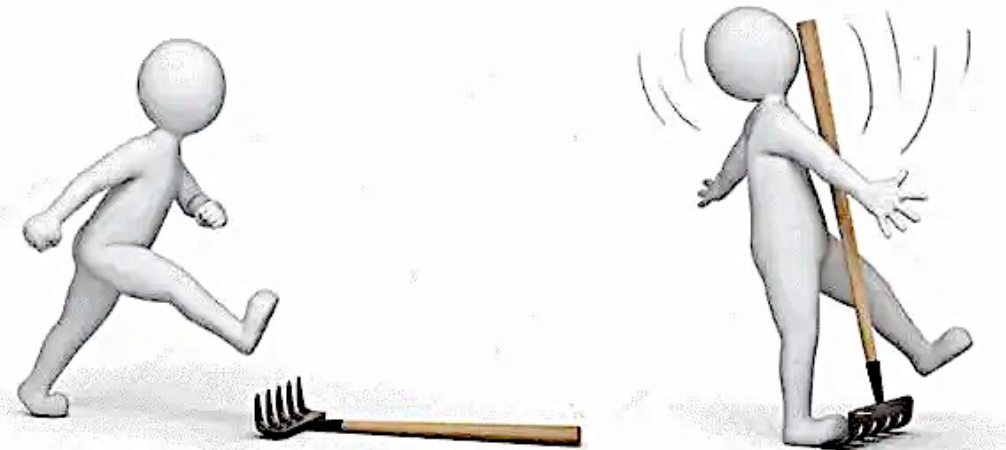
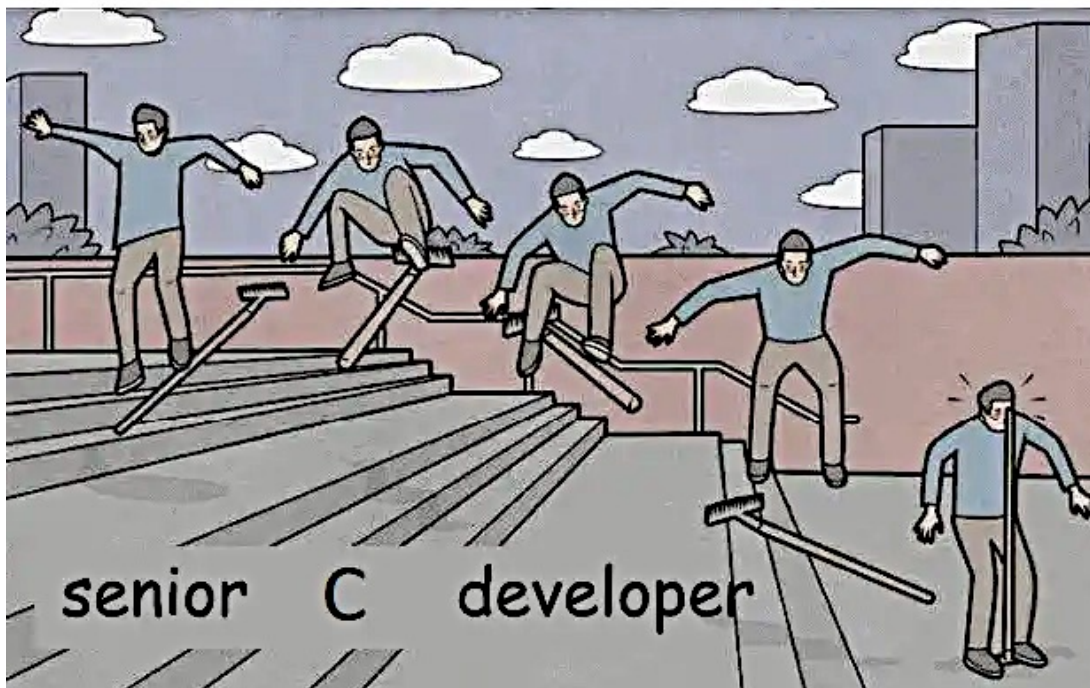
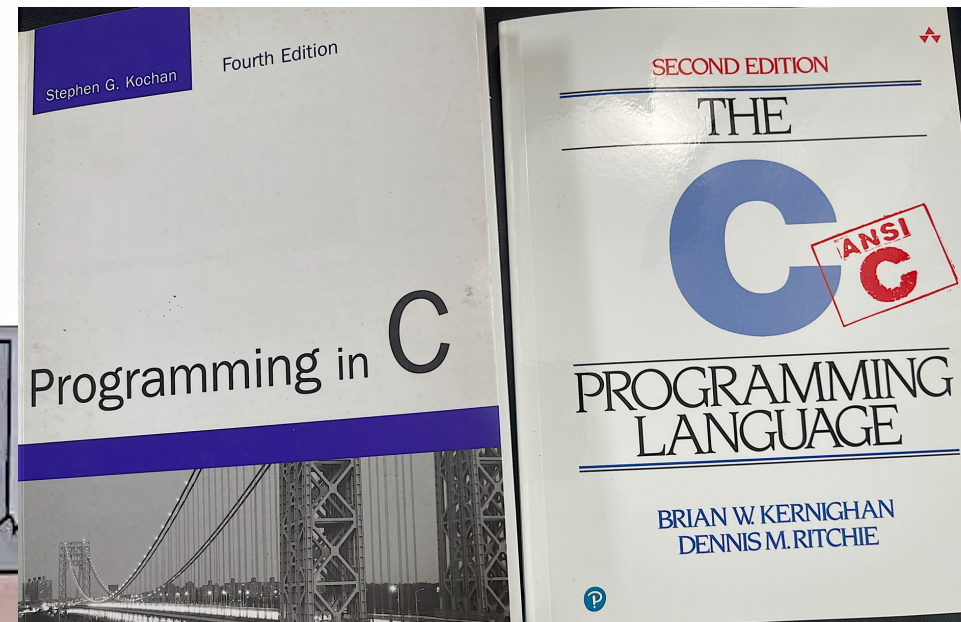


Lecture 10: C part III



junior C developer

Suggested Literature



senior C developer

RECAP: Loops (**while**): interesting case

```
#include <stdio.h>
int main()
{
    printf( "\t--- Calculate Averages ---\n" );
    printf( "Enter some small numbers (type a letter to end input) \n" );

    double x;
    double sum = 0;
    int count = 0;

    while (scanf("%lf", &x) == 1)
    {
        printf("x = %lf\n", x);
        sum += x;
        ++count;
    }
    if (count == 0)
    {
        printf("No input data!\n" );
    }
    else
    {
        printf("Average = %.2f\n", sum/count );
    }
    return 0;
}
```

scanf does return an integer!
scanf returns the number of
fields that were successfully
converted and assigned



RECAP: Arrays

Even worse than segfault is when C just return garbage (out of bounds)

```
#include <stdio.h>

int main (void)
{
    int x[5];

    x[0] = 197;
    x[1] = 200;
    x[2] = -100;
    x[3] = x[0] + x[1];
    x[4] = 350;

    for (int i=0; i<100; ++i)
    {
        printf ("x[%i] = %i\n", i, x[i]);
    }
    return 0;
}
```

out of bounds access to
array is **undefined**
behavior (it could cause
a segmentation fault, but
it doesn't have to)

```
whovian@phy504:~/hos
whovian@phy504:~/hos
x[0] = 197
x[1] = 200
x[2] = -100
x[3] = 397
x[4] = 350
x[5] = 64
x[6] = 946356992
x[7] = 244184714
x[8] = 0
x[9] = 0
x[10] = 25616515
x[11] = 64
x[12] = 107
x[13] = 1
x[14] = 25187512
x[15] = 64
x[16] = 27469728
x[17] = 1
x[18] = 4457
```

RECAP: Functions in C

Function return values (just one!)

```
#include <stdio.h>
double KmhToMph (const double kmh)
{
    return kmh * 0.62137119223;
}

int main ()
{
    printf("1 km/h corresponds to "
           "%lf miles/h \n", KmhToMph(1));

    return 0;
}
```

return type

Having “magic numbers” out of thin air is bad coding practice.

Btw: not enough digits to maintain double precision

```
whovian@phy504:~/host$ gcc functions4.c -o functions4
whovian@phy504:~/host$ ./functions4
1 km/h corresponds to 0.621371 miles/h
```

RECAP: Functions in C

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

```
int min (int vals[10])
{
    int tmp = vals[0];

    for (int i=1; i<10; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }
    return tmp;
}
```

```
int main (void)
{
    int scores[10] = {2, 1, 3, 4, 5,
                     6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(scores));

    int scores2[5] = {2, 1, 3, 4, 5};
    printf ("Minimum score is %i\n", min(scores2));

    int scores3[11] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8, 0};
    printf ("Minimum score is %i\n", min(scores3));

    return 0;
}
```

mildly dangerous

(but important case): arrays

```
whovian@phy504:~/host$ ./functions5
```

```
Minimum score is 1
```

```
Minimum score is 0
```

```
Minimum score is 1
```



wrong



wrong

UNDEFINED BEHAVIOUR

May be right/wrong – who knows? Nobody! Undefined

Wrong – but behavior is predictable

RECAP: Functions in C

```
#include <stdio.h>

int min (int n, int vals[])
{
    if (n <= 0)
    {
        printf ("Error, size = %i \n", n);
    }

    int tmp = vals[0];

    for (int i=1; i<n; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }

    return tmp;
}

int main (void)
{
    int scores[10] = {2, 1, 3, 4, 5,
                     6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(10, scores));

    int scores2[5] = {2, 1, 3, 4, 5};
    printf ("Minimum score is %i\n", min(5, scores2));

    int scores3[11] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8, 0};
    printf ("Minimum score is %i\n", min(11, scores3));

    return 0;
}
```

Check
input

With VLAs and an extra integer argument, we can fix the problem

```
whovian@phy504:~/host$ ./functions5
Minimum score is 1
Minimum score is 1
Minimum score is 0
```



Static variables (in functions)

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

```
int fun(void)
```

```
{  
    static int count = 0;
```

```
    count++;
```

```
    return count;
```

```
}
```

```
int main(void)
```

```
{
```

```
    printf("count = %d \n", fun());
```

```
    printf("count = %d \n", fun());
```

```
    printf("count = %d \n", fun());
```

```
    printf("count = %d \n", fun());
```

```
    return 0;
```

```
}
```

extremely
dangerous in
parallel computation

```
whovian@phy504:~/host$ gcc functions9.c -o functions9
```

```
whovian@phy504:~/host$ ./functions9
```

```
count = 1
```

```
count = 2
```

```
count = 3
```

```
count = 4
```

Dangerous, but can
be useful. My
research code
contains them

Variable remember
values between calls

Static variables
inside functions have
memory

They last between calls

Error Handling

Code should always check for errors constantly.

There are many ways to do it

One is the “**let it crash**” approach. Code that crashes with exit code `!= 0` is usually much better than code that runs with wrong/undefined behavior

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

int fun(int posnum)
{
    if (posnum <= 0)
    {
        exit(1);
    }
    // code that requires posnum > 0
    // ....
    // return some result
    int result = 10;
    return result;
}
```

```
int main(void)
{
    fun(-10);
    return 0;
}
```

whovian@phy504:~/host\$ gcc functions10.c -o functions10
whovian@phy504:~/host\$ echo "exit code \$?"
exit code 0
whovian@phy504:~/host\$./functions10
whovian@phy504:~/host\$ echo "exit code \$?"
exit code 1

Error Handling

C provides the assert function

Nice feature about assert: it provides the file name and line where the code failed

Assert also writes the check that failed. In this case **posnum > 0**

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

int fun(int posnum)
{
    assert (posnum > 0);

    // code that requires posnum > 0
    // ....
    // return some result
    int result = 10;
    return result;
}

int main(void)
{
    fun(-10);
    printf("End of main\n");
    return 0;
}
```

```
whovian@phy504:~/host$ gcc functions11.c -o functions11
whovian@phy504:~/host$ ./functions11
functions11: functions11.c:6: fun: Assertion `posnum > 0' failed.
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted
whovian@phy504:~/host$ echo "exit code $?"
exit code 134
```

Global Variables

Global variables are evil (sometimes unavoidable evil!)

```
#include <stdio.h>

int count;

int fun(void)
{
    count++;
    return count;
}

int main(void)
{
    int count = 0;
    printf("count = %d \n", fun());
    printf("count = %d \n", fun());
    printf("count = %d \n", fun());
    printf("count = %d \n", fun());
    return 0;
}
```

Everyone has access to Global Variables



Global Variables

Global variables are evil (sometimes unavoidable evil!)

```
#include <stdio.h>

// source:
// https://www.geeksforgeeks.org/global-variables-in-c/

int a, b; // global variables

void add()
{
    printf("%d", a + b);
}

int main()
{
    a = 10;
    b = 15;
    add();
    return 0;
}
```

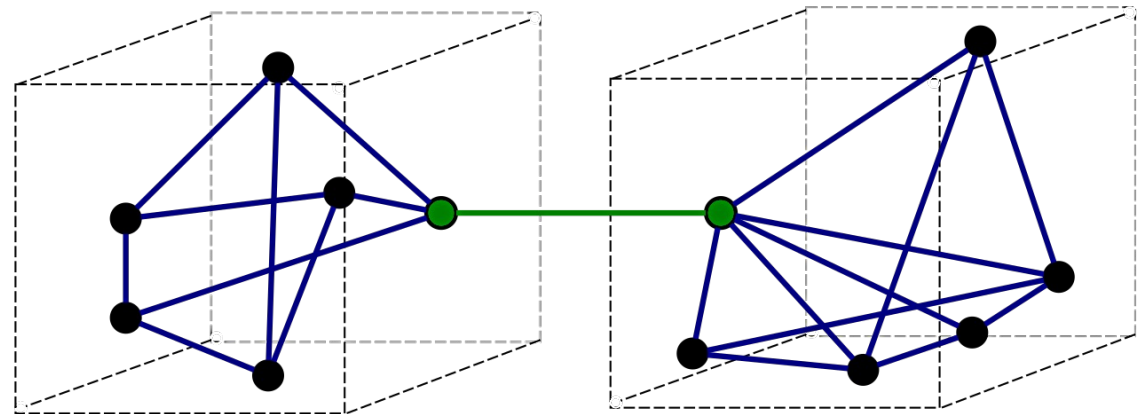
Global Variables

Why Global variables are Evil? Answers from SO

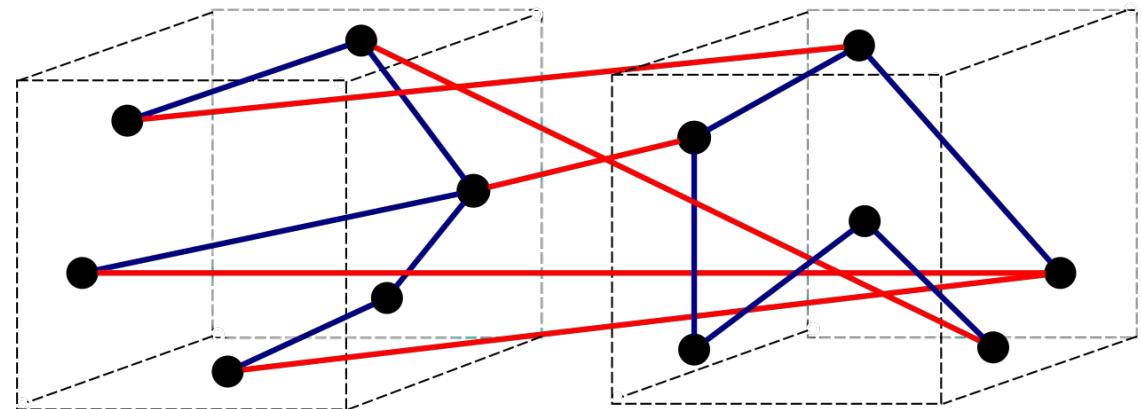
“The problem with global variables is that since every function has access to these, it becomes increasingly hard to figure out which functions actually read and write these variables” Source: <https://stackoverflow.com/a/485020/2472169>

The problem that global variables create for the programmer is that it expands the inter-component coupling surface between the various components that are using the global variables.

Source: <https://stackoverflow.com/a/24501533/2472169>



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

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

```
int main()
```

```
{
```

```
    enum day {  
        sunday,  
        monday,  
        tuesday,  
        wednesday,  
        thursday,  
        friday,  
        saturday
```

```
};
```

```
enum day d = thursday;
```

```
printf("The day number stored in d is %d\n", d);
```

```
enum day f = wednesday;
```

```
printf("The day number stored in f is %d\n", f);
```

```
if (d == f)
```

```
{
```

```
    printf("d and f represent the same day\n");
```

```
}
```

```
else
```

```
{
```

```
    printf("d and f represent different days\n");
```

```
}
```

```
return 0;
```

```
}
```

**If you don't specify
the vars values,
compiler
automatically write 0,
1, 2, 3...**

Working with Structures: **enum**

enum is a set of
integer variables
with names that,
in some ways,
vaguely look like
a type

C **enum** only
holds **integers**

```

#include <stdio.h>

int main()
{
    enum day {
        sunday,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday,
        saturday
    };
    enum day d = thursday;
    printf("The day number stored in d is %d\n", d);
    enum day f = wednesday;
    printf("The day number stored in f is %d\n", f);

    if (d == f)
    {
        printf("d and f represent the same day\n");
    }
    else
    {
        printf("d and f represent different days\n");
    }
    return 0;
}

```

```

whovian@phy504:~/host/C$ gcc enum1.c -o enum1
whovian@phy504:~/host/C$ ./enum1
The day number stored in d is 4
The day number stored in f is 3
d and f represent different days

```

Working with Structures:

enum

C **enum** only
holds **integers**

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

```
// an interesting example from ChatGPT
```

```
enum Error {
```

```
    SUCCESS,
```

```
    FILE_NOT_FOUND,
```

```
    PERMISSION_DENIED,
```

```
    NETWORK_ERROR,
```

```
    OUT_OF_MEMORY
```

```
};
```

```
const char* error_messages[] = {
```

```
    "Success",
```

```
    "File not found",
```

```
    "Permission denied",
```

```
    "Network error",
```

```
    "Out of memory"
```

```
};
```

```
int main() {
```

```
    enum Error error = FILE_NOT_FOUND;
```

```
    printf("Error: %s\n", error_messages[error]);
```

```
    return 0;
```

```
}
```

Working with Structures: **enum**

C **enum** can provide
“names” to positions
in arrays. An
interesting application
is for printing error
messages

Switch statements (with **enums**)

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

```
int main()  
{
```

```
    enum day {  
        sunday,  
        monday,  
        tuesday,  
        wednesday,  
        thursday,  
        friday,  
        saturday
```

```
};
```

```
enum day e = wednesday;
```

```
switch (e)  
{
```

```
    case sunday:
```

```
    {
```

```
        printf("Today is Sunday\n");
```

```
        break;
```

```
    }
```

```
    case monday:
```

```
    {
```

```
        printf("Today is Monday\n");
```

```
        break;
```

```
    }
```

```
    case tuesday:
```

```
    {
```

```
        printf("Today is Tuesday\n");
```

```
        break;
```

```
    }
```

```
    case wednesday:
```

```
whovian@phy504:~/host/C$ gcc enum2.c -o enum2
```

```
whovian@phy504:~/host/C$ ./enum2
```

```
Today is Wednesday
```

{ } are not mandatory, but
I always include them
(they isolate local variables
you may create inside each
case)

enums is a bit weird

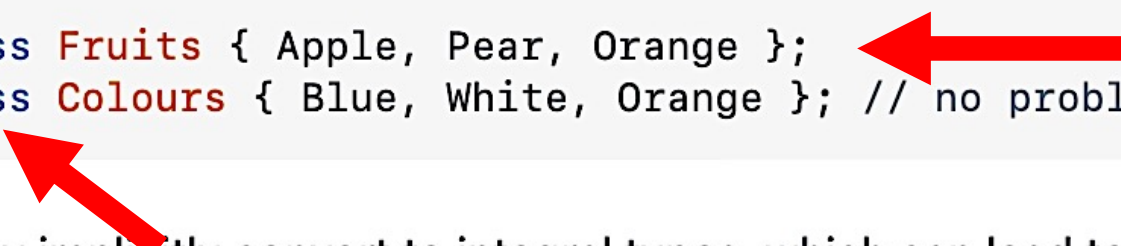
C **enum** is not strongly typed ([a mistake that C++ fixed](#))

OK, first example: old-style enums do not have their own scope:

```
enum Animals {Bear, Cat, Chicken};  
enum Birds {Eagle, Duck, Chicken}; // error! Chicken has already been declared!  
  
enum class Fruits { Apple, Pear, Orange };  
enum class Colours { Blue, White, Orange }; // no problem!
```

This is C

This is C++



Second, they implicitly convert to integral types, which can lead to strange behaviour:

```
bool b = Bear && Duck; // what?
```

This is bad!!!



Because of these problems, I avoid C **enum**,
but I am perfectly happy with C++ **enum**

typedef keyword

Assign an alternate name to a data type

```
#include <stdio.h>

typedef int chicken;
int main (void)
{
    chicken x = 100;
    printf ("Physics 504 is fun. Your grade %i\n", x);
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc typedef.c -o typedef
whovian@phy504:~/host/C$ ./typedef
Physics 504 is fun. Your grade 100
```

struct in C

struct allow C to do
Object Oriented
Program (OOP)

C++ will take structs
to another level with
Classes

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

typedef struct
{
    double x;
    double y;
} Point;

double distance (Point p1, Point p2)
{
    const double diffx = p1.x - p2.x;
    const double diffy = p1.y - p2.y;
    return sqrt (diffx * diffx + diffy * diffy);
}

int main ()
{
    Point p1;
    p1.x = 0.0;
    p1.y = 1.0;

    Point p2;
    p2.x = 1.0;
    p2.y = 0.0;

    printf("P1 = (%.2f, %.2f)\n", p1.x, p1.y);
    printf("P2 = (%.2f, %.2f)\n", p2.x, p2.y);
    printf("|P1-P2| = %.4f \n", distance(p1, p2));
}
```

```
whovian@phy504:~/host$ gcc struct.c -o struct -lm
whovian@phy504:~/host$ ./struct
P1 = (0.00, 1.00)
P2 = (1.00, 0.00)
|P1-P2| = 1.4142
```

struct in C

struct works like any type. For example, you can make arrays

PS: why not

birthday[0].name = "Viv"
array type is not assignable

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

typedef struct
{
    char name[100];
    int day;
    int month;
    int year;
} Date;

void print_date(Date x)
{
    printf("Name: %s, Birthday: %.2d-%.2d-%.4d\n",
        x.name, x.month, x.day, x.year);
}

int main ()
{
    Date birthdays[2];
    strcpy(birthdays[0].name, "Vivian Miranda");
    birthdays[0].day = 28;
    birthdays[0].month = 02;
    birthdays[0].year = 1986;

    strcpy(birthdays[1].name, "Taylor Swift");
    birthdays[1].day = 13;
    birthdays[1].month = 12;
    birthdays[1].year = 1989;

    print_date(birthdays[0]);
    print_date(birthdays[1]);
}
```

```
whovian@phy504:~/host$ gcc struct2.c -o struct2
whovian@phy504:~/host$ ./struct2
Name: Vivian Miranda, Birthday: 02-28-1986
Name: Taylor Swift, Birthday: 12-13-1989
```

struct in C (user-defined type)

Be careful with overflow – undefined behavior

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

typedef struct
{
    char name[10];
    int day;
    int month;
    int year;
} Date;

void print_date(Date x)
{
    printf("Name: %s, Birthday: %.2d-%2d-%4d\n",
        x.name, x.month, x.day, x.year);
}

int main ()
{
    Date birthdays[2];
    strcpy(birthdays[0].name, "Vivian Miranda Bla bla bla bla bla bla bla ok");
    birthdays[0].day = 28;
    birthdays[0].month = 02;
    birthdays[0].year = 1986;

    print_date(birthdays[0]);
}
```

whovian@phy504:~/host\$ gcc struct2.c -o struct2
whovian@phy504:~/host\$./struct2
Name: Vivian Miran, Birthday: 02-28-1986



struct in C

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

```
typedef struct
```

```
{
```

```
    char name[100];
```

```
    int day;
```

```
    int month;
```

```
    int year;
```

```
} Date;
```

```
void print_date(Date x)
```

```
{
```

```
    printf("Name: %s, Birthday: %.2d-%.2d-%.4d\n",
```

```
           x.name, x.month, x.day, x.year);
```

```
}
```

```
int main ()
```

```
{
```

```
    Date test = {"Vivian Miranda", 28, 02, 1986};
```

```
    print_date(test);
```

```
/* Code below does not work, it is not initialization
```

```
Code below is assignment.
```

```
Initialization != assignment
```

```
Date birthdays[2];
```

```
birthdays[0] = {"Vivian Miranda", 28, 02, 1986};
```

```
*/
```

```
}
```

```
whovian@phy504:~/host$ gcc struct3.c -o struct3
```

```
whovian@phy504:~/host$ ./struct3
```

```
Name: Vivian Miranda, Birthday: 02-28-1986
```

Initialization
not the same
as assignment

struct in C

OOP style

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

```
typedef struct
```

```
{
```

```
    char name[100];
```

```
    int day;
```

```
    int month;
```

```
    int year;
```

```
} Date;
```

```
whovian@phy504:~/host$ gcc struct4.c -o struct4
```

```
whovian@phy504:~/host$ ./struct4
```

```
Date = 28
```

```
int get_day(const Date* const self)
```

```
{
```

```
    return self->day;
```

```
}
```

```
void set_day(Date* const self, const int day)
```

```
{
```

```
    self->day = day;
```

```
}
```

```
int main()
```

```
{
```

```
    Date x;
```

```
    set_day(&x, 28);
```

```
    printf("Date = %.2d\n", get_day(&x));
```

```
}
```

In OOP, we don't change member variables directly. We use functions

First time seeing pointers and the ability to change function arguments passed by value

Nested struct

Nested struct is a good way to make struct manageable

```
typedef struct
```

```
{  
    int day;  
    int month;  
    int year;  
}
```

```
Date;
```

```
typedef struct
```

```
{  
    char name[100];  
    Date bday;
```

```
Student;
```

```
int main ()
```

```
whovian@phy504:~/host$ gcc struct5.c -o struct5
```

```
whovian@phy504:~/host$ ./struct5
```

```
Date = 28
```

```
Student x;
```

```
strcpy(x.name, "Vivian Miranda");
```

```
x.bday.day = 28;
```

```
x.bday.month = 2;
```

```
x.bday.year = 1986;
```

```
printf("Date = %.2d\n", x.bday.day);
```

Union and Anonymous Unions

advanced
feature:
types stored
in the same
memory
location

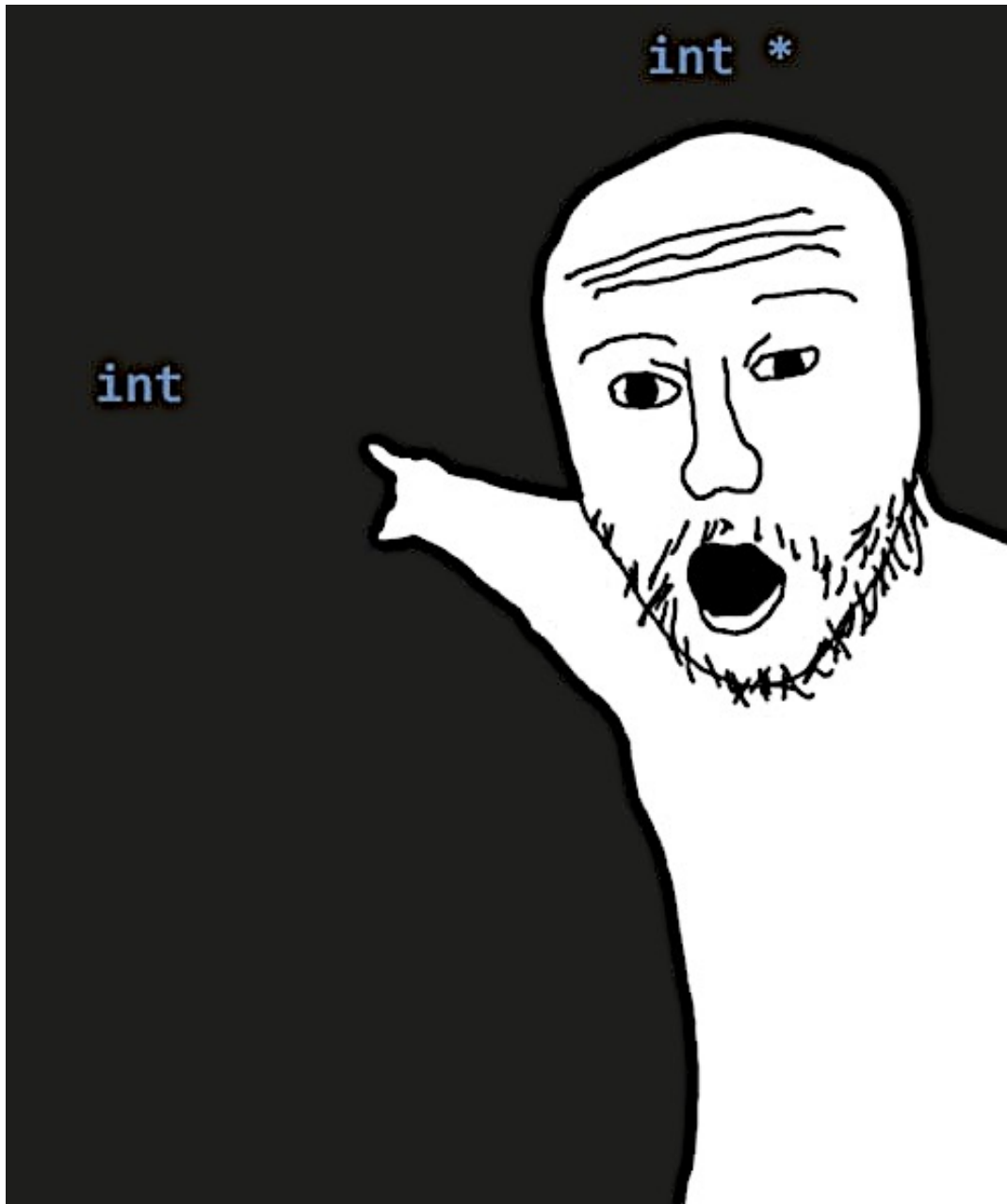
```
#include <stdio.h>

typedef struct
{
    union
    {
        struct
        {
            double x;
            double y;
            double z;
        };
        double raw[3];
    };
} vec3d;
```

```
int main ()
{
    vec3d v;
    v.x = 4.0;
    v.raw[1] = 3.0; // Equivalent to v.y = 3.0
    v.z = 2.0;

    printf("(x,y,z) = (%.2f,%.2f,%.2f) \n", v.x, v.y, v.z);
    // an alternative way to get the same data
    printf("(x,y,z) = (%.2f,%.2f,%.2f) \n", v.raw[0], v.raw[1], v.raw[2]);
}
```

```
whovian@phy504:~/host$ gcc union1.c -o union1
whovian@phy504:~/host$ ./union1
(x,y,z) = (4.00,3.00,2.00)
(x,y,z) = (4.00,3.00,2.00)
```



pointers in C

Pointers

Pointers point to the memory location of a variable



```
#include <stdio.h>

int main ()
{
    int x = 10;
    int* y = &x;

    printf("x = %d\n", x);
    printf("The value of pointer y = %d\n", *y);
    printf("The address of x = %p\n", (void*) &x);
    printf("The address of y = %p\n", (void*) y);
}
```

whovian@phy504:~/host/C\$ gcc pointers1.c -o pointers
whovian@phy504:~/host/C\$./pointers
x = 10
The value of pointer y = 10
The address of x = 0x400180539c
The address of y = 0x400180539c

indirection operator

A red arrow pointing from the text "indirection operator" to the asterisk in "*y" in the printf statement.

Memory mapping

Mapping is written in Hexadecimals

0xNum implies **Num** is in Hexadecimal

```
whovian@phy504:~/host/C$ gcc pointers1.c -o pointers
whovian@phy504:~/host/C$ ./pointers
x = 10
The value of pointer y = 10
The address of x = 0x400180539c
The address of y = 0x400180539c
```

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

convenience: nums shorter and easy to convert to binary

Conversion from [0-F] to binary takes 4 bits

Therefore, **0xE2** = $14 * 16 + 2 = \underline{1110} \underline{0010}$ (binary)

E 2

Memory mapping

Mapping is written in Hexadecimals

0xNum implies **Num** is in Hexadecimal

```
whovian@phy504:~/host/C$ gcc pointers1.c -o pointers
whovian@phy504:~/host/C$ ./pointers
x = 10
The value of pointer y = 10
The address of x = 0x400180539c
The address of y = 0x400180539c
```

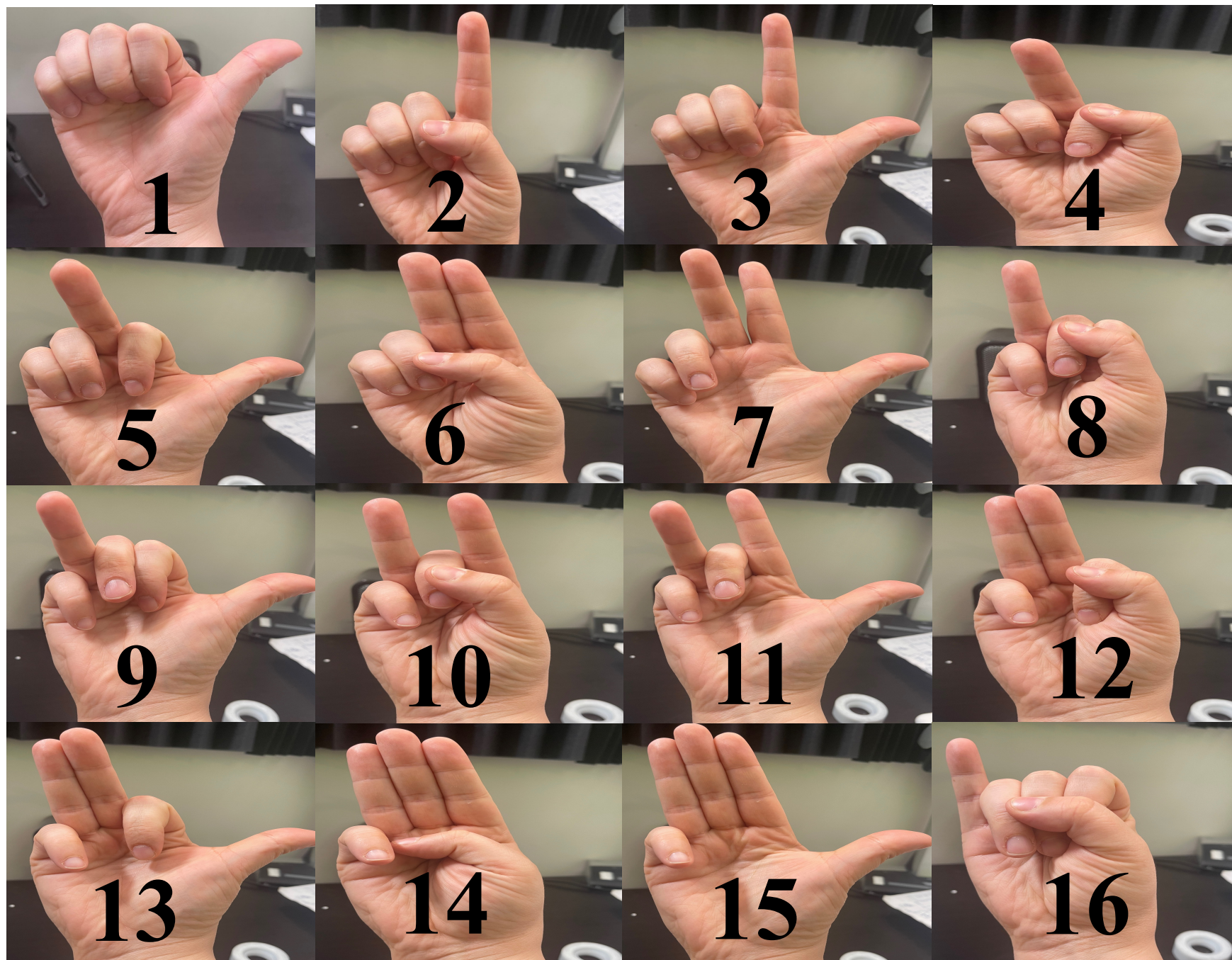
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

More complicated: **0xAE0** to binary

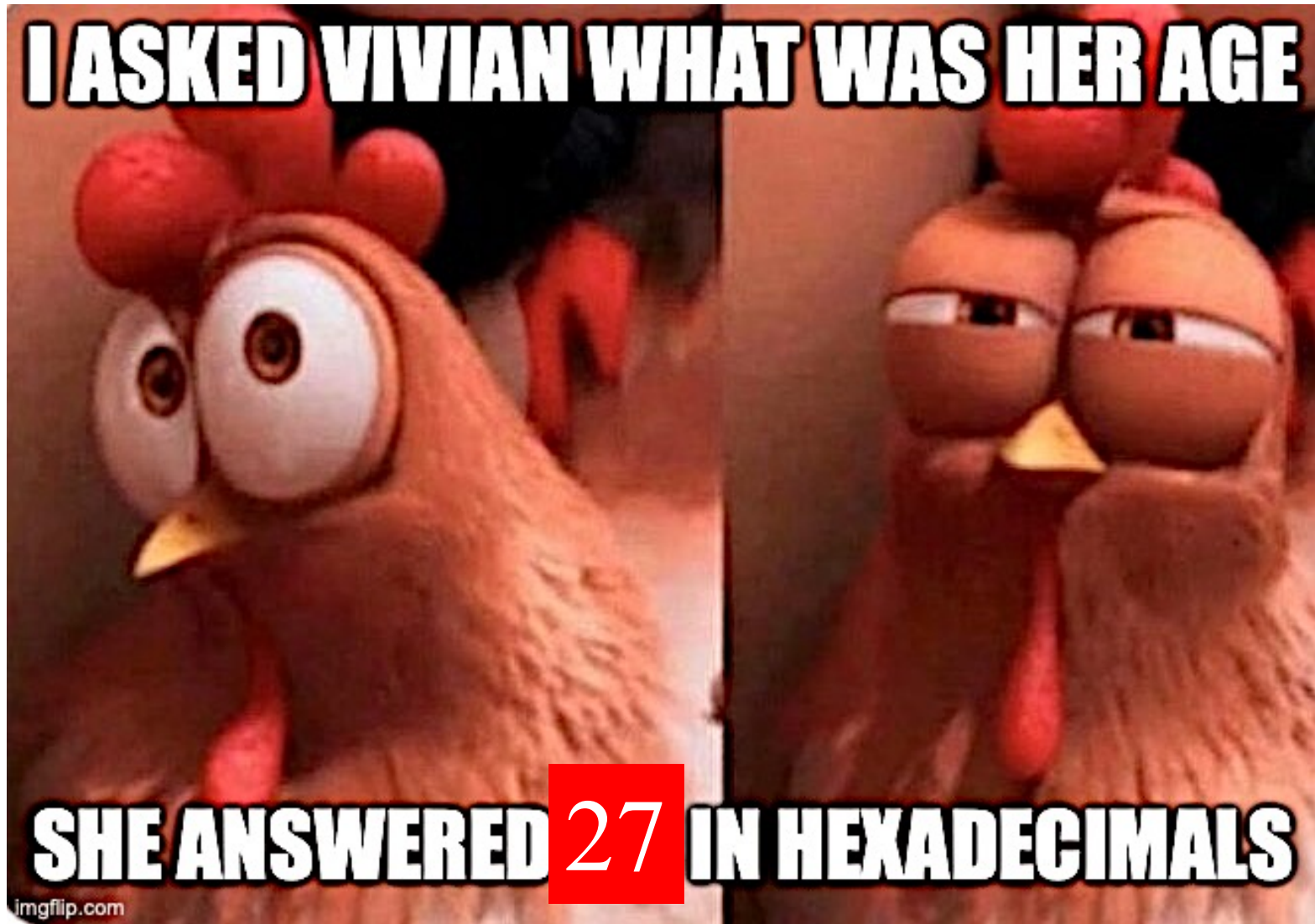
Conversion from [0-F] to binary takes 4 bits

Therefore, **0xAE2** = 1010 1110 0010 (binary)
A E 2

Practice 4 bits binary with one hand



I only answer my age in Hexadecimals



Pointers

Pointers keep track of the value of the variable it points to

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

```
int main ()  
{  
    int x = 10;  
    int* y = &x;  
    int z = *y;  
    x++;  
  
    printf("x = %d\n", x);  
    printf("y = %d\n", *y);  
    printf("z = %d\n", z);  
}
```

```
whovian@phy504:~/host/C$ ./pointers2
```

```
x = 11
```

```
y = 11
```

```
z = 10
```

How? **int* y = &x;**
copies the address of
the memory reserved
for the variable **x**,
not the value
assigned to **x**

Pointers

Pointers keep track of the value of the variable it points to

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

```
int main ()
```

```
{
```

```
    int x = 10;
```

```
    int* y = &x;
```

```
    int z = *y;
```

```
    x++;
```

```
    printf("x = %p\n", (void*) &x);
```

```
    printf("y = %p\n", (void*) y);
```

```
    printf("z = %p\n", (void*) &z);
```

```
}
```

```
whovian@phy504:~/host/C$ gcc pointers3.c -o pointers3
```

```
whovian@phy504:~/host/C$ ./pointers3
```

```
x = 0x4001805398
```

```
y = 0x4001805398
```

```
z = 0x400180539c
```



How?

int* y = &x;

copies the
address, not the
value!

Pointers

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

```
int main (void)  
{
```

```
    char c = 'Q';  
    char* ptr = &c;
```

```
    printf ("%c %c\n", c, *ptr);
```

```
    c = '/';  
    printf ("%c %c\n", c, *ptr);
```

```
    *ptr = 'C';  
    printf ("%c %c\n", c, *ptr);
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host/C$ gcc pointers4.c -o pointers4
```

```
whovian@phy504:~/host/C$ ./pointers4
```

```
Q Q
```

```
/ /
```

```
C C
```



Pointers allow you to
change the original
value as well

They are “in sync”
because they get the
value stored in the
same memory
location

Homework I

- Write a program in C to copy the elements of one array into another array
- Write a program in C to count the total number of duplicate elements in an array.
- Write a program in C to find an array's maximum and minimum element.
- Write a program in C to separate odd and even integers in separate arrays.
- Write a program in C to count the frequency of each element of an array.

Homework II

- Write a program in C that checks if a partially assigned 9x9 Sudoku game is a valid Sudoku solution. Hint: Pseudocode in the picture

```
Algorithm isValidSudoku(board):
    // Check rows and columns simultaneously.
    For i = 0 to 8:
        rowSeen[1..9] ← false, colSeen[1..9] ← false
        For j = 0 to 8:
            if board[i][j] ≠ 0 then
                if rowSeen[board[i][j]] then return false
                else rowSeen[board[i][j]] ← true
            if board[j][i] ≠ 0 then
                if colSeen[board[j][i]] then return false
                else colSeen[board[j][i]] ← true

    // Check 3x3 sub-boxes.
    For boxRow = 0 to 2:
        For boxCol = 0 to 2:
            seen[1..9] ← false
            For i = 0 to 2:
                For j = 0 to 2:
                    digit ← board[boxRow*3 + i][boxCol*3 + j]
                    if digit ≠ 0 then
                        if seen[digit] then return false
                        else seen[digit] ← true

    return true
```

Explanation

- **Rows and Columns:**

For each index `i`, we initialize two arrays (`rowSeen` and `colSeen`) to track digits in the current row and column. We loop through each cell in the row and column simultaneously, returning false if a duplicate (nonzero) is found.

- **Sub-boxes:**

The board is divided into nine 3x3 boxes. For each box (determined by `boxRow` and `boxCol`), we track digits in a `seen` array and check all 9 cells. If any duplicate is found, the board is invalid.

Homework III

Question 1

Consider the following declarations and assignments.

```
int *a, b[5], c, *d;
```

```
for (c = 0; c < 5; c++) {  
    b[c] = 1+c;  
}
```

```
d = b;
```

```
a = &c;
```

```
c = d[3];
```

What are the TYPE and VALUE of each of the following expressions (if the expression is invalid, write “Illegal Expression”, and if it is an address describe what it is the address of):

	TYPE	VALUE
1. a	-----	-----
2. b	-----	-----
3. c	-----	-----
4. &b[1]	-----	-----
5. d	-----	-----
6. *d	-----	-----

Homework IV

For the next three questions, consider the following short program:

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

int main(void)
{
    char* pC = NULL;
    pC = (char*) malloc(sizeof(char));
    *pC = 'x';
    return 0;
}
```

1. Which of the following best describes the situation after Line 1 has been executed?
 - 1) pC does not have a target, and pC does not have a known value.
 - 2) pC does not have a target, but pC does have a known value.
 - 3) pC has a target, but its target does not have a known value.
 - 4) pC has a target, and its target does have a known value.
 - 5) None of these
2. Which of the following best describes the situation after Line 2 has been executed?
 - 1) pC does not have a target, and pC does not have a known value.
 - 2) pC does not have a target, but pC does have a known value.
 - 3) pC has a target, but its target does not have a known value.
 - 4) pC has a target, and its target does have a known value.
 - 5) None of these
3. Which of the following best describes the situation after Line 3 has been executed?
 - 1) pC does not have a target, and pC does not have a known value.
 - 2) pC does not have a target, but pC does have a known value.
 - 3) pC has a target, but its target does not have a known value.
 - 4) pC has a target, and its target does have a known value.
 - 5) None of these

Homework V

- C program to swap two numbers (integers) using pointers
- Write a program called **mysort** that sort an integer array. The program should take a function pointer that tells the **mysort** function how integers are compared as shown below

void mysort(int* arr, int size, int (*compare)(int*, int*))

- Write a C program to return multiple value from function.

Homework VI

```
#include <stdio.h>

typedef struct
{
    int rows;
    int cols;
    double data[10000];
} Matrix;

// implement the following functions

// double get_elems(Matrix a, const int row, const int col)
// that returns a(i,j).
// The function should check if row and col are out of bounds

// void set_elems(Matrix a, const int row, const int col)
// The function should check if row and col are out of bounds

// Matrix msum(Matrix a, Matrix b)
// The function should check if a and b have same rows and cols

// Matrix mmult(Matrix a, Matrix b)
// The function should check if a can be multiplied by b

int main()
{
    return 0.0;
}
```