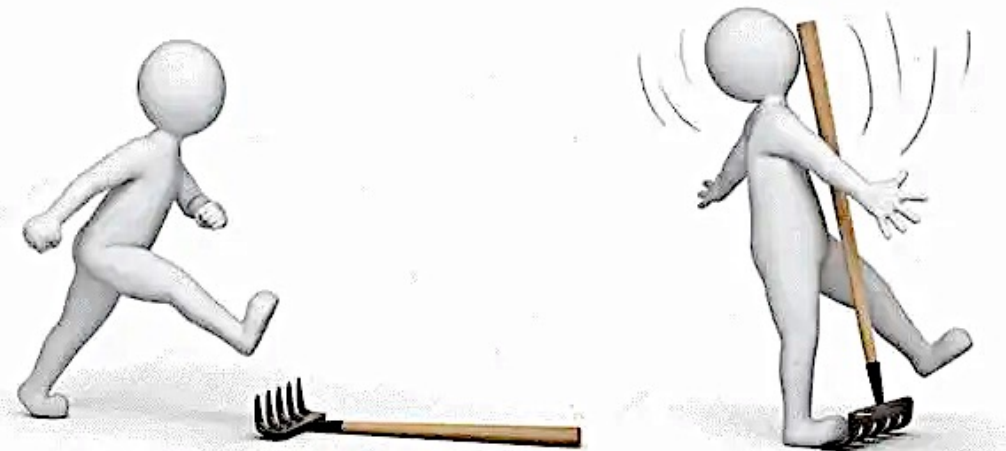
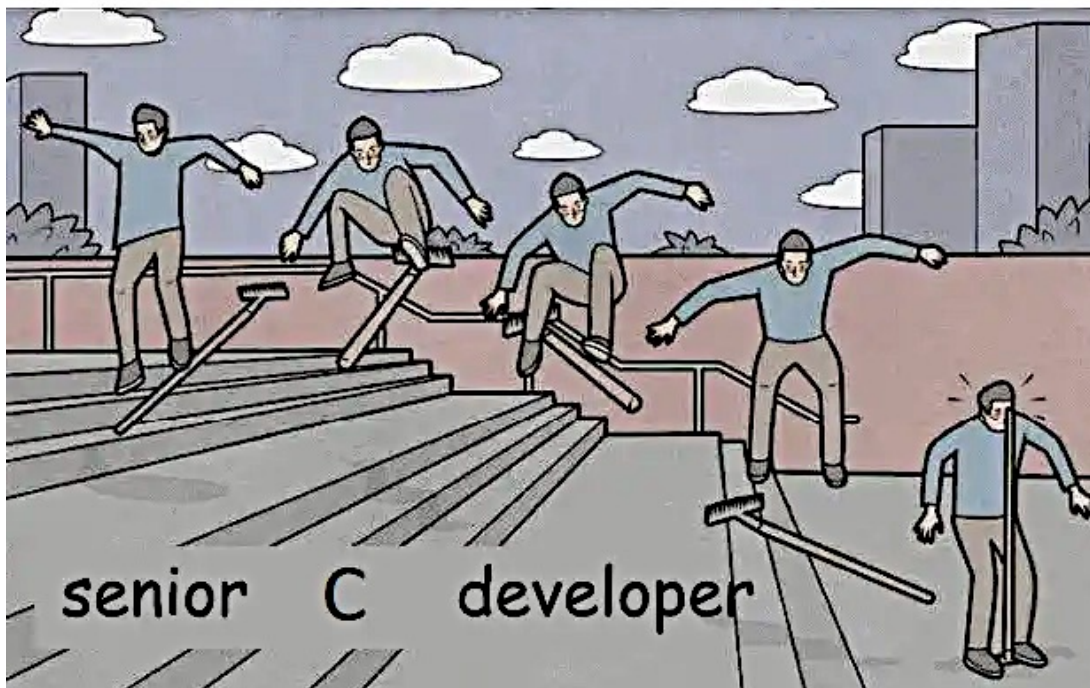
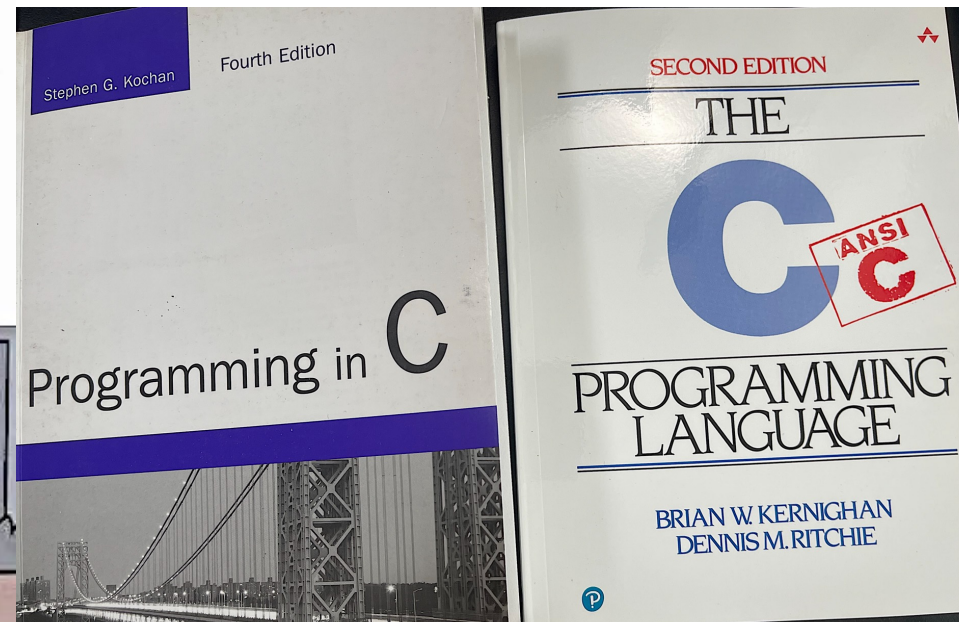


Lecture 11: C part IV



junior C developer

Suggested Literature



senior C developer

RECAP: 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;
}
```

RECAP: Switch statements (w/ **enum**)

```
#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)

RECAP: **struct** in C

```
#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));
}
```

struct allows C to do
Object Oriented Program
(OOP)

C++ will take **struct** to
another level with
Classes

```
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
```

RECAP: **struct** in C

```
#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]);
}
```

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

PS: why not

birthday[0].name = "Viv"?

array type is not assignable

```
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
```

RECAP: 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

RECAP: 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 in
memory, not the
value

```
#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);
```

```
    Date* ptr = &test;  
    print_date(*ptr);
```

```
}
```

```
whovian@phy504:~/host/C$ gcc pointers5.c -o pointers5
```

```
whovian@phy504:~/host/C$ ./pointers5
```

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

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

Pointers

The same logic
applies to **structs**

Pointer point to the
memory location
of the **struct**

This is important
for optimization.

Copying **structs**
can be expensive

Pointers

```
#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);
}
```

whovian@phy504:~/host/C\$ gcc pointers6.c -o pointers6
whovian@phy504:~/host/C\$./pointers6
Name: Vivian Miranda, Birthday: 02-28-1986

Copy the address by value (not the contents of the struct)

Use the symbol ->

This is important for optimization

Copying structs can be expensive

Pointers

```
#include <stdio.h>

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

void test1(Date x)
{
    x.year = 2000;
}

void test2(Date* x)
{
    x->year = 2000;
}

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};
    test1(test);
    print_date(&test);
    test2(&test);
    print_date(&test);
}
```

whovian@phy504:~/host/C\$ gcc pointers7.c -o pointers7
whovian@phy504:~/host/C\$./pointers7
Name: Vivian Miranda, Birthday: 02-28-1986
Name: Vivian Miranda, Birthday: 02-28-2000

 Copied the address

Passing the address
by value allow
functions to modify
the content of its
arguments

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

```
void test(int* const ptr)
{
    *ptr = 1;
    // Assigning ptr to point to another
    // address is not allowed. Example
    // ptr = NULL; // not allowed!!!
}
```

```
int main ()
{
    int x = 4;
    int* const y = &x;
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);

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

Pointers

Passing the address by value allow functions to modify the content of its arguments

This can be good or bad

How do we prevent this?

const correctness

```
whovian@phy504:~/host/C$ gcc pointers8.c -o pointers8
whovian@phy504:~/host/C$ ./pointers8
x = 4
&x = 0x400180539c
y = 0x400180539c
x = 1
&x = 0x400180539c
y = 0x400180539c
```

Address unchanged

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

```
const int z = 10;
```

```
void test(const int* ptr)
```

```
{
```

```
    ptr = &z;
```

```
    // Assigning (*ptr) to another
```

```
    // value is not allowed. Example
```

```
    // *ptr = 1; // not allowed!!!
```

```
    // However since ptr was passed
```

```
    // by value, the change will not
```

```
    // propagate outside the function
```

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

```
}
```

```
int main ()
```

```
{
```

```
    int x = 4;
```

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

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

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

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

```
    test(y);
```

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

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

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

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

```
}
```

Pointers

```
whovian@phy504:~/host/C$ gcc pointers9.c -o pointers9
```

```
whovian@phy504:~/host/C$ ./pointers9
```

```
x = 4
```

```
&x = 0x400180539c
```

```
y = 0x400180539c
```

```
ptr = 0x4000002004
```

```
x = 4
```

```
&x = 0x400180539c
```

```
y = 0x400180539c
```

```
&z = 0x4000002004
```

Passing the address by value
allow functions to modify the
content of its arguments (but
not the pointer itself)

What if we need to modify the
pointer itself? We need pointer
of pointers!

Pointers – Common Mistakes Part I

```
#include <stdio.h>

int main()
{
    int* p1, p2;
    int n = 30;

    p1 = &n;
    p2 = &n; // error
}
```

The problem here is that **p2** is an **integer**!
How to fix that?

int *p1, *p2;

```
whovian@phy504:~/host/Lecture11$ gcc p1.c -o p1
p1.c: In function 'main':
p1.c:9:8: error: assignment to 'int' from 'int *' makes integer from pointer without a cast [-Wint-conversion]
   9 |     p2 = &n; // error
     |         ^
```

Pointers – Common Mistakes Part I

```
#include <stdio.h>

int main()
{
    typedef int* ptr;

    ptr p1, p2;
    int n = 30;

    p1 = &n;
    p2 = &n; // perfection!
}
```

An even better solution
use **typedef**
So elegant.

```
whovian@phy504:~/host/Lecture11$ gcc p2.c -o p2
whovian@phy504:~/host/Lecture11$ █ No errors!
```

Pointers – Common Mistakes Part II

```
#include <stdio.h>

int main()
{
    int* p1;
    int n = *p1;
    printf("%d\n", n);
    return 0;
}
```

Unutilized pointer =
memory/access violation.

Compiler only catches it
when the .c is compiled with
additional flags (use them!)

```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p3.c -o p3
p3.c: In function 'main':
p3.c:6:9: warning: 'p1' is used uninitialized [-Wuninitialized]
     6 |     int n = *p1;           // Error on debug builds
       |           ^
p3.c:5:10: note: 'p1' was declared here
     5 |     int* p1;               // p1 can point to any location in memory
       |           ^~
whovian@phy504:~/host/Lecture11$ gcc p3.c -o p3
whovian@phy504:~/host/Lecture11$ ./p3
-1423263951
```

Pointers – Common Mistakes Part II

The problem with memory violation is that the compiler may not catch it, and the program may run!

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

```
int main()
```

```
{
```

```
    int* p1;           // p1 can point to any location in memory
```

```
    *p1 = 2;           // memory violation!
```

```
    int n = *p1;
```

```
    printf("%d\n", n); // access violation on release builds
```

```
    return 0;
```

```
}
```



```
whovian@phy504:~/host/Lecture11$ nano p4.c
whovian@phy504:~/host/Lecture11$ gcc p4.c -o p4
whovian@phy504:~/host/Lecture11$ ./p4
2
```



Pointers – Common Mistakes Part III

The problem with memory violation is that the compiler may not catch it, and the program may run!

```
#include <stdio.h>

int main()
{
    int* p1;           // p1 can point to any location in memory
    for(int i=0; i<100; i++)
    {
        *(p1+i) = i;    // insane memory violation!
    }
    int n = *(p1+99);
    printf("%d\n", n);
    return 0;
}
```



```
whovian@phy504:~/host/Lecture11$ nano p5.c
whovian@phy504:~/host/Lecture11$ gcc p5.c -o p5
whovian@phy504:~/host/Lecture11$ ./p5
```

99

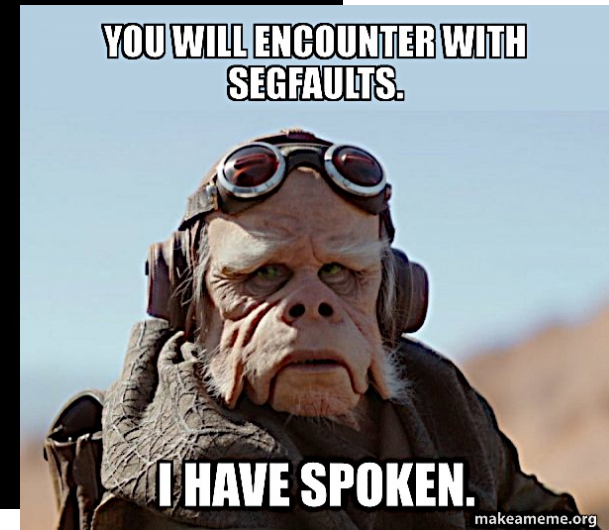


Pointers – Common Mistakes Part III

But at some point, you will run out of luck!

```
#include <stdio.h>

int main()
{
    int* p1;           // p1 can point to any location in memory
    for(int i=0; i<1000; i++)
    {
        *(p1+i) = i;    // insane memory violation!
    }
    int n = *(p1+99);
    printf("%d\n", n);
    return 0;
}
```



```
whovian@phy504:~/host/Lecture11$ gcc p6.c -o p6
```

```
whovian@phy504:~/host/Lecture11$ ./p6
```

```
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
Segmentation fault
```

Pointers – Common Mistakes Part IV

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

```
int main()
{
    int *p1;
    int m;
    p1 = &m; // init pointer with an
             // uninitialized variable
    int n = *p1;
    printf("%d\n", n);
    return 0;
}
```

A pointer to an uninitialized variable will print garbage (can be hard to track in a large code)



```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p7.c -o p7
p7.c: In function 'main':
p7.c:9:9: warning: 'm' is used uninitialized [-Wuninitialized]
   9 |     int n = *p1;
     |         ^
p7.c:6:9: note: 'm' declared here
   6 |     int m;
     |         ^
whovian@phy504:~/host/Lecture11$ ./p7
-1423122528
whovian@phy504:~/host/Lecture11$ gcc p7.c -o p7
whovian@phy504:~/host/Lecture11$ ./p7
-1423122528
```

Pointers – Common Mistakes Part V

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

```
int main()
```

```
{
```

```
    int *p1;
```

```
    int m = 100;
```

```
    p1 = &m;
```

```
    *p1++;    // ERROR
```

```
    printf("%d\n", *p1);
```

```
    printf("%d\n", m);
```

```
    return 0;
```

```
}
```

The order of ***p1++** is opposite of intended and result in memory access violation

- First deference operator “*”
- Second, add one to the pointer address (remember: location is a hex number)

```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p8.c -o p8
```

```
p8.c: In function ‘main’:
```

```
p8.c:8:3: warning: value computed is not used [-Wunused-value]
```

```
8 |     *p1++; // ERROR:
```

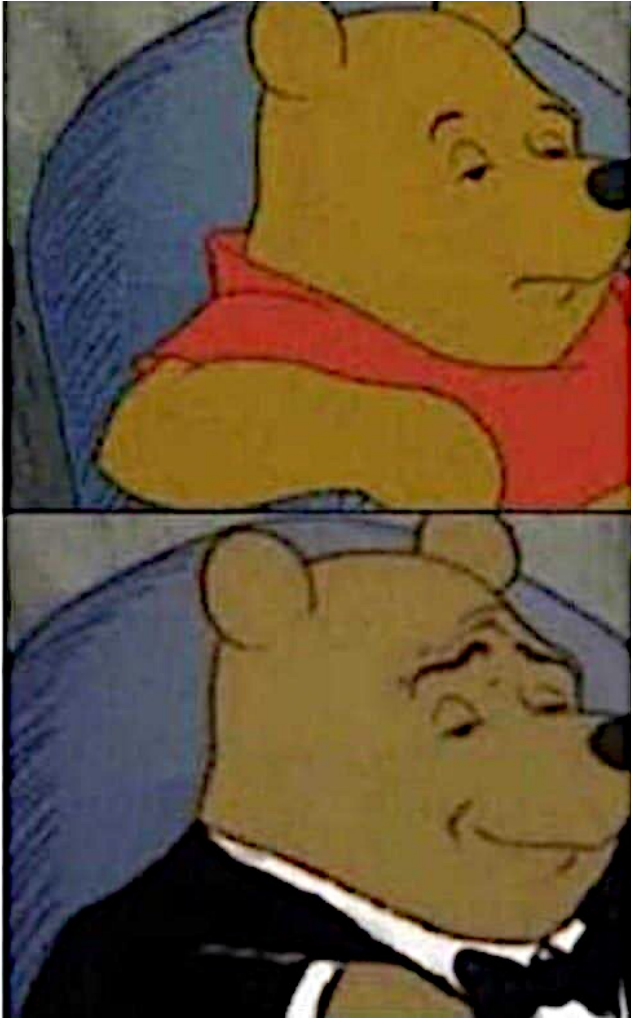
```
  |     ^~~~
```

```
whovian@phy504:~/host/Lecture11$ ./p8
```

```
-1423264768
```

```
100
```

Dynamically allocated memory



```
Int a[10];
```

```
Int* a = (int*)malloc(10*sizeof(int));
```

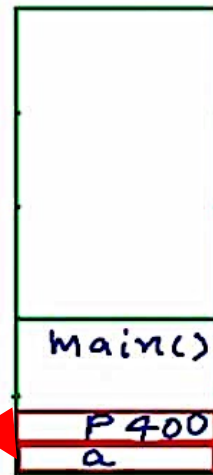
Why do we need malloc?

Malloc allows programs to access a large pool of memory: **heap**

Without **malloc**, we only have access to stack automatic memory (fast but small)

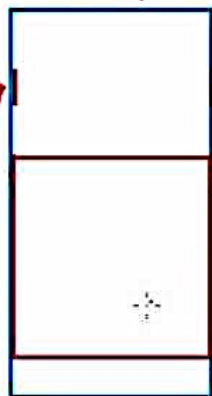
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a; // goes on stack
    int *p;
    p = (int*)malloc(sizeof(int));
    *p = 10;
    free(p);
    p = (int*)malloc(20*sizeof(int));
}
```

Stack

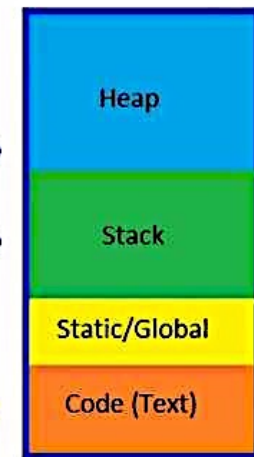


Global

Heap



Application's memory



Free Store

How to generate a stack overflow? Part I

This function calls itself infinite times.

```
int foo()
{
    return foo();
}

int main()
{
    foo();
}
```

Generates a stack overflow when compiled with **-O1**.

Excessive deep or **infinite recursion** generates a stack overflow.

However, **-O2** does **NOT** create a stack overflow (code just runs forever). Why?

-O2 enables **-foptimize-sibling-calls** that converts recursive calls into loops

```
whovian@phy504:~/host/Lecture11$ gcc -O1 so.c -o so
whovian@phy504:~/host/Lecture11$ ./so
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
Segmentation fault
whovian@phy504:~/host/Lecture11$ gcc -O1 -foptimize-sibling-calls so.c -o so
whovian@phy504:~/host/Lecture11$ ./so
```

How to generate a stack overflow? Part II

```
#include <stdio.h>

long int mypow(long int base,
               long int exp) {
    if (exp > 0)
        return base*
            mypow(base, exp - 1);
    else
        return 1;
}

int main()
{
    printf("%ld\n",
           mypow(1, 120000000));
}
```

Generates a stack overflow when compiled with **-O1**.

Excessive deep or infinite recursion generates a stack overflow.

However, **-O2** does **NOT** create a stack overflow. Why?

-foptimize-sibling-calls

```
whovian@phy504:~/host/Lecture11$ gcc -O1 so2.c -o so2
whovian@phy504:~/host/Lecture11$ ./so2
gemu: uncaught target signal 11 (Segmentation fault) - core dumped
Segmentation fault
whovian@phy504:~/host/Lecture11$ gcc -O1 -foptimize-sibling-calls so2.c -o so2
whovian@phy504:~/host/Lecture11$ ./so2
1
```

How to generate a stack overflow? Part III

Stack size is ~8 Megabytes of memory. How do I know that?

```
whovian@phy504:~/host/Lecture11$ ulimit -a
real-time non-blocking time (microseconds, -R) unlimited
core file size              (blocks, -c) 0
data seg size               (kbytes, -d) unlimited
scheduling priority         (-e) 0
file size                   (blocks, -f) unlimited
pending signals             (-i) 31323
max locked memory           (kbytes, -l) unlimited
max memory size             (kbytes, -m) unlimited
open files                  (-n) 1048576
pipe size                   (512 bytes, -p) 8
POSIX message queues        (bytes, -q) 819200
real-time priority          (-r) 0
stack size                  (kbytes, -s) 8192
cpu time                    (seconds, -t) unlimited
max user processes          (-u) unlimited
virtual memory              (kbytes, -v) unlimited
file locks                  (-x) unlimited
```



How to generate a stack overflow? Part III

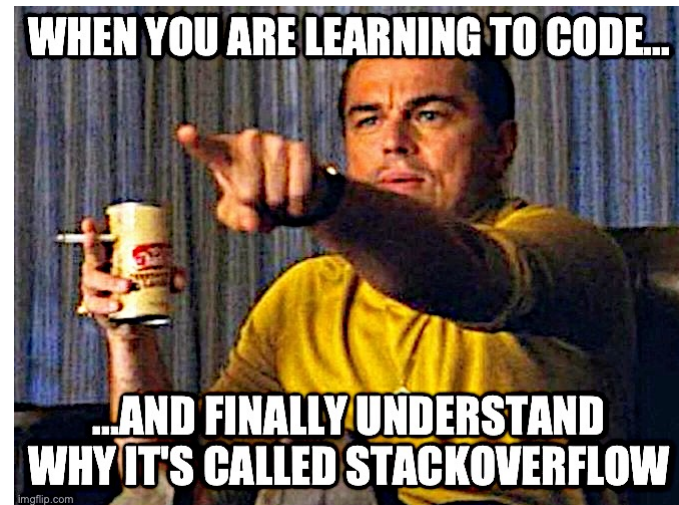
Stack size is ~8 Megabytes of memory.

```
int foo()
{
    double x[1048576];
    return 1;
}

int main()
{
    foo();
    return 0;
}
```

So, let's create an array with more than 8 Megabytes.

Given that double is 8 bytes. We need an array $\sim 10^6$ elements



```
whovian@phy504:~/host/Lecture11$ gcc so3.c -o so3
whovian@phy504:~/host/Lecture11$ ./so3
gemu: uncaught target signal 11 (Segmentation fault) - core dumped
Segmentation fault
```

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

```
int main ()
{
```

```
    const int sz = 10;
    double* x = (double*) malloc(sz*sizeof(double));
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
```

```
    for(int i=0; i<sz; i++)
    {
        x[i] = i;
    }
```

```
    for(int i=0; i<sz; i++)
    {
        printf("x[%d] = %0.2f\n", i, x[i]);
    }
```

```
    // dynamically allocated memory is persistent
    // it only deallocates when either the program
    // ends or when you free the pointers
    free(x);
```

```
    return 0;
```

```
}
```

always check if
the pointer is
NULL

Dynamically allocated memory

malloc provides
memory but does
not initialize
array values

can't forget to
free the memory

```
whovian@phy504:~/host/Phy504Docker/C$ ./pointers12
x[0] = 0.00
x[1] = 1.00
x[2] = 2.00
x[3] = 3.00
x[4] = 4.00
x[5] = 5.00
x[6] = 6.00
x[7] = 7.00
x[8] = 8.00
x[9] = 9.00
```

Dynamically allocated memory

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

int main ()
{
    const int sz = 10;
    double* x = (double*) calloc(sz, sizeof(double));
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for(int i=0; i<sz; i++)
    {
        // ok because calloc init all vars to zero
        printf("x[%d] = %0.2f\n", i, x[i]);
    }
    // dynamically allocated memory is persistent
    // it only deallocates when either the program
    // ends or when you free the pointers
    free(x);

    return 0;
}
```



Both **malloc** and **calloc** return **void***. So casting is needed.

malloc provides memory but does not initialize array values

calloc does init the array to zero

malloc and **calloc** have different arguments (1 x 2)

Don't forget to free the memory

Memory leak with **malloc**

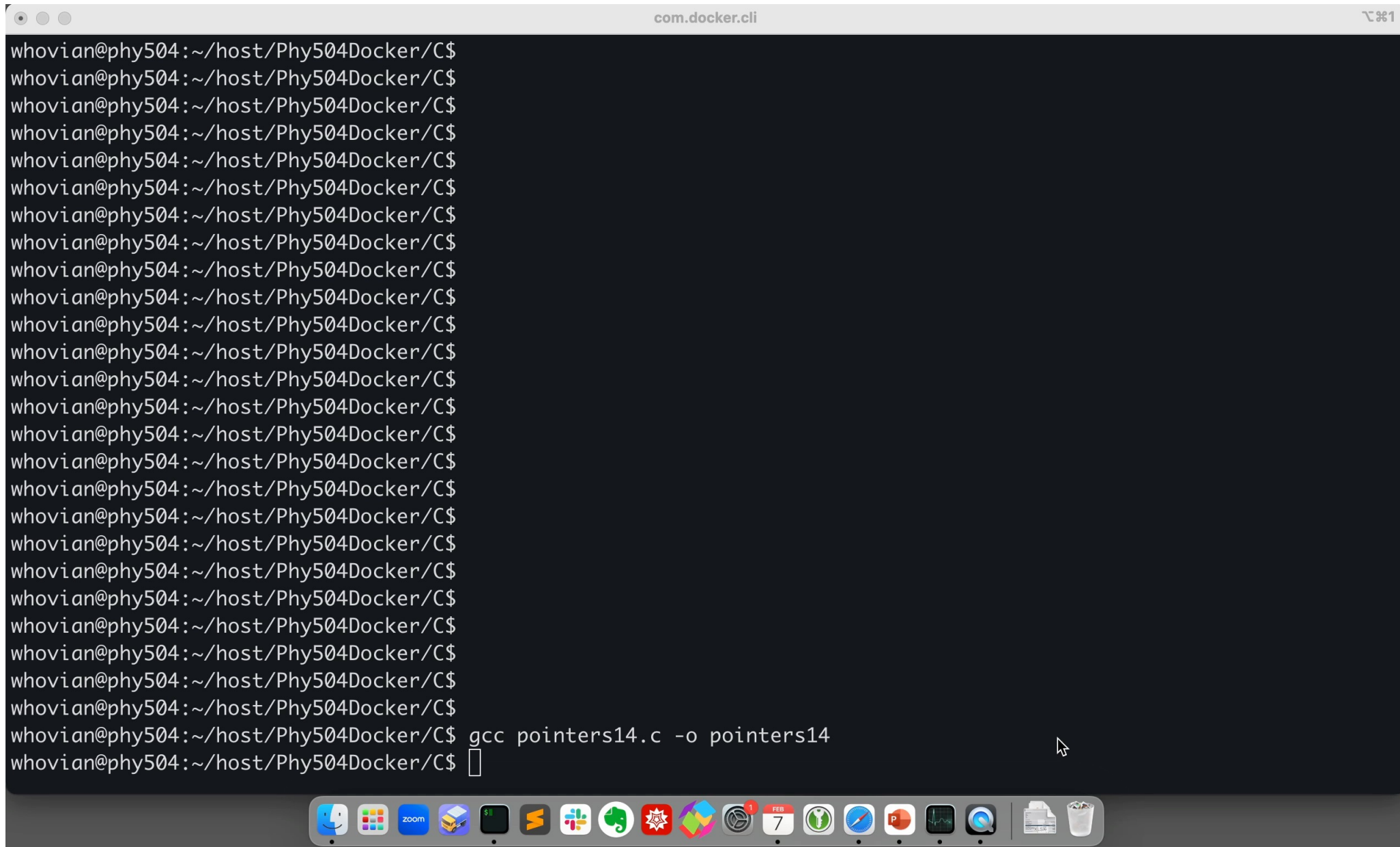
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main ()
{
    const int sz = 200000000;
    // DANGEROUS CODE - THIS WILL CRASH YOUR PC!
    for(int i=0; i<50; i++)
    {
        double* x = (double*) malloc(sz*sizeof(double));
        if (x == NULL)
        {
            printf("Error in memory: memory exhaustion \n");
            exit(1);
        }
        x[0] = 10;
        sleep(1);
        x = NULL; // reference to the memory lost but
                  // the memory continues to be allocated
    }
    return 0;
}
```

Memory leak



Memory leak with **malloc**



The screenshot shows a macOS desktop with a terminal window titled "com.docker.cli". The terminal displays a series of 20 identical prompts: "whovian@phy504:~/host/Phy504Docker/C\$". The 20th prompt is followed by the command "gcc pointers14.c -o pointers14", and the 21st prompt is followed by a cursor. The macOS dock is visible at the bottom, containing icons for various applications including Safari, Photos, Zoom, and a calendar showing "FEB 7".

```
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$
whovian@phy504:~/host/Phy504Docker/C$ gcc pointers14.c -o pointers14
whovian@phy504:~/host/Phy504Docker/C$
```

Pointers – Common Mistakes Part VI

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

```
int main()
{
    int* p1;
    int m = 100;
    p1 = &m;
    free(p1); // dont free stack!
    return 0;
}
```

You are not allowed to free variables that are defined on the stack memory!

```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p9.c -o p9
p9.c: In function 'main':
p9.c:9:3: warning: 'free' called on unallocated object 'm' [-Wfree-nonheap-object]
   9 |     free(p1);
     |     ^~~~~~
p9.c:7:7: note: declared here
   7 |     int m = 100;
     |     ^
whovian@phy504:~/host/Lecture11$ ./p9
free(): invalid pointer
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted
```

Pointers – Common Mistakes Part VII

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

int main()
{
    int* p1 =
        (int*) malloc(sizeof(int));
    *p1 = 99;

    free(p1);
    *p1 = 100; // BAD
    printf("%d\n", (*p1));
    return 0;
}
```

You can't access a stored value once you have freed a pointer!
That is memory corruption.



```
whovian@phy504:~/host/Lecture11$ gcc p10.c -o p10
whovian@phy504:~/host/Lecture11$ ./p10
100
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p10.c -o p10
p10.c: In function 'main':
p10.c:12:3: warning: pointer 'p1' used after 'free' [-Wuse-after-free]
   12 |     printf("%d\n", (*p1));
      |     ~~~~~^~~~~~
p10.c:10:3: note: call to 'free' here
   10 |     free(p1);
      |     ~~~~~
p10.c:11:7: warning: pointer 'p1' used after 'free' [-Wuse-after-free]
   11 |     *p1 = 100; // BAD
      |     ~~~~^~~~~
p10.c:10:3: note: call to 'free' here
   10 |     free(p1);
      |     ~~~~~
```

Pointers – Common Mistakes Part VIII

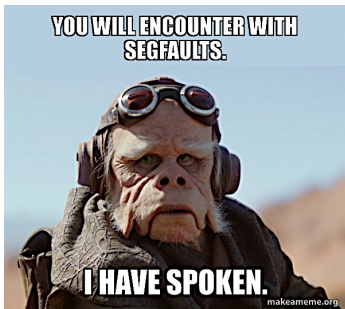
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* p = (int*) malloc(10*sizeof(int));
    p[0] = 1;
    p[1] = 2;
    free(p); // first free
    free(p); // double free
    return 0;
}
```

You can't deallocate
2x memory allocated
with a single **malloc**!

That is memory
corruption.

```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p11.c -o p11
p11.c: In function 'main':
p11.c:12:3: warning: pointer 'p' used after 'free' [-Wuse-after-free]
    12 |     free(p); // double free
        |     ^~~~~~
p11.c:11:3: note: call to 'free' here
    11 |     free(p); // first free
        |     ^~~~~~
whovian@phy504:~/host/Lecture11$ ./p11
free(): double free detected in tcache 2
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted
```



Pointers – Common Mistakes Part IX

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

int main()
{
    const char arr[] = "hello";
    const char *cp = arr;

    printf("Size of arr %d\n", (int) sizeof(arr));
    printf("Size of *cp %d\n", (int) sizeof(*cp));

    return 0;
}
```

Using pointer
and use
sizeof(type) to
determine the
size of an array

```
whovian@phy504:~/host/Lecture11$ nano p12.c
whovian@phy504:~/host/Lecture11$ gcc p12.c -o p12
whovian@phy504:~/host/Lecture11$ ./p12
Size of arr 6
Size of *cp 1
```

Homework I

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

int* F()
{
    int i = 42;
    return &i;
}

int* G()
{
    int j=25;
    return &j;
}

int main()
{
    int* f = F();
    int* g = G();
    printf("f = %i\n", *f);
    return 0;
}
```

Is this a healthy bug-free code?

If not, explain why and the output from **printf**.

Homework II

- Multiple Enumerated Types. Write a program or programs to answer each of the following questions:
 1. What happens if you try to reuse the name of a value within a single enumerated type?
 2. What happens if you try to reuse the name of a value across enumerated types?
 3. What does the compiler report if you try to assign a value from one enumerated type to a variable of another enumerated type?
 4. What happens if you try to give two values the same name in a single enumerated type?
- What do you expect the value of a to be after the following series of commands?

`b = 5; a = ++b - ++b;`

- What do you expect the value of a to be after the following series of commands?

`b = 5; a = b++ - b++;`

Homework III

For questions 7 through 9, assume that P and Q are pointers of the same type, and that each has been assigned a value.

7. What comparison would reliably decide if P and Q have targets with the same value?

- | | | |
|----------------------------------|-----------------|------------------|
| 1) <code>P == Q</code> | 4) All of them | 7) 2 and 3 only |
| 2) <code>*P == *Q</code> | 5) 1 and 2 only | 8) None of these |
| 3) <code>&P == &Q</code> | 6) 1 and 3 only | |

8. What comparison would reliably decide if P and Q have the same target?

- | | | |
|----------------------------------|-----------------|------------------|
| 1) <code>P == Q</code> | 4) All of them | 7) 2 and 3 only |
| 2) <code>*P == *Q</code> | 5) 1 and 2 only | 8) None of these |
| 3) <code>&P == &Q</code> | 6) 1 and 3 only | |

9. What comparison would reliably decide if P and Q have the same value?

- | | | |
|----------------------------------|-----------------|------------------|
| 1) <code>P == Q</code> | 4) All of them | 7) 2 and 3 only |
| 2) <code>*P == *Q</code> | 5) 1 and 2 only | 8) None of these |
| 3) <code>&P == &Q</code> | 6) 1 and 3 only | |
-

Homework IV

For the next four questions, consider implementing the function to calculate the average temperature. The for loop is required to be implemented using pointers to access elements rather than direct array indexing.

```
double Average(const int* const Data, int Sz) {           // Line 1

    if ( Data == NULL || Sz <= 0 ) return 0.0;           //      2

    int Sum = 0;                                         //      3
    const int *Current = Data;                          //      4
    for (int Pos = 0; Pos < Sz; Pos++) {                 //      5
        Sum = Sum + _____;                         //      6
        _____;                                     //      7
    }
    return ( double(Sum) / Sz );                        //      8
}
```

12. What is the effect of the double use of const in Line 1?

- | | |
|--|------------------|
| 1) The first prevents the pointer Data from being modified. | 5) 1 and 3 only |
| 2) The first prevents the target of the pointer Data from being modified. | 6) 1 and 4 only |
| 3) The second prevents the pointer Data from being modified. | 7) 2 and 3 only |
| 4) The second prevents the target of the pointer Data from being modified. | 8) 2 and 4 only |
| | 9) None of these |

13. Current is declared using const in Line 4. Does that conflict with the statement that must be given in Line 7?

- | | | |
|--------|-------|---------------------------|
| 1) Yes | 2) No | 3) Not enough information |
|--------|-------|---------------------------|

Homework V

For the next four questions, consider implementing the function to calculate the average temperature. The for loop is required to be implemented using pointers to access elements rather than direct array indexing.

```
double Average(const int* const Data, int Sz) {           // Line 1

    if ( Data == NULL || Sz <= 0 ) return 0.0;           //      2

    int Sum = 0;                                         //      3
    const int *Current = Data;                          //      4
    for (int Pos = 0; Pos < Sz; Pos++) {                 //      5
        Sum = Sum + _____;                         //      6
        _____                                     //      7
    }
    return ( double(Sum) / Sz );                        //      8
}
```

14. How should the blank in Line 6 be filled?

- | | |
|-------------|------------------|
| 1) Current | 3) *Current |
| 2) &Current | 4) None of these |

15. How should the blank in Line 7 be filled?

- | | |
|-----------------|-----------------------------|
| 1) Current++; | 4) It should be left blank. |
| 2) (*Current)++ | 5) None of these |
| 3) Pos + 1 | |
-

Homework V

For the next five questions, consider implementing the function to find the maximum temperature. The `for` loop is required to be implemented using pointers to access elements rather than direct array indexing.

```
int maxEntry(const int* const Data, int Sz) {           // Line 1

    if ( Data == NULL || Sz <= 0 ) return INT_MIN;      //      2

    int Count = 0;                                     //      3

    // Set hiSoFar to point to the first array element:
    const int *hiSoFar = _____;                  //      4

    // Set Current to point to the second array element:
    const int *Current = _____;                  //      5

    for ( ; Count < Sz; _____ ) {                 //      6

        if ( _____ )                             //      7

            hiSoFar = Current;                         //      8
    }

    return ( _____ );                             //      9
}
```

Homework V

16. How should the blank in Line 4 be filled?

- | | | |
|----------|----------------|------------------|
| 1) &Data | 4) Data[0] | 7) 3 or 4 only |
| 2) *Data | 5) &Data[0] | 8) None of these |
| 3) Data | 6) 3 or 5 only | |

17. How should the blank in Line 5 be filled?

- | | | |
|--------------|----------------|------------------|
| 1) hiSoFar | 4) Data[1] | 7) 2 or 5 only |
| 2) hiSoFar++ | 5) &Data[1] | 8) None of these |
| 3) Data++ | 6) 2 or 4 only | |

18. How should the blank in Line 6 be filled?

- | | |
|-----------------------|-----------------------------|
| 1) Count++ | 4) It should be left blank. |
| 2) Current++ | 5) None of these |
| 3) Count++, Current++ | |

Homework V

19. How should the blank in Line 7 be filled?

- | | |
|--|--|
| 1) <code>*Current > *hiSoFar</code> | 4) <code>*Current > *hiSoFar</code> |
| 2) <code>Current > hiSoFar</code> | 5) None of these |
| 3) <code>&Current > &hiSoFar</code> | |

20. How should the blank in Line 9 be filled?

- | | |
|------------------------------|-----------------------------|
| 1) <code>hiSoFar</code> | 4) It should be left blank. |
| 2) <code>*hiSoFar</code> | 5) None of these |
| 3) <code>&hiSoFar</code> | |
-

Homework VI

What is the output of the following program and why? Does this program have a memory leak? Why?

```
#include <stdio.h>

int main()
{
    int *ptr1, *ptr2, i = 10, j = 20;
    ptr1 = &i;
    ptr2 = &j;
    ptr2 = ptr1;
    *ptr1 = *ptr1 + *ptr2;
    *ptr2 = 2*(*ptr2);

    printf("Val = %d\n", *ptr1 + *ptr2);
    return 0;
}
```

Homework VII

What is the output of the following program and why? Does this program have a memory leak? Why?

```
#include <stdio.h>

int main()
{
    int *ptr1, i = 10;
    double *ptr2, j = 1.234;
    ptr1 = &i;
    ptr2 = &j;
    *ptr1 = *ptr2;

    printf("%d %d %d\n", i, sizeof(ptr1), sizeof(ptr2));
    return 0;
}
```

Homework VIII

What is the output of the following program and why? Does this program have a memory leak? Why?

```
#include <stdio.h>
int main()
{
    int a = 0, b = 1, c = 2, d = 3, arr[3];
    int *ptr[4] = {&a, &b, &c, &d};

    for(int m = 0; m < 3; m++)
    {
        arr[*ptr[m]] = *ptr[m+1];
    }
    for(int m = 0; m < 3; m++)
    {
        printf("%d\n", arr[m]);
    }
    return 0;
}
```

Additional Reading

- **Difference between Stack and Heap memory**
- Video on Stack vs Heap:
<https://www.youtube.com/watch?v=8-ht2AKyH4>
- StackOverflow discussion on contiguous vs non-contiguous memory: **<https://stackoverflow.com/a/26999092/2472169>**
- Top 20 pointer mistakes (I used this post to create this lecture):
<https://www.acodersjourney.com/top-20-c-pointer-mistakes/>