memory

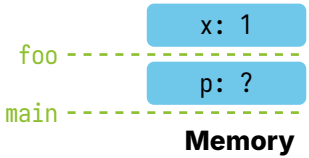
Local Variables Only Exist While the Function Runs

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int x = 1;
    return &x;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



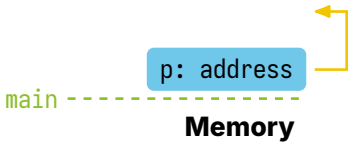
Local Variables Only Exist While the Function Runs

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int x = 1;
    return &x;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



Local Variables Only Exist While the Function Runs

```c
#include <stdio.h>
#include <stdlib.h>

int *foo(void) {
    int x = 1;
    return &x;
}

int main(void) {
    int *p = foo();
    printf("*p: %d\n", *p);
    return EXIT_SUCCESS;
}
```



We Can Return Pointers That are Still Valid

```c
#include <stdio.h>

int* maxPointer(int *a, int *b) {
    if (*a >= *b) {
        return a;
    }
    else {
        return b;
    }
}

int main(void) {
    int x = 1;
    int y = 2;
    int *p = maxPointer(&x, &y);
    printf("max: %d\n", *p);
    return 0;
}
```

main ----------

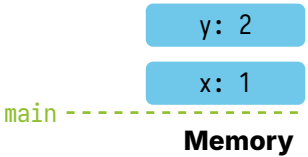
Memory

We Can Return Pointers That are Still Valid

```c
#include <stdio.h>

int* maxPointer(int *a, int *b) {
    if (*a >= *b) {
        return a;
    }
    else {
        return b;
    }
}

int main(void) {
    int x = 1;
    int y = 2;
    int *p = maxPointer(&x, &y);
    printf("max: %d\n", *p);
    return 0;
}
```

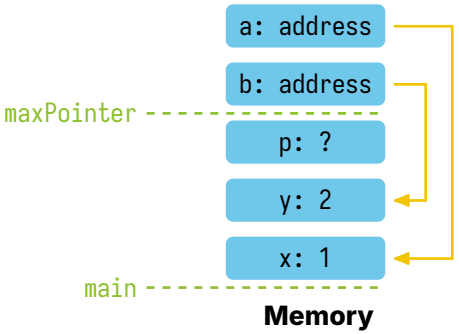
We Can Return Pointers That are Still Valid

```c
#include <stdio.h>

int* maxPointer(int *a, int *b) {
    if (*a >= *b) {
        return a;
    }
    else {
        return b;
    }
}

int main(void) {
    int x = 1;
    int y = 2;
    int *p = maxPointer(&x, &y);
    printf("max: %d\n", *p);
    return 0;
}
```



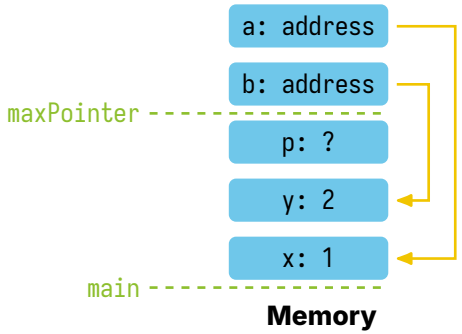
We Can Return Pointers That are Still Valid

```c
#include <stdio.h>

int* maxPointer(int *a, int *b) {
    if (*a >= *b) {
        return a;
    }
    else {
        return b;
    }
}

int main(void) {
    int x = 1;
    int y = 2;
    int *p = maxPointer(&x, &y);
    printf("max: %d\n", *p);
    return 0;
}
```



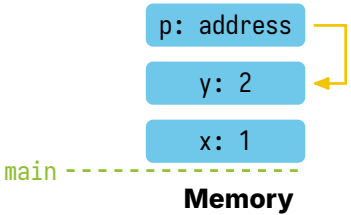
We Can Return Pointers That are Still Valid

```c
#include <stdio.h>

int* maxPointer(int *a, int *b) {
    if (*a >= *b) {
        return a;
    }
    else {
        return b;
    }
}

int main(void) {
    int x = 1;
    int y = 2;
    int *p = maxPointer(&x, &y);
    printf("max: %d\n", *p);
    return 0;
}
```



We Can Return Pointers That are Still Valid

```c
#include <stdio.h>

int* maxPointer(int *a, int *b) {
    if (*a >= *b) {
        return a;
    }
    else {
        return b;
    }
}

int main(void) {
    int x = 1;
    int y = 2;
    int *p = maxPointer(&x, &y);
    printf("max: %d\n", *p);
    return 0;
}
```



Be Very Careful with Pointers

Only use them when necessary
  Usually only when a function needs to “return” multiple values

You should try to write functions that only need to return one value

Variables only exist in memory at run-time while the function call is running