# Data Structures in C

Prof. Georg Feil

# Pointers

Summer 2018

# Acknowledgement

- These lecture slides are partly based on slides by Professor Simon Hood
- Additional sources are cited separately

# Reading Assignment (required)

- C for Programmers (supplementary textbook)
  - Chapter 7, sections 7.1 to 7.9

# Pointers

- Pointers are probably the most difficult concept when learning C programming
- It will require a lot of practice to understand pointers well
- Java does not really have pointers, so for Java programmers this material will be new
  - However Java has references… any object variable is a reference
  - Also Java has null pointer exceptions!
    - Internally, a Java reference is like a C pointer

# Pointers are memory addresses

- The computer's memory (RAM) is used to store the values of variables (also code)
- Imagine your computer had only 1 million bytes of memory
  - Its memory locations would range from 0 to 999,999
- Now suppose you declare a variable

  ```
      int num = 0;
  ```

  and that variable gets stored in memory at location 5000
- The storage address, or simply address, of the variable num is 5000
  - A pointer with value 5000 will point to the variable num
  - The number actually occupies 4 bytes at 5000, 5001, 5002, 5003

# The address-of operator &

- The C address-of operator obtains the address of a variable (not its value)
  - The operator & (ampersand) goes to the left of the variable name

- If we apply the & operator to our num variable it would “return” 5000

  &num is 5000

The & operator creates a pointer

# Using the address-of operator &

- We've used the & operator with scanf

```
    scanf("%d", &grade);
```

- This passes a pointer to scanf
  - The pointer points to the variable named grade
- This allows the scanf function to access the value of the variable in memory and change it
  - The parameter is an output parameter, not an input parameter
  - We use the term "call-by-reference"

# Pointer variables

- C programmers often store pointers in variables
- This is how to store a pointer to num in a variable:

```
    int* ptr = &num;
```

- This stores the address of num in ptr
  - It does not store the value of num!
- We can say that ptr “points to” num
- The star after the data type means “pointer to int”
  - Most pointers have a data type
  - In this case we use int* because num is an int

# What are pointers good for?

- We’ve seen that pointers enable us to pass function parameters in both directions (input and output) … But what else are C pointers good for?
- Interact with hardware to write device drivers
- Control the speed of wheels on a Mars rover, or joints on a robot (memory-mapped devices)
- Efficiently work with data in memory to create advanced data structures (this course!)
- These things are impossible or difficult to do in Java

# Pointer variables

- We can use any of the following:

```
int *ptr;
int* ptr;
int * ptr;
```

- These are all the same to the compiler
  - C programmers tend to use `int *ptr`
  - C++ programmers might prefer `int* ptr`
- When declaring pointer variables avoid declaring more than one per line or it gets tricky!

# Pointer variables

- Suppose the address of num is 5000, and the value of num is 36
- The statement

```
    ptr = &num;
```

stores 5000 in ptr (imagine ptr was declared earlier)

- But wait… our int pointer ‘ptr’ is also a variable, so it has a memory location too
  - Suppose the location of ptr is 800
  - The value stored at memory location 800 will be 5000!

# Pointer variables

- This means that:
  - We have a variable called ptr at memory location 800 storing the value 5000, where 5000 is a memory address storing an int
  - And, we have a variable called num at memory location 5000 storing the value 36 where 36 is an actual int value

- If you follow that, everything else will be easy ☺
  - If not, read these slides again, read the supplementary textbook, and practice, practice, practice!

# Exercise 1

- Create a C program that declares
  - An int variable ‘num’ with value 36
  - A pointer ‘ptr’ that points to the variable ‘num’
- Now print the values of ‘num’ and ‘ptr’
  - First use %d to print the pointer (you’ll get a warning)
  - Then use %p for the pointer (you’ll still get a warning)
- Add 1 to the int and print it again
- Add 1 to the pointer and print it again

# Reference types and & (address of)

- When reading value types we use & in scanf, for example

```
double value;
scanf("%lf", &value);
```

- With reference types or pointers we usually don’t use &

```
double* ptr = &value;
scanf("%lf", ptr);
char str[80];
scanf("%s", str);
```

- ‘ptr’ and ‘str’ are already references/pointers
  - Don’t need &

# Void pointers and casting pointers

- In C a void pointer is a pointer without a data type
  - It can point to any type of data
- You can create a void pointer variable like this

```
void* pv;
```

- You can assign any pointer to a void pointer variable, but to turn it back into a "real" pointer that has a data type requires a cast (more about this later…)
  - You can also cast any pointer to a void pointer if you like
- To eliminate warnings when printing pointers in Exercise 1, cast them to void like this: `(void*)ptr`

# Using (Dereferencing) Pointers

# Dereferencing Pointers

- Using a pointer to access or change the value it points to is called dereferencing
- Put the * operator before a pointer variable to deference or “follow” the pointer
  - The * operator is used to dereference as well as declare pointers… this can be a bit confusing
- Example: Access the value pointed to by ‘ptr’

```
int num2 = *ptr;   // Assume ptr is int*
```

- Example: Change (assign) the value pointer to by ‘ptr’

```
*ptr = 7;
```

- Note that you can’t dereference void pointers

- When * appears to the right of a data type, it's a pointer type
- When * appears to the left of a pointer variable, it dereferences the pointer

# Exercise 2

Continuing with Exercise 1,

- Try to print the value of ‘num’ using ‘ptr’
- Now try to add 1 to ‘num’ using ‘ptr’
  - Print num again to prove it worked
  - Hint: Don’t use ++ for now or you might have trouble

# Dereferencing pointers and ++, --

- How can we ++ or -- values pointed to by pointers?

```
ptr++
```

Adds to the address (changes the pointer). This is called pointer arithmetic, not what we want here.

```
*ptr++
```

Gets the value pointed to by ptr, but then adds to the address (changes the pointer). Also not what we want.

```
(*ptr)++
```

Gets the value pointed to by ptr, then adds 1 to it. This is what we want! So parentheses are needed for ++ and -- to work on values.
Try it in your program from Exercise 2!

# Derefencing and &

- If ptr points to num then `*ptr` (dereferencing ptr) is the same as num
- Since & turns something into a pointer and * turns a pointer into a value, they can be considered opposites
- Therefore `*(&num)` is the same as num
- … Try this in your Exercise 2 program and see!

# Relationship between C arrays and pointers

- Arrays and pointers are almost exactly the same thing!
  - You can think of an array as a pointer to the first element of its data
  - Also, square brackets are just a special way to dereference a pointer!
- That means you can assign an array to a pointer, and use the pointer to access or change the array

```
int arr[20];     // Declare an array
int *ptr;        // Declare a pointer
ptr = arr;      // ptr will point to start of arr
*ptr = 5;        // Sets the 1st element of arr, arr[0]
printf("%d",*ptr);  // Prints 1st element of arr
printf("%d",*arr);  // Also prints 1st element of arr
ptr[0] = 5;      // Also sets the first element of arr
ptr[4] = 77;     // Sets the fifth element of arr
```

# Relationship between C arrays and pointers

- Are there any differences between arrays and pointers in C?
- Yes… you can't do this

```
int arr[20];
arr = ptr;
```

- Arrays always point to the same address in memory, but pointers can be changed to point to different addresses
- Also it doesn't make much sense to do &arr, but advanced programs sometimes do &ptr
  - This is a pointer to a pointer!
- A string is an array of characters, so char* means string!

# Pointers Summary

- Pointers are memory addresses
- The * operator is used to declare pointers, and to dereference them
- Pointers have a data type, except void pointers (void*)
- The & (address of) operator is used to get a pointer to a variable
- Pointers and arrays are *almost* the same thing
- When de-referencing, `*ptr` is equal to `*(&num)` is equal to ptr[0] is equal to `num` (if ptr points to num)

# Exercise 3

- Write a short C program that declares and initializes (to any value you like) a double, an int, and a string.
- Print the value stored in each variable
- Print the address of each variable using (remember to use %p for pointers/addresses)
- Modify your program by rearranging the variable declarations and/or changing the length of the string. Does this change the results you got previously?

# Exercise 4

- What is the output of the following code segment? Study the code and write your answer on paper before you try running it.

```
int count = 10;
int* temp;
int sum = 0;
temp = &count;
*temp = 20;
temp = &sum;
*temp = count;
printf("count = %d, *temp = %d, sum = %d\n",
       count, *temp, sum);
```