CS107 Spring 2019, Lecture 6
More Pointers and Arrays

Reading: K&R (5.2-5.5) or Essential C section 6
CS107 Topic 3: How can we effectively manage all types of memory in our programs?
Plan For Today

• Pointers and Parameters
• Arrays in Memory
• Arrays of Pointers
• **Announcements**
• Pointer Arithmetic
• Other topics: `const`, `struct` and ternary
Plan For Today

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- Announcements
- Pointer Arithmetic
- Other topics: `const`, `struct` and ternary
Pointers

• A pointer is a variable that stores a memory address.

• Because there is no pass-by-reference in C like in C++, pointers let us pass around the address of one instance of memory, instead of making many copies.

• One (8 byte) pointer can refer to any size memory location!

• Pointers are also essential for allocating memory on the heap, which we will cover later.

• Pointers also let us refer to memory generically, which we will cover later.
Memory

• Memory is a big array of bytes.
• Each byte has a unique numeric index that is commonly written in hexadecimal.
• A pointer stores one of these memory addresses.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x105</td>
<td>'\0'</td>
</tr>
<tr>
<td>0x104</td>
<td>'e'</td>
</tr>
<tr>
<td>0x103</td>
<td>'l'</td>
</tr>
<tr>
<td>0x102</td>
<td>'p'</td>
</tr>
<tr>
<td>0x101</td>
<td>'p'</td>
</tr>
<tr>
<td>0x100</td>
<td>'a'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Memory

• Memory is a big array of bytes.
• Each byte has a unique numeric index that is commonly written in hexadecimal.
• A pointer stores one of these memory addresses.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>262</td>
<td>\0</td>
</tr>
<tr>
<td>260</td>
<td>e</td>
</tr>
<tr>
<td>259</td>
<td>l</td>
</tr>
<tr>
<td>258</td>
<td>p</td>
</tr>
<tr>
<td>257</td>
<td>p</td>
</tr>
<tr>
<td>256</td>
<td>a</td>
</tr>
</tbody>
</table>

...
int x = 2;

// Make a pointer that stores the address of x.
// (& means "address of")
int *xPtr = &x;

// Dereference the pointer to get the data it points to.
// (* means "dereference")
printf("%d", *xPtr); // prints 2
A pointer is a variable that stores a memory address!

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```

![Stack diagram showing memory allocation and pointer usage](attachment://stack-diagram.png)
A pointer is a variable that stores a memory address!

```c
void myFunc(int *intPtr) {
   *intPtr = 3;
}

int main(int argc, char *argv[]) {
   int x = 2;
   myFunc(&x);
   printf("%d", x);   // 3!
   ...
}
```
A pointer is just a variable that stores a memory address!

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
A pointer is just a variable that stores a memory address!

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
A pointer is just a variable that stores a memory address!

void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
A pointer is just a variable that stores a memory address!

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);  // 3!
    ...
}
```
Pointers

A pointer is just a variable that stores a memory address!

```c
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x); // 3!
    ...
}
```

Address | Value
---|---
0x1f0 | 3

STACK

main()
When you pass a value as a parameter, C passes a copy of that value.

```c
void myFunction(int x) {
    ...
}

int main(int argc, char *argv[]) {
    int num = 4;
    myFunction(num); // passes copy of 4
}
```
C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```c
void myFunction(int *x) {
    ...
}

int main(int argc, char *argv[]) {
    int num = 4;
    myFunction(&num);     // passes copy of e.g. 0xffed63
}
```
C Parameters

When you pass a value as a parameter, C passes a copy of that value.

```c
void myFunction(char ch) {
    ...
}

int main(int argc, char *argv[]) {
    char myStr[] = "Hello!";
    myFunction(myStr[1]); // passes copy of 'e'
}
```
If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.
If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.

```c
void myFunction(char ch) {
    printf("%c", ch);
}

int main(int argc, char *argv[]) {
    char myStr[] = "Hello!";
    myFunction(myStr[1]); // prints 'e'
}
```
If you are performing an operation with some input and do not care about any changes to the input, pass the data type itself.

```c
int myFunction(int num1, int num2) {
    return x + y;
}

int main(int argc, char *argv[]) {
    int x = 5;
    int y = 6;
    int sum = myFunction(x, y);    // returns 11
}
```
C Parameters

If you are modifying a specific instance of some value, pass the location of what you would like to modify.

Do I care about modifying this instance of my data? If so, I need to pass where that instance lives as a parameter so it can be modified.
Pointers

If you are modifying a specific instance of some value, pass the *location* of what you would like to modify.

```c
void capitalize(char *ch) {
    // modifies what is at the address stored in ch
}

int main(int argc, char *argv[]) {
    char letter = 'h';
    /* We don't want to capitalize any instance of 'h'.
     * We want to capitalize *this* instance of 'h'!
     */
    capitalize(&letter);
    printf("%c", letter); // want to print 'H';
}
```
Pointers

If you are modifying a specific instance of some value, pass the location of what you would like to modify.

```c
void doubleNum(int *x) {
    // modifies what is at the address stored in x
}

int main(int argc, char *argv[]) {
    int num = 2;
    /* We don’t want to double any instance of 2.
     * We want to double *this* instance of 2! */
    doubleNum(&num);
    printf("%d", num); // want to print 4;
}
```
If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```c
void capitalize(char *ch) {
    // *ch gets the character stored at address ch.
    char newChar = toupper(*ch);

    // *ch = goes to address ch and puts newChar there.
    *ch = newChar;
}
```
Points

If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.

```c
void capitalize(char *ch) {
    /* go to address ch and put the capitalized version * of what is at address ch there. */
    *ch = toupper(*ch);
}
```
If a function takes an address (pointer) as a parameter, it can go to that address if it needs the actual value.

```c
void capitalize(char *ch) {
    // this capitalizes the address ch! 😞
    char newChar = toupper(ch);

    // this stores newChar in ch as an address! 😞
    ch = newChar;
}
```
Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```c
void printSquare(__?__) {
    int square = __?__ * __?__;
    printf("%d", square);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(__?__); // should print 9
}
```
Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```c
void printSquare(int x) {
    int square = x * x;
    printf("%d", square);
}

int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num);  // should print 9
}
```

We are performing a calculation with some input and do not care about any changes to the input, so we pass the data type itself.
Exercise 1

We want to write a function that prints out the square of a number. What should go in each of the blanks?

```c
void printSquare(int x) {
    x = x * x;
    printf("%d", x);
}
```

```c
int main(int argc, char *argv[]) {
    int num = 3;
    printSquare(num);  // should print 9
}
```
We want to write a function that flips the case of a letter. What should go in each of the blanks?

```c
void flipCase(__?__) {
    if (isupper(__?__)) {
        __?__ = __?__;
    } else if (islower(__?__)) {
        __?__ = __?__;
    }
}

int main(int argc, char *argv[]) {
    char ch = 'g';
    flipCase(__?__);
    printf("%c", ch);  // want this to print ‘G’
```
Exercise 2

We want to write a function that flips the case of a letter. What should go in each of the blanks?

```c
void flipCase(char *letter) {
    if (isupper(*letter)) {
        *letter = tolower(*letter);
    } else if (islower(*letter)) {
        *letter = toupper(*letter);
    }
}

int main(int argc, char *argv[]) {
    char ch = 'g';
    flipCase(&ch);
    printf("%c", ch);  // want this to print ‘G’
}```

We are modifying a specific instance of the letter, so we pass the location of the letter we would like to modify.
Pointers Summary

• If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**.

• If you are modifying a specific instance of some value, **pass the location** of what you would like to modify.

• If a function takes an address (pointer) as a parameter, it can **go to** that address if it needs the actual value.
Tip: setting a function parameter equal to a new value usually doesn’t do what you want. Remember that this is setting the function’s own copy of the parameter equal to some new value.

```c
void doubleNum(int x) {
    x = x * x;  // modifies doubleNum’s own copy!
}

void advanceStr(char *str) {
    str += 2;   // modifies advanceStr’s own copy!
}
```
Exercise 3

We want to write a function that advances a string pointer past any initial spaces. What should go in each of the blanks?

```c
void skipSpaces(__?__) {
    int numSpaces = strspn(__?__, " ");
    __?__ += numSpaces;
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skipSpaces(__?__);
    printf("%s", str); // should print "hello"
}
```
Exercise 3

We want to write a function that advances a string pointer past any initial spaces. What should go in each of the blanks?

```c
void skipSpaces(char **strPtr) {
    int numSpaces = strspn(*strPtr, " ");
    *strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *str = "    hello"
    skipSpaces(&str);
    printf("%s", str); // should print "hello"
}
```

We are modifying a specific instance of the string pointer, so we pass the location of the string pointer we would like to modify.
Exercise 3

We want to write a function that advances a string pointer past any initial spaces. What should go in each of the blanks?

```c
void skipSpaces(char *strPtr) {
    int numSpaces = strspn(strPtr, " ");
    strPtr += numSpaces;
}

int main(int argc, char *argv[]) {
    char *str = "   hello"
    skipSpaces(str);
    printf("%s", str);  // should print "hello"
}
```

This advances skipSpace's own copy of the string pointer, not the instance in main.
Demo: SkipSpaces
Plan For Today

• Pointers and Parameters
• Arrays in Memory
• Arrays of Pointers
• Announcements
• Pointer Arithmetic
• Other topics: const, struct and ternary
When you declare an array, contiguous memory is allocated on the stack to store the contents of the entire array.

```c
char str[] = "apple";
```

The array variable (e.g. `str`) is not a pointer; it refers to the entire array contents. In fact, `sizeof` returns the size of the entire array!

```c
int arrayBytes = sizeof(str); // 6
```
Arrays

An array variable refers to an entire block of memory. You cannot reassign an existing array to be equal to a new array.

```c
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2; // not allowed!
```

An array’s size cannot be changed once you create it; you must create another new array instead.
Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element*, and passes it (a pointer) to the function.

```c
void myFunc(char *myStr) {
  ...
}

int main(int argc, char *argv[]) {
  char str[] = "hi";
  myFunc(str);
  ...
}
```
Arrays as Parameters

When you pass an array as a parameter, C makes a copy of the address of the first array element, and passes it (a pointer) to the function.

```c
void myFunc(char *myStr) {
  ...
}

int main(int argc, char *argv[]) {
  char str[] = "hi";
  // equivalent
  char *arrPtr = str;
  myFunc(arrPtr);
  ...
}
```
Arrays as Parameters

This also means we can no longer get the full size of the array using `sizeof`, because now it is just a pointer.

```c
void myFunc(char *myStr) {
    int size = sizeof(myStr); // 8
}

int main(int argc, char *argv[]) {
    char str[] = "hi";
    int size = sizeof(str); // 3
    myFunc(str);
    ...
}
```
When you declare a char pointer equal to a string literal, the string literal is *not* stored on the stack. Instead, it’s stored in a special area of memory called the “Data segment”. You *cannot modify memory in this segment*.

```
char *str = "hi";
```

The pointer variable (e.g. `str`) refers to the *address of the first character of the string in the data segment*. Since this variable is just a pointer, `sizeof` returns 8, no matter the total size of the string!

```
int stringBytes = sizeof(str);  // 8
```
You can also make a pointer equal to an array; it will point to the first element in that array.

```c
int main(int argc, char *argv[]) {
  char str[] = "hi";
  char *ptr = str;
  ...
}
```
You can also make a pointer equal to an array; it will point to the first element in that array.

```c
int main(int argc, char *argv[]) {
    char str[] = "hi";
    char *ptr = str;

    // equivalent
    char *ptr = &str[0];

    // equivalent, but avoid
    char *ptr = &str;
    ...
}
```
Plan For Today

• Pointers and Parameters
• Arrays in Memory
• Arrays of Pointers
• Announcements
• Pointer Arithmetic
• Other topics: const, struct and ternary
You can make an array of pointers to e.g. group multiple strings together:

```c
char *stringArray[5];  // space to store 5 char *s
```

This stores 5 `char *s`, *not* all of the characters for 5 strings!

```c
char *str0 = stringArray[0];  // first char *
```
Arrays Of Pointers

```
./swapwords apple banana orange peach pear
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x128</td>
<td>0xf8a5 p e a r \0</td>
</tr>
<tr>
<td>0x120</td>
<td>0xf89f p e a c h \0</td>
</tr>
<tr>
<td>0x118</td>
<td>0xf898 o r a n g e \0</td>
</tr>
<tr>
<td>0x110</td>
<td>0xf887 b a n a n a \0</td>
</tr>
<tr>
<td>0x108</td>
<td>0xf881 a p p l e \0</td>
</tr>
<tr>
<td>0x100</td>
<td>0xf838 s w a p w o r d s \0</td>
</tr>
</tbody>
</table>

argc: 6
argv: 0x100
What is the value of `argv[2]` in this diagram?
• Pointers and Parameters
• Arrays in Memory
• Arrays of Pointers

• **Announcements**
• Pointer Arithmetic
• Other topics: `const`, `struct` and ternary
Announcements

• Lab 2
• GPS Rollover Event
GPS Rollover

• GPS is linked to the US Naval Observatory clock for timekeeping by tracking the number of weeks since the beginning of August 21, 1999

• The “week number” counter is 10 bits long

• On April 6, 2019, it overflowed to 0!

• Not the first time this has happened – it happens every 1,024 weeks

• Most newer GPS receivers are programmed to handle this overflow, but old ones were not. Also, other old un-updated systems such as cell networks, electrical utilities, etc. use GPS receivers for timing. Uh oh!

• Modernization plan: increase the week counter to 13 bits (157.5 year max)

• More info: https://arstechnica.com/information-technology/2019/04/gps-rollover-event-on-april-6-could-have-some-side-effects/
Plan For Today

- Pointers and Parameters
- Arrays in Memory
- Arrays of Pointers
- Announcements
- Pointer Arithmetic
- Other topics: `const`, `struct` and ternary
When you do pointer arithmetic, you are adjusting the pointer by a certain *number of places* (e.g. characters).

```c
char *str = "apple";  // e.g. 0xff0
char *str1 = str + 1;  // e.g. 0xff1
char *str3 = str + 3;  // e.g. 0xff3

printf("%s", str);    // apple
printf("%s", str1);   // pple
printf("%s", str3);   // le
```

DATA SEGMENT

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff0</td>
<td>'a'</td>
</tr>
<tr>
<td>0xff1</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff2</td>
<td>'p'</td>
</tr>
<tr>
<td>0xff3</td>
<td>'l'</td>
</tr>
<tr>
<td>0xff4</td>
<td>'e'</td>
</tr>
<tr>
<td>0xff5</td>
<td>'\0'</td>
</tr>
</tbody>
</table>
Pointer arithmetic does not work in bytes. Instead, it works in the size of the type it points to.

```c
// nums points to an int array
int *nums = ...  // e.g. 0xff0
int *nums1 = nums + 1;  // e.g. 0xff4
int *nums3 = nums + 3;  // e.g. 0xffc

printf("%d", *nums);  // 52
printf("%d", *nums1);  // 23
printf("%d", *nums3);  // 34
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1004</td>
<td>1</td>
</tr>
<tr>
<td>0x1000</td>
<td>16</td>
</tr>
<tr>
<td>0xffc</td>
<td>34</td>
</tr>
<tr>
<td>0xff8</td>
<td>12</td>
</tr>
<tr>
<td>0xff4</td>
<td>23</td>
</tr>
<tr>
<td>0xff0</td>
<td>52</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Pointer arithmetic does not work in bytes. Instead, it works in the size of the type it points to.

// nums points to an int array
int *nums = ... // e.g. 0xff0
int *nums3 = nums + 3; // e.g. 0xffc
int *nums2 = nums3 - 1; // e.g. 0xff8

printf("%d", *nums); // 52
printf("%d", *nums2); // 12
printf("%d", *nums3); // 34
When you use bracket notation with a pointer, you are actually performing pointer arithmetic and dereferencing:

```c
char *str = "apple";  // e.g. 0xff0

// both of these add two places to str,  
// and then dereference to get the char there.  
// E.g. get memory at 0xff2.
char thirdLetter = str[2];  // 'p'
char thirdLetter = *(str + 2);  // 'p'
```
Pointer arithmetic with two pointers does not give the byte difference. Instead, it gives the number of places they differ by.

```
// nums points to an int array
int *nums = ...  // e.g. 0xff0
int *nums3 = nums + 3;  // e.g. 0xffc
int diff = nums3 - nums;  // 3
```
How does the code know how many bytes it should look at once it visits an address?

```
int x = 2;
int *xPtr = &x;    // e.g. 0xff0

// How does it know to print out just the 4 bytes at xPtr?
printf("%d", *xPtr);    // 2
```
How does the code know how many bytes it should add when performing pointer arithmetic?

```c
int nums[] = {1, 2, 3};

// How does it know to add 4 bytes here?
int *intPtr = nums + 1;

char str[] = "CS107";

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```
• At compile time, C can figure out the sizes of different data types, and the sizes of what they point to.

• For this reason, when the program runs, it knows the correct number of bytes to address or add/subtract for each data type.
Plan For Today

• Pointers and Parameters
• Arrays in Memory
• Arrays of Pointers
• Announcements
• Pointer Arithmetic

• Other topics: const, struct and ternary
Use `const` to declare global constants in your program. This indicates the variable cannot change after being created.

```c
const double PI = 3.1415;
const int DAYS_IN_WEEK = 7;

int main(int argc, char *argv[]) {
  ...
  if (x == DAYS_IN_WEEK) {
    ...
  }
  ...
}
```
• Use `const` with pointers to indicate that the data that is pointed to cannot change.

```c
char str[] = "Hello";
const char *s = str;

// Cannot use s to change characters it points to
s[0] = 'h';
```
Sometimes we use `const` with pointer parameters to indicate that the function will not / should not change what it points to. The actual pointer can be changed, however.

```c
// This function promises to not change str’s characters
int countUppercase(const char *str) {
    int count = 0;
    for (int i = 0; i < strlen(str); i++) {
        if (isupper(str[i])) {
            count++;
        }
    }
    return count;
}
```
By definition, C gets upset when you set a non-const pointer equal to a const pointer. You need to be consistent with const to reflect what you cannot modify.

// This function promises to not change str’s characters
int countUppercase(const char *str) {
    // compiler warning and error
    char *strToModify = str;
    strToModify[0] = ...
By definition, C gets upset when you set a non-const pointer equal to a const pointer. You need to be consistent with const to reflect what you cannot modify. Think of const as part of the variable type.

```c
// This function promises to not change str’s characters
int countUppercase(const char *str) {
    const char *strToModify = str;
    // strToModify[0] = ... }```
**Const**

`const` can be confusing to interpret in some variable types.

// cannot modify this char
const char c = 'h';

// cannot modify chars pointed to by str
const char *str = ...;

// cannot modify chars pointed to by *strPtr
const char **strPtr = ...;
A *struct* is a way to define a new variable type that is a group of other variables.

```c
struct date { // declaring a struct type
    int month;
    int day;     // members of each date structure
};
```

```c
struct date today; // construct structure instances
today.month = 1;
today.day = 28;

struct date new_years_eve = {12, 31}; // shorter initializer syntax
```
Wrap the struct definition in a `typedef` to avoid having to include the word `struct` every time you make a new variable of that type.

```c
typedef struct date {
    int month;
    int day;
} date;
```

```c
date today;
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};
```
If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct.

```c
void advance_day(date d) {
    d.day++;
}

int main(int argc, char *argv[]) {
    date my_date = {1, 28};
    advance_day(my_date);
    printf("%d", my_date.day);  // 28
    return 0;
}
```
If you pass a struct as a parameter, like for other parameters, C passes a copy of the entire struct. **Use a pointer to modify a specific instance.**

```c
void advance_day(date *d) {
    (*d).day++;
}

int main(int argc, char *argv[]) {
    date my_date = {1, 28};
    advance_day(&my_date);
    printf("%d", my_date.day); // 29
    return 0;
}
```
The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```c
void advance_day(date *d) {
    d->day++;
}

int main(int argc, char *argv[]) {
    date my_date = {1, 28};
    advance_day(&my_date);
    printf("%d", my_date.day);  // 29
    return 0;
}
```
C allows you to return structs from functions as well. It returns whatever is contained within the struct.

date create_new_years_date() {
    date d = {1, 1};
    return d;     // or return (date){1, 1};
}

int main(int argc, char *argv[]) {
    date my_date = create_new_years_date();
    printf("%d", my_date.day);  // 1
    return 0;
}
sizeof gives you the entire size of a struct, which is the sum of the sizes of all its contents.

```c
typedef struct date {
    int month;
    int day;
} date;

int main(int argc, char *argv[]) {
    int size = sizeof(date);   // 8
    return 0;
}
```
arrays of structs

You can create arrays of structs just like any other variable type.

typedef struct my_struct {
    int x;
    char c;
} my_struct;

...

my_struct array_of_structs[5];
Arrays of Structs

To initialize an entry of the array, you must use this special syntax to confirm the type to C.

```c
typedef struct my_struct {
    int x;
    char c;
} my_struct;

my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```
Arrays of Structs

You can also set each field individually.

typedef struct my_struct {
    int x;
    char c;
} my_struct;

...

my_struct array_of_structs[5];
array_of_structs[0].x = 2;
array_of_structs[0].c = 'A';
The ternary operator is a shorthand for using if/else to evaluate to a value.

```
int x;
if (argc > 1) {
    x = 50;
} else {
    x = 0;
}

// equivalent to
int x = argc > 1 ? 50 : 0;
```
Recap

• Pointers and Parameters
• Arrays in Memory
• Arrays of Pointers
• **Announcements**
• Pointer Arithmetic
• Other topics: `const`, `struct` and ternary

**Next time:** dynamically allocated memory