CMPUT201W20B2 Week 4

Abram Hindle

March 10, 2020

Contents

1 Week4

1.1 Copyright Statement

1.1.1 License

1.1.2 Hazel Code is licensed under AGPL3.0+

1.2 Init ORG-MODE

```
;; I need this for org-mode to work well

(require 'ob-sh)
;(require 'ob-shell)
(org-babel-do-load-languages 'org-babel-load-languages '((sh . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((C . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((python . t)))
(setq org-src-fontify-natively t)
```

1.2.1 Org export

```
(org-html-export-to-html)
(org-latex-export-to-pdf)
(org-ascii-export-to-ascii)
```

1.3 Org Template

Copy and paste this to demo C

```
#include <stdio.h>

int main(int argc, char**argv) {
    return 0;
}
```

1.4 Remember how to compile?

gcc -std=c99 -Wall -pedantic -Werror -o programname programname.c

1.5 Functions

Functions replicate functions in mathematics. They allocate space on the stack and have local variables.

Very similar to python functions

Define a function:

$\text{return}_{\text{type}}$ functionName(ArgType1 arg1, ArgType2 arg2, ArgType3 arg3) { ... }

Call a function:

functionName(arg1, arg2, arg3);

$\text{return}_{\text{type}}$ returnValue = functionName(arg1, arg2, arg3) ;

IN C89 all variable declarations are at the top of the function.

1.5.1 $\mathbf{return_{types}}$

- void – nothing
- int
- char
- float
- double
- ...
- pointer (array or string)

1.5.2 Example

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

void example() {
    printf("I have been made an example of\n");
    // return; // void return
}

int main() {
    example();
    return 0;
}

I have been made an example of
```

1.5.3 Pass by Value

The value of parameters are COPIED into registers and sometimes the stack. Thus the original variables that the parameters come from are safe.

Except pointers are not safe because given a pointer the called function can manipulate the data the pointer points to, but they cannot modify the original pointer.

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

int example(int x) {
    x++;
    printf("example x:\t%p\n", (void*)&x);
    return x;
}

int main() {
    int x = 10;
    printf("main x :\t%p\n", (void*)&x);
    printf("x: %d\n", x);
    int rx = example(x);
    printf("x: %d\n", x);
    printf("returned x vs x: %d vs %d\n", rx, x);
}
```

```
main x : 0x7ffe19cd7700
x: 10
example x: 0x7ffe19cd76ec
x: 10
returned x vs x: 11 vs 10
```

1.5.4 Arrays again

- void initArray(int cols, int values[cols]) {
- void initArray(int cols, int values[]) {

You can specify array sizes in C99 but the size has to come earlier

- void init2D(int rows, int cols, int values[rows][cols]) {
- void init2D(int rows, int cols, int values[][cols]){
- void init3D(int planes, int rows, int cols, int values[planes][rows][cols]) {
- void init3D(int planes, int rows, int cols, int values[][rows][cols]) {

1.5.5 Don't trust sizeof inside of functions!

sizeof is only trustable if you declared the variable in your scope

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

void init2D(int rows, int cols, int values[][cols]) {
    int i = 0;
    printf("init2D: sizeof(values)=%lu\n", sizeof(values));
    printf("init2D: sizeof(values[0])=%lu\n", sizeof(values[0]));

    for (int row = 0; row < rows; row++) {
        for (int col = 0; col < cols; col++) {
            values[row][col] = i++;
        }
    }
}
void example() {
    unsigned int n = 1 + rand() % 10;
    unsigned int m = 1 + rand() % 10;
    printf("%d X %d was chosen!\n", m, n);
    int values[m][n]; // SO the compiler can't predict this allocation ahead of time
    printf("sizeof(values) = %ld\n", sizeof(values));
    printf("sizeof(&values) = %ld\n", sizeof(&values));
    printf("sizeof(values[0]) = %ld\n", sizeof(values[0]));
    init2D(  m, n, values );
}
int main() {
    srand(time(NULL)); //initialze based on the clock
    example();
    example();
    example();
}
```

```
10 X 7 was chosen!
sizeof(values) = 280
sizeof(&values) = 8
sizeof(values[0]) = 28
init2D: sizeof(values)=8
```

```
init2D: sizeof(values[0])=28
5 X 8 was chosen!
sizeof(values) = 160
sizeof(&values) = 8
sizeof(values[0]) = 32
init2D: sizeof(values)=8
init2D: sizeof(values[0])=32
5 X 2 was chosen!
sizeof(values) = 40
sizeof(&values) = 8
sizeof(values[0]) = 8
init2D: sizeof(values)=8
init2D: sizeof(values[0])=8
```

1.5.6 Returns

Don't return arrays in general.

To return a value and exit the function immediately run:
return expr

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

int squareInt(int x) {
    return x*x;
}

float squareFloat(float x) {
    return x*x;
}
int intDiv(int x, int y) {
    return x/y;
}
float floatDiv(float x, float y) {
    return x/y;
}
char returnChar( int i ) {
    return i;
}
```

```
int main() {
    printf("squareInt\t %d\n", squareInt(25));
    printf("squareInt\t %d\n", squareInt(1.47));
    printf("squareFloat\t %f\n", squareFloat(1.47));
    printf("squareFloat\t %f\n", squareFloat(25));
    printf("intDiv\t %d\n", intDiv(64,31));
    printf("intDiv\t %d\n", intDiv(64.2,31));
    printf("floatDiv\t %f\n", floatDiv(64,31));
    printf("floatDiv\t %f\n", floatDiv(64.2,31));
    printf("returnChar\t %hhu\n", returnChar( 578 ) );
    printf("returnChar\t %hhu\n", returnChar( 'a' ) );
    printf("returnChar\t %hhu\n", returnChar( 66.1 ) );
    printf("returnChar\t %c\n", returnChar( 578 ) );
    printf("returnChar\t %c\n", returnChar( 'a' ) );
    printf("returnChar\t %c\n", returnChar( 66.1 ) );

}
```

```
squareInt  625
squareInt  1
squareFloat  2.160900
squareFloat  625.000000
intDiv  2
intDiv  2
floatDiv  2.064516
floatDiv  2.070968
returnChar  66
returnChar  97
returnChar  66
returnChar  B
returnChar  a
returnChar  B
```

1.5.7 Recursion

1. Recursion
    (a) Recursion
        i. Recursion

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

```
#include <stdlib.h>

int divisibleBy(int x, int y);

int main() {
    printf("%d\n",divisibleBy(33,32));
}

int divisibleBy(int x, int y) {
    printf("%d %d\n", x,y);
    if (x == 0) { return 0; }
    if (y <= 0) { return 0; }
    if (x % y == 0) { return y; }
    return divisibleBy(x, y - 1);
}
```

```
33 32
33 31
33 30
33 29
33 28
33 27
33 26
33 25
33 24
33 23
33 22
33 21
33 20
33 19
33 18
33 17
33 16
33 15
33 14
33 13
33 12
33 11
11
```

1.5.8 Prototypes

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

/* this is a prototype
   it predeclares that a function with this
   name will be available.
*/
// This program will not compile in C99 without this line:
//
int divisibleBy(int x, int y);

int main() {
    printf("%d\n",divisibleBy(16,15));
}

int divisibleBy(int x, int y) {
    printf("%d %d\n", x,y);
    if (x == 0) { return 0; }
    if (y <= 0) { return 0; }
    if (x % y == 0) { return y; }
    return divisibleBy(x, y - 1);
}
```

```
16 15
16 14
16 13
16 12
16 11
16 10
16 9
16 8
8
```

1. Prototypes and corecursive routines

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

```
/* this is a prototype
   it predeclares that a function with this
   name will be available.
   This is useful for co-recursive functions.
*/
// This program will not compile in C99 without this line:
//
int aReliesOnB(int x, int y);
int bReliesOnA(int x, int y);
//


int main() {
    printf("%d\n",aReliesOnB(0,100));
}

int aReliesOnB(int x, int y) {
    printf("> aReliesOnB( %d, %d)\n", x, y);
    if (x >= y) {
        return y;
    }
    return bReliesOnA(x+x+1, y);
}
int bReliesOnA(int x, int y) {
    printf("> bReliesOnA( %d, %d)\n", x, y);
    if (x >= y) {
        return y;
    }
    return aReliesOnB(x * x + 1, y);
}



> aReliesOnB( 0, 100)
> bReliesOnA( 1, 100)
> aReliesOnB( 2, 100)
> bReliesOnA( 5, 100)
> aReliesOnB( 26, 100)
> bReliesOnA( 53, 100)
```

```
> aReliesOnB( 2810, 100)
100
```

1.5.9 Exercise

1. - make a recursive countdown function, printing each number until 0 is met.

```
#include <stdio.h>

void countDown(int n) {
    printf("%d\n",n);
    if (n > 0) {
        countDown(n-1);
    }
}

int main() {
    countDown(10);
    return 0;
}
```

```
10
9
8
7
6
5
4
3
2
1
0
```

2. - make a recursive fibonacci fib(0) = 1 fib(1) = 1 fib(n) = fib(n-1) + fib(n-2)

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

```
int fibonacci(int n) {
   if (n == 0 || n == 1) {
       return 1;
   } else {
       return fibonacci(n-1) + fibonacci(n-2);
   }
}
int main() {
    printf("%d\n",fibonacci(45));
    return 0;
}

1836311903
```

1.6 Scope

1.6.1 const

Instead of define you can use const for constants.

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

const int nine = 9;

int catLives(int ncats) {
    return nine * ncats;
}
int main() {
    printf("10 cats %d lives\n", catLives( 10 ));
    // you can't modify nine
    // nine++;
    // *(&nine) = 10;
    void * totally_not_nine = (void*)&nine;
    int * not_nine = (int *)totally_not_nine;
    *not_nine = 10;
    printf("%d\n",*not_nine);
}
```

1.6.2 Local variables

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

```
#include <stdlib.h>
// no x here
int example(int x) { // < this x is visible -- main's x is NOT visible here
    x++;             // < within
    return x;        // < this scope
}
// no x here
int main() {
    int x = 10;           // < this x is visible within all of main
    printf("x: %d\n", x);
    int rx = example(x);
    printf("x: %d\n", x);
    printf("returned x vs x: %d vs %d\n", rx, x);
}

x: 10
x: 10
returned x vs x: 11 vs 10
```

1.7 Global Variables (BAD) / External Variables / File-level variables

Too common. Too error prone. You will usually cause lots of bugs by making top-level variables. They will only be available within the file you declare.

Global constants are fine. They are safe.

If you make a global in a file, explicitly limit it to the current file with the static keyword.

If static is not used and the variable is in included files then it will be visible across all files.

```
#include <stdio.h>
#include <stdlib.h>
// BAD
// int x = 111; // visible in all lines below unless occluded by local definitions

// BADISH
const int x = 111; // visible in all lines below unless occluded by local definitions


// BETTER but still not OK
```

```
//static int x = 111;

// BEST and allowed
static const int x = 111;

int globalX() {
    return x; // returns the static global x
}

int example(int x) { // <x_2 this x, x_2 is visible -- main's x is NOT visible here nor
    x++;             // <x_2 within
    return x;        // <x_2 this scope
}
int main() {
    printf("Global x %d\n", globalX());
    int x = 10;           // < this x, x_3 is visible within all of main
    const int y = globalX() * globalX();
    printf("y: %d\n", y); // x_3
    printf("x: %d\n", x); // x_3
    int rx = example(x);  // x_3
    printf("x: %d\n", x); // x_3
    printf("returned x vs x: %d vs %d\n", rx, x); // x_3
}
```

1.8 Static Function Scope

Static function local variables keep their old values. It is similar to defining a global per function

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

unsigned int counter() {
    static unsigned int counter = 0; // this keeps its value
    printf("%u\n", counter);
    return ++counter;
}

static unsigned int __worseCounter__ = 0; // whoo don't touch this AKA DONT DO IT
```

```
unsigned int worseCounter() {
    return ++__worseCounter__;
}


#define N 10
int main() {
    srand(time(NULL));
    unsigned int count = 0;
    unsigned int wCount = 0;
    for (int i = 0 ; i < N; i++) {
        if (rand() % 3 == 0) {
            count = counter();
            wCount = worseCounter();
        }
    }
    printf("Counted %u / %u numbers divisible by 3 generated by rand\n", count, N);
    printf("Worse: Counted %u / %u numbers divisible by 3 generated by rand\n", wCount
}
```

```
0
1
2
Counted 3 / 10 numbers divisible by 3 generated by rand
Worse: Counted 3 / 10 numbers divisible by 3 generated by rand
```

1.9 Pointers!

- What is a pointer? A number that is a memory address.
- What's at that memory address? the type of the pointer.
  - char * str;
- Why?
  - you want to know the address so you can manipulate a value or manipulate a shared value.
  - you want to return multiple values from a function.
  - your computer deals with memory as location and offsets the entire time

  - the local variables is the current base pointer + an offset

- What is str? A integer that is a memory address.
- What does str point to? A character, but many an array of characters!
- Can I tell if it is an array of characters? No.
- How can I get the first element of a character array at str?
  - str[0]
  - *str
- How can I make a pointer to:
  - char myChar = 'a';
  - char * ptrToMyChar = &myChar;
- Can I manipulate pointers?
  - char * ptrToChar = &myChar;
  - ptrToChar++; // <— goes to the following character in a character array
  - *ptrToChar = 'b'; // Dereference ptrToChar and chance myChar to the value of 'b'

1.9.1 Operators

- & unary operator means "address of"
- * unary operator means "dereference pointer" – that is return the value it points to
- don't confuse declaration of a variable int * x with dereferencing a variable in an expression: *x

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

// These are macros they cover up syntax
// Return the address of X
#define ADDRESSOF(X) (&X)
```

```
// Dereference X
#define DEREF(X)     (*X)
typedef int * intptr_t;

int main() {
    int i = 99;
    intptr_t ptrToI1 = ADDRESSOF(i); // these 2 lines
    int * ptrToI2 = &i;              // are the same
    printf("i: %4d,\naddress of i:  %p\n\tptrToI1: %p, *ptrToI1: %d\n\tptrToI2: %p, *pt
           i,
           (void*)&i,
           (void*)ptrToI1,
           DEREF(ptrToI1),
           (void*)ptrToI2,
           *ptrToI2
    );
    printf("addressof i: %p,\naddress of ptrToI1:  %p\n\tptrToI2: %p\n",
           (void*)&i,
           (void*)&ptrToI1,
           (void*)&ptrToI2
    );

    return 0;
}

i:   99,
address of i:  0x7ffef7f5a274
ptrToI1: 0x7ffef7f5a274, *ptrToI1: 99
ptrToI2: 0x7ffef7f5a274, *ptrToI2: 99
addressof i: 0x7ffef7f5a274,
address of ptrToI1:  0x7ffef7f5a278
ptrToI2: 0x7ffef7f5a280
```

1.9.2 Character Arrays and Pointers

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

int main() {
```

```
    char myChars[] = "Abram believes he is a benevolent professor";
    // char * strnstr(const char *big, const char *little, size_t len); from string.h
    char * professor = strstr(myChars, "professor");
    char * believes  = strstr(myChars, "believes");
    printf("Size of a pointer %lu\n", sizeof(professor));
    printf("Location pointed to %p\n", professor);
    printf("full representation %016lX\n", (long unsigned int)professor); // look how r
    printf("myChars: %s\n", myChars);
    printf("myChars location: %p\n", myChars);
    printf("professor: %s\n", professor);
    printf("professor location: %p\n", professor);
    printf("believes: %s\n", believes);
    printf("believes location: %p\n", believes);
    printf("believes - myChars location: %llu\n", (long long unsigned int)believes - (l
    printf("professor - myChars location: %llu\n", (long long unsigned int)professor -

    printf("\nBut where are myChars and professor and believes?\n");
    printf("myChars location:   %p\t ptr address: %p \t*ptr %c\n", (void*)&myChars, myC
    printf("professor location: %p\t ptr address: %p \t*ptr %c\n", (void*)&professor, p
    printf("believes location:  %p\t ptr address: %p \t*ptr %c\n", (void*)&believes, be
}

Size of a pointer 8
Location pointed to 0x7ffc5301a2d2
full representation 00007FFC5301A2D2
myChars: Abram believes he is a benevolent professor
myChars location: 0x7ffc5301a2b0
professor: professor
professor location: 0x7ffc5301a2d2
believes: believes he is a benevolent professor
believes location: 0x7ffc5301a2b6
believes - myChars location: 6
professor - myChars location: 34

But where are myChars and professor and believes?
myChars location:   0x7ffc5301a2b0  ptr address: 0x7ffc5301a2b0  *ptr A
professor location: 0x7ffc5301a2a0  ptr address: 0x7ffc5301a2d2  *ptr p
believes location:  0x7ffc5301a2a8  ptr address: 0x7ffc5301a2b6  *ptr b
```

1.9.3 Int arrays

Now character arrays are easy because the size is 1 for a character but what about arrays of larger size datatypes?

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 1000

int main() {
    int myInts[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    // char * strnstr(const char *big, const char *little, size_t len); from string.h
    int * ptrToMyInts = &myInts[0];
    int * five       = &myInts[5];
    int * fiveAgain  = myInts + 5;
    printf("myInts: %p\n", (void*)myInts);
    printf("ptrToMyInts: %p\n", (void*)ptrToMyInts);
    printf("five location:      %p five value:      %d\n", (void*)five, *five);
    printf("fiveAgain location: %p fiveAgain value: %d\n", (void*)fiveAgain, *fiveAgain
    printf("five - myInts location: %llu\n",
           (long long unsigned int)five - (long long unsigned int)myInts);
    printf("five - myInts location / sizeof(int): %llu\n",
           ((long long unsigned int)five - (long long unsigned int)myInts)/(sizeof(int)

    printf("\n OK... Where are they?\n");
    printf("myInts     Location: %p\t ptr address: %p \t*ptr %d\n", (void*)&myInts, (vo
    printf("ptrToMyIntsLocation: %p\t ptr address: %p \t*ptr %d\n", (void*)&ptrToMyInts
    printf("five       Location: %p\t ptr address: %p \t*ptr %d\n", (void*)&five, (voi
    printf("fiveAgain  Location: %p\t ptr address: %p \t*ptr %d\n", (void*)&fiveAgain,

    printf("\nLet's add 1 to five\n");
    int * six = five + 1;
    printf("five       Location: %p\t ptr address: %p \t*ptr %d\n", (void*)&five, (voi
    printf("six        Location: %p\t ptr address: %p \t*ptr %d\n", (void*)&six, (void*


}
```

```
myInts: 0x7ffe80918f50
ptrToMyInts: 0x7ffe80918f50
```

```
five location:      0x7ffe80918f64 five value:      5
fiveAgain location: 0x7ffe80918f64 fiveAgain value: 5
five - myInts location: 20
five - myInts location / sizeof(int): 5

 OK... Where are they?
myInts     Location: 0x7ffe80918f50  ptr address: 0x7ffe80918f50  *ptr 0
ptrToMyIntsLocation: 0x7ffe80918f30  ptr address: 0x7ffe80918f50  *ptr 0
five       Location: 0x7ffe80918f38  ptr address: 0x7ffe80918f64  *ptr 5
fiveAgain  Location: 0x7ffe80918f40  ptr address: 0x7ffe80918f64  *ptr 5

Let's add 1 to five
five       Location: 0x7ffe80918f38  ptr address: 0x7ffe80918f64  *ptr 5
six        Location: 0x7ffe80918f48  ptr address: 0x7ffe80918f68  *ptr 6
```

myInts: 0x7ffd0bdc3f40 ptrToMyInts: 0x7ffd0bdc3f40 five location: 0x7ffd0bdc3f54 five value: 5 fiveAgain location: 0x7ffd0bdc3f54 fiveAgain value: 5 five - myInts location: 20 five - myInts location / sizeof(int): 5

OK... Where are they? myInts Location: 0x7ffd0bdc3f40 ptr address: 0x7ffd0bdc3f40 *ptr 0 ptrToMyIntsLocation: 0x7ffd0bdc3f20 ptr address: 0x7ffd0bdc3f40 *ptr 0 five Location: 0x7ffd0bdc3f28 ptr address: 0x7ffd0bdc3f54 *ptr 5 fiveAgain Location: 0x7ffd0bdc3f30 ptr address: 0x7ffd0bdc3f54 *ptr 5

Let's add 1 to five five Location: 0x7ffd0bdc3f28 ptr address: 0x7ffd0bdc3f54 *ptr 5 six Location: 0x7ffd0bdc3f38 ptr address: 0x7ffd0bdc3f58 *ptr 6

1.9.4 Arrays as pointers

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 1000

int main() {
    int myInts[] = { 99, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int * ptrToMyInts = myInts;
    int * ptrToMyInts2 = &myInts[0];
    printf("myInts:\t%p\n", (void*)myInts);
    printf("ptrToMyInts:\t%p\n", (void*)ptrToMyInts);
    printf("ptrToMyInts2:\t%p\n", (void*)ptrToMyInts2);
```

```
    printf("deref myInts:\t%d\n", *myInts);
    printf("deref ptrToMyInts:\t%d\n", *ptrToMyInts);
    printf("deref ptrToMyInts2:\t%d\n", *ptrToMyInts2);
    return 0;
}

myInts: 0x7ffe24475770
ptrToMyInts: 0x7ffe24475770
ptrToMyInts2: 0x7ffe24475770
deref myInts: 99
deref ptrToMyInts: 99
deref ptrToMyInts2: 99
```

1.9.5 Pointer arthimetic again

When you add to pointers you add not just an integer but your n*sizeof(*p) + p where p is a pointer.

*ptr++ is a common idiom, it means give me the current value and transition to the next memory location.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 1000

int main() {
    long int myInts[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    long int * ptr = &myInts[0];
    size_t count = sizeof(myInts) / sizeof(myInts[0]);
    while(count > 0) {
       printf("%ld \t %p\n", *ptr, (void*)ptr);
       ptr++;
       count--;
    }
    ptr = &myInts[10];
    count = sizeof(myInts) / sizeof(myInts[0]);
    while( count-- > 0) {
       void * oldptr = (void*) ptr;
       printf("%ld \t %p\t", *ptr--, oldptr); // this *ptr++ is
                                              // idiomatic in C and
```

```
                                                // confusing but you must
                                                // learn it
       printf("ptr - oldptr %ld\n", (unsigned long int)ptr - (unsigned long int)oldptr)
    }
    printf("%p %ld\n", (void*)ptr, *ptr);
    return 0;
}

0   0x7fff0c5ec000
1   0x7fff0c5ec008
2   0x7fff0c5ec010
3   0x7fff0c5ec018
4   0x7fff0c5ec020
5   0x7fff0c5ec028
6   0x7fff0c5ec030
7   0x7fff0c5ec038
8   0x7fff0c5ec040
9   0x7fff0c5ec048
10   0x7fff0c5ec050
10   0x7fff0c5ec050 ptr - oldptr -8
9   0x7fff0c5ec048 ptr - oldptr -8
8   0x7fff0c5ec040 ptr - oldptr -8
7   0x7fff0c5ec038 ptr - oldptr -8
6   0x7fff0c5ec030 ptr - oldptr -8
5   0x7fff0c5ec028 ptr - oldptr -8
4   0x7fff0c5ec020 ptr - oldptr -8
3   0x7fff0c5ec018 ptr - oldptr -8
2   0x7fff0c5ec010 ptr - oldptr -8
1   0x7fff0c5ec008 ptr - oldptr -8
0   0x7fff0c5ec000 ptr - oldptr -8
0x7fff0c5ebff8 140733400924160
```

1. Now with Chars

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 1000

int main() {
```

```
    char str[] = "Polar bears are cool bears";
    char * strLiteral = "Polar bears are cool bears";
    char * ptr = str;
    char tmp = 0;
    while( (tmp = *ptr++) ) {
       putchar(tmp);
    }
    putchar('\n');
    ptr = str;
    tmp = 0;
    while( (tmp = *ptr++) ) {
       printf("%c %p %20lu\n", tmp, (void*)ptr, (unsigned long int)ptr);
    }
    // now watch the addresses
    ptr = strLiteral;
    printf("The start of this function's stack frame is pretty close to %p\n", (vo
    while( (tmp = *ptr++) ) {
       printf("%c %p %20lu\n", tmp, (void*)ptr, (unsigned long int)ptr);
    }
    // wow that's super far away in memory
    printf("str - strLiteral in bytes: %lu\n", (unsigned long int)str - (unsigned
    printf("&str - &strLiteral in bytes: %lu\n", (unsigned long int)&str - (unsign
    return 0;
}

Polar bears are cool bears
P 0x7ffc89491f51      140722611756881
o 0x7ffc89491f52      140722611756882
l 0x7ffc89491f53      140722611756883
a 0x7ffc89491f54      140722611756884
r 0x7ffc89491f55      140722611756885
  0x7ffc89491f56      140722611756886
b 0x7ffc89491f57      140722611756887
e 0x7ffc89491f58      140722611756888
a 0x7ffc89491f59      140722611756889
r 0x7ffc89491f5a      140722611756890
s 0x7ffc89491f5b      140722611756891
  0x7ffc89491f5c      140722611756892
a 0x7ffc89491f5d      140722611756893
r 0x7ffc89491f5e      140722611756894
```

```
e 0x7ffc89491f5f      140722611756895
  0x7ffc89491f60      140722611756896
c 0x7ffc89491f61      140722611756897
o 0x7ffc89491f62      140722611756898
o 0x7ffc89491f63      140722611756899
l 0x7ffc89491f64      140722611756900
  0x7ffc89491f65      140722611756901
b 0x7ffc89491f66      140722611756902
e 0x7ffc89491f67      140722611756903
a 0x7ffc89491f68      140722611756904
r 0x7ffc89491f69      140722611756905
s 0x7ffc89491f6a      140722611756906
The start of this function's stack frame is pretty close to 0x7ffc89491f50
P 0x563ee77aa919       94828171536665
o 0x563ee77aa91a       94828171536666
l 0x563ee77aa91b       94828171536667
a 0x563ee77aa91c       94828171536668
r 0x563ee77aa91d       94828171536669
  0x563ee77aa91e       94828171536670
b 0x563ee77aa91f       94828171536671
e 0x563ee77aa920       94828171536672
a 0x563ee77aa921       94828171536673
r 0x563ee77aa922       94828171536674
s 0x563ee77aa923       94828171536675
  0x563ee77aa924       94828171536676
a 0x563ee77aa925       94828171536677
r 0x563ee77aa926       94828171536678
e 0x563ee77aa927       94828171536679
  0x563ee77aa928       94828171536680
c 0x563ee77aa929       94828171536681
o 0x563ee77aa92a       94828171536682
o 0x563ee77aa92b       94828171536683
l 0x563ee77aa92c       94828171536684
  0x563ee77aa92d       94828171536685
b 0x563ee77aa92e       94828171536686
e 0x563ee77aa92f       94828171536687
a 0x563ee77aa930       94828171536688
r 0x563ee77aa931       94828171536689
s 0x563ee77aa932       94828171536690
str - strLiteral in bytes: 45894440220216
```

```
    &str - &strLiteral in bytes: 16
```

1.9.6 Hazel's ptrs.c

The intent here is to demonstrate the use and features of pointers and how to manipulate values via pointers within functions.

```
#include <stdio.h>

int pbv(int passed) {
    passed++;
    printf("   passed = %d\n", passed);
    printf("  &passed = %p\n", (void *) &passed);
    return passed;
}

void pbr(int *passed) {
    printf("   passed = %p\n", (void *) passed);
    printf("  *passed = %d\n", *passed);
    printf("  &passed = %p\n", (void *) &passed);
    (*passed)++;
}

/*
 * 4 byte integer (32-bit PC)
 * Example: our integer uses these 4 bytes
 * byte 4287409512 (0xff8cad68)
 * byte 4287409513 (0xff8cad69)
 * byte 4287409514 (0xff8cad6a)
 * byte 4287409515 (0xff8cad6b)
 */

int main() {
    int thing_1 = 100;
    int thing_2 = 200;

    // type: define a_pointer as a pointer to an int
    int *a_pointer = NULL;
    // type of a_pointer is "int *"
    // NULL: the NULL pointer, gives the pointer the value 0
```

```
// used to indicate that the pointer doesn't point to anything

printf("thing_1 = %d\n", thing_1);
printf("thing_2 = %d\n", thing_2);
// error: 'a_pointer' is used uninitialized in this function [-Werror=uninitialized
//printf("a_pointer = %p\n", (void *) a_pointer);
//printf("a_pointer = %zu\n", (size_t) a_pointer);

printf("\nsizes:\n");
printf("sizeof(thing_1) = %zu\n", sizeof(thing_1));
printf("sizeof(thing_2) = %zu\n", sizeof(thing_2));
printf("sizeof(a_pointer) = %zu (%zu bits)\n", sizeof(a_pointer), sizeof(a_pointer)

// unary & operator: get address of (reference)
a_pointer = &thing_1;

printf("\na_pointer = &thing_1;\n");
printf(" &thing_1 = %p\n", (void *) &thing_1);
printf(" &thing_2 = %p\n", (void *) &thing_2);
printf("a_pointer = %p\n", (void *) a_pointer);
printf("a_pointer = %zu\n", (size_t) a_pointer);
// unary * operator: get value at (dereference)
printf("*a_pointer = %d\n", *a_pointer);

a_pointer = &thing_2;
printf("\na_pointer = &thing_2;\n");
printf("a_pointer = %p\n", (void *) a_pointer);

// unary * operator: get value at (dereference)
printf("*a_pointer = %d\n", *a_pointer);

// We're going to copy thing_1 and take a look
printf("\ncopy value:\n");
printf("\nint value = thing_1;\n");
int value = thing_1;
printf("thing_1 = %d\n", thing_1);
printf("  value = %d\n", value);
printf(" &thing_1 = %p\n", (void *) &thing_1);
printf("   &value = %p\n", (void *) &value);
```

```
    printf("\ncopy value using pointer:\n");
    printf("\nvalue = *(&thing_2);\n");
    value = *(&thing_2);
    printf("thing_2 = %d\n", thing_2);
    printf("  value = %d\n", value);
    printf(" &thing_2 = %p\n", (void *) &thing_2);
    printf("   &value = %p\n", (void *) &value);

    printf("\ncopy value using pointer:\n");
    a_pointer = &thing_2;
    printf("\na_pointer = &thing_2;\n");
    printf("a_pointer = %p\n", (void *) a_pointer);
    // unary * operator: get value at (dereference)
    printf("*a_pointer = %d\n", *a_pointer);
    printf("value = *a_pointer;\n");
    value = *a_pointer;
    printf("thing_2 = %d\n", thing_2);
    printf("  value = %d\n", value);
    printf(" &thing_2 = %p\n", (void *) &thing_2);
    printf("   &value = %p\n", (void *) &value);

    printf("\npass-by-value (copy):\n");
    printf("\npbv(thing_1);\n");
    printf("  thing_1 = %d\n", thing_1);
    printf(" &thing_1 = %p\n", (void *) &thing_1);
    pbv(thing_1);
    printf("  thing_1 = %d\n", thing_1);
    printf(" &thing_1 = %p\n", (void *) &thing_1);

    printf("\npass-by-reference (no copy):\n");
    printf("\npbr(&thing_1);\n");
    printf("  thing_1 = %d\n", thing_1);
    printf(" &thing_1 = %p\n", (void *) &thing_1);
    pbr(&thing_1);
    printf("  thing_1 = %d\n", thing_1);
    printf(" &thing_1 = %p\n", (void *) &thing_1);

    return 0;
}
```

```
thing_1 = 100
thing_2 = 200

sizes:
sizeof(thing_1) = 4
sizeof(thing_2) = 4
sizeof(a_pointer) = 8 (64 bits)

a_pointer = &thing_1;
 &thing_1 = 0x7ffe8deb9864
 &thing_2 = 0x7ffe8deb9868
a_pointer = 0x7ffe8deb9864
a_pointer = 140731279448164
*a_pointer = 100

a_pointer = &thing_2;
a_pointer = 0x7ffe8deb9868
*a_pointer = 200

copy value:

int value = thing_1;
thing_1 = 100
  value = 100
 &thing_1 = 0x7ffe8deb9864
   &value = 0x7ffe8deb986c

copy value using pointer:

value = *(&thing_2);
thing_2 = 200
  value = 200
 &thing_2 = 0x7ffe8deb9868
   &value = 0x7ffe8deb986c

copy value using pointer:

a_pointer = &thing_2;
a_pointer = 0x7ffe8deb9868
*a_pointer = 200
```

```
value = *a_pointer;
thing_2 = 200
  value = 200
 &thing_2 = 0x7ffe8deb9868
   &value = 0x7ffe8deb986c

pass-by-value (copy):

pbv(thing_1);
  thing_1 = 100
 &thing_1 = 0x7ffe8deb9864
   passed = 101
  &passed = 0x7ffe8deb984c
  thing_1 = 100
 &thing_1 = 0x7ffe8deb9864

pass-by-reference (no copy):

pbr(&thing_1);
  thing_1 = 100
 &thing_1 = 0x7ffe8deb9864
   passed = 0x7ffe8deb9864
  *passed = 100
  &passed = 0x7ffe8deb9848
  thing_1 = 101
 &thing_1 = 0x7ffe8deb9864
```

1.9.7 Hazel's $\text{ptr}_{\text{const.c}}$

The intent here is to show that you shouldn't mess with const vars but you can eventually mutate them with pointers.

```
#include <stdio.h>


int main() {
    int mut_i = 100; // mutable integer
    printf("mut_i = %d\n", mut_i);
    const int const_i = 200; // constant integer
    printf("const_i = %d\n", const_i);
```

```
// mutable pointer to mutable integer
int * mut_p = &mut_i;
printf("mut_p = %p\n", (void *) mut_p);
printf("*mut_p = %d\n", *mut_p);
// constant pointer to mutable integer
int * const const_p = &mut_i;
printf("const_p = %p\n", (void *) const_p);
printf("*const_p = %d\n", *const_p);
// mutable pointer to constant integer
const int * p_to_const = &const_i;
printf("p_to_const = %p\n", (void *) p_to_const);
printf("*p_to_const = %d\n", *p_to_const);
// constant pointer to constant integer
const int * const const_p_to_const = &const_i;
printf("const_p_to_const = %p\n", (void *) const_p_to_const);
printf("*const_p_to_const = %d\n", *const_p_to_const);

/*
// Don't do this!
// "warning: assignment discards ‘const’ qualifier from pointer target type"
mut_p = &const_i;

const char *str_lit = "String literals are const char *";
printf("%s\n", str_lit);
// but remember this means we can change str_lit to point to a different string!
str_lit = "String literal #2";
printf("%s\n", str_lit);

// This protects us from:
// str_lit[0] = 'D';

// this is wrong:
char *wrong = "We will try to change this string literal";
printf("%s\n", wrong);

// Because it doesn't protect us from:
// wrong[0] = 'D';
// what happens if you uncomment the above line?
```

```
    // This might be better:
    const char * const RIGHT = "Don't go changing on me!";
    printf("%s\n", RIGHT);
    // Because it protects us from:
    // RIGHT[0] = 'L';
    // and
    // RIGHT = wrong;
    */
}
```

```
mut_i = 100
const_i = 200
mut_p = 0x7ffd165d3e50
*mut_p = 100
const_p = 0x7ffd165d3e50
*const_p = 100
p_to_const = 0x7ffd165d3e54
*p_to_const = 200
const_p_to_const = 0x7ffd165d3e54
*const_p_to_const = 200
```

1.9.8 Hazel's Pointer No No's

ptr_nonos.c

Note the lack of flags below.

```
gcc -std=c99 -Wall -pedantic  -o ptr_nonos ptr_nonos.c && \
./ptr_nonos
```

```
*pointer = 100
    Three fives is 15
*pointer = 22089
    Three fives is 15
  result  = 15
  &result = 0x7ffee57fce84
&result_p = 0x7ffee57fce88
```

```
#include <stdio.h>

#define SIZE 10
```

```
// This function tries to print out the int which is at address 0 in memory...
// Don't do this!
void dereference_null() {
    printf("\ndereference null\n");
    int *a_pointer = NULL;
    printf(" a_pointer = %p\n", (void *) a_pointer);
    printf("*a_pointer = %d\n", *a_pointer);
}

// This function tries to print out the int which is at some address we don't know in r
// Don't do this!
void dereference_uninit() {
    printf("\ndereference unitialized pointer\n");
    int *a_pointer;
    printf(" a_pointer = %p\n", (void *) a_pointer);
    printf("*a_pointer = %d\n", *a_pointer);
}

// This function returns a pointer to an "automatic" local variable...
// Don't do this!
int *return_pointer_to_local() {
    int local_int = 100;
    int *pointer = &local_int;
    // when we return we give up the memory we allocated for "local_int"!
    return pointer;
}

// This function just does some things...
int do_things() {
    int three = 3;
    int five = 5;
    int three_fives = three * five;
    printf("    Three fives is %d\n", three_fives);
    return three_fives;
}

int main() {
//     dereference_null();
    int * pointer = return_pointer_to_local();
    printf("*pointer = %d\n", *pointer);
```

```
    do_things();
    printf("*pointer = %d\n", *pointer);

    // You can't get a pointer to some things...
    // This won't compile:
    // &(do_things());

    // We can't do this for the same reason...
    // &10;

    // This one is actually exactly the same as the one above...
    // &SIZE;

    // You have to make memory to store the value to get a pointer to it!
    int result = do_things();
    printf("  result  = %d\n",           result);
    printf("  &result = %p\n", (void *) &result);

    // This won't compile either. Same reason.
    //     &(&result);
    // You have to make memory to store the pointer to get a pointer to it!
    int * result_p = &result;
    printf("&result_p = %p\n", (void *) &result_p);
    int **result_pp = &result_p;
    int ***result_ppp = &result_pp;
    printf("result_ppp = %p\n", (void *) result_ppp);
    printf("&result_ppp = %p\n", (void *) &result_ppp);
    printf("***result_ppp = %d\n", ***result_ppp);
    return 0;
}

*pointer = 100
    Three fives is 15
*pointer = 21920
    Three fives is 15
  result  = 15
  &result = 0x7ffd1638f9f4
&result_p = 0x7ffd1638f9f8
result_ppp = 0x7ffd1638fa00
&result_ppp = 0x7ffd1638fa08
```

```
***result_ppp = 15
```

1.9.9 Multidimensional Arrays and Pointers

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 10

void init2D(int rows, int cols, int values[][cols]) {
    int i = 0;
    for (int row = 0; row < rows; row++) {
        for (int col = 0; col < cols; col++) {
            values[row][col] = i++;
        }
    }
}

int main() {
    int myInts[N][N];
    init2D(N, N, myInts);
    // int * ptrToMyInts = myInts; // THIS WILL NOT WORK
    int (* ptrToMyInts)[N][N] = &myInts;
    int (* secondRow)[N] = &myInts[1];
    printf("myInts:\t%p\n", (void*)myInts);
    printf("ptrToMyInts:\t%p\n", (void*)ptrToMyInts);
    printf("deref myInts:\t%d\n", **myInts);
    printf("deref myInts + 1:\t%d\n", **(myInts + 1) ); // this hops a row!
    printf("deref secondRow:\t%d\n", *secondRow[0]);
    printf("deref *myInts + 1:\t%d\n", *(*myInts + 1) ); // this hops a col!
    //printf("deref ptrToMyInts:\t%d\n", *ptrToMyInts);
    return 0;
}

myInts: 0x7ffc32f5f8f0
ptrToMyInts: 0x7ffc32f5f8f0
deref myInts: 0
deref myInts + 1: 10
deref secondRow: 10
deref *myInts + 1: 1
```

1.9.10 Arrays of Pointers or Pointers of Pointers

Be aware that when declaring arrays there are arrays of pointers and pointers to arrays.

They are different.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 4
int main() {
   char * ptrs[4]; // an array of character poiunters!
   char stringOnStack[] = "ON STACK";
   // these literals will not be on the stack
   ptrs[0] = "Anaxagoras";
   ptrs[1] = "mummifies";
   ptrs[2] = "shackles";
   ptrs[3] = stringOnStack;
   printf("sizeof(ptrs)=%lu sizeof(ptrs[0])=%lu\n",sizeof(ptrs), sizeof(ptrs[0]));
   printf("sizeof(stringOnStack)=%lu sizeof(stringOnStack[0])=%lu\n",
          sizeof(stringOnStack),
          sizeof(stringOnStack[0]));
   printf("sizeof(&stringOnStack)=%lu sizeof(&stringOnStack[0])=%lu\n",
          sizeof(&stringOnStack),
          sizeof(&stringOnStack[0]));
   for (int i = 0; i < N; i++) {
       printf("S:%s\t", ptrs[i]);
       printf("P:%p\t", (void*)ptrs[i]);
       printf("L:%p\n", (void*)&ptrs[i]);
   }

   char ** pointsToPointers = ptrs; // it is pointers to pointers (like an array!)
   printf("sizeof(pointsToPointers)=%lu sizeof(pointsToPointers[0])=%lu\n",
          sizeof(pointsToPointers),
          sizeof(pointsToPointers[0]));
   puts(*(pointsToPointers + 0));
   puts(pointsToPointers[0]);
   putchar('\n');
   puts(*(pointsToPointers + 2));
   puts(pointsToPointers[2]);
   putchar('\n');
```

```
   return 0;
}

sizeof(ptrs)=32 sizeof(ptrs[0])=8
sizeof(stringOnStack)=9 sizeof(stringOnStack[0])=1
sizeof(&stringOnStack)=8 sizeof(&stringOnStack[0])=8
S:Anaxagoras P:0x55cb7601d978 L:0x7fffba6e7380
S:mummifies P:0x55cb7601d983 L:0x7fffba6e7388
S:shackles P:0x55cb7601d98d L:0x7fffba6e7390
S:ON STACK P:0x7fffba6e73af L:0x7fffba6e7398
sizeof(pointsToPointers)=8 sizeof(pointsToPointers[0])=8
Anaxagoras
Anaxagoras

shackles
shackles
```

1.9.11 Confusing Array Pointer interactions and syntax

- int * myInts != int (* myInts)[]
-

1. Make a pointer to the first element

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

#define N 5

void init2D(int rows, int cols, int values[][cols]) {
    int i = 0;
    for (int row = 0; row < rows; row++) {
        for (int col = 0; col < cols; col++) {
            values[row][col] = i++;
        }
    }
}
```

```
int main() {
    int matrix[N][N];
    init2D( N, N, matrix );
    int * pointToMatrix = &matrix[0][0];
    for (int i = 0; i < N*N; i++) {
        printf("%c", (i%N==0)?'\n':'\t');
        printf("%d", pointToMatrix[i]);

    }
    return 0;
}
```

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

2. Make a pointer to the first row

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

#define N 5
#define M 3

void init2D(int rows, int cols, int values[][cols]) {
    int i = 0;
    for (int row = 0; row < rows; row++) {
        for (int col = 0; col < cols; col++) {
            values[row][col] = i++;
        }
    }
}

int main() {
    int matrix[M][N];
    init2D( M, N, matrix );
```

```
// a pointer to an int array of size [N]
int (* pointToRow)[N] = &matrix[0];
printf("sizeof(pointToRow)=%lu\n", sizeof(pointToRow));
printf("sizeof(pointToRow[0])=%lu\n", sizeof(pointToRow[0]));
printf("Take a ref to row\n");
for (int i = 0; i < M; i++) {
    int * row = pointToRow[i];
    for (int j = 0 ; j < N; j++) {
        printf("%d\t", row[j]);
    }
    printf("\n");
}
printf("Take a ref to row w/ pointer arithmetic\n");
pointToRow = &matrix[0];
for (int i = 0; i < M; i++) {
    int * row = *pointToRow; //deref that row
    pointToRow++; // go to next row
    for (int j = 0 ; j < N; j++) {
        printf("%d\t", row[j]);
    }
    printf("\n");
}
printf("Direct index\n");
pointToRow = &matrix[0];
// direct index
for (int i = 0; i < M; i++) {
    for (int j = 0 ; j < N; j++) {
        printf("%d\t", pointToRow[i][j]);
    }
    printf("\n");
}
printf("Skip a row\n");
// skip a row
pointToRow = &matrix[1];
for (int i = 1; i < M; i++) { // try not to go over our bounds
    int * row = *pointToRow; //deref that row
    pointToRow++; // go to next row
    for (int j = 0 ; j < N; j++) {
        printf("%d\t", row[j]);
    }
```

```
        printf("\n");
    }

    return 0;
}
```

```
sizeof(pointToRow)=8
sizeof(pointToRow[0])=20
Take a ref to row
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
Take a ref to row w/ pointer arithmetic
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
Direct index
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
Skip a row
5 6 7 8 9
10 11 12 13 14
```