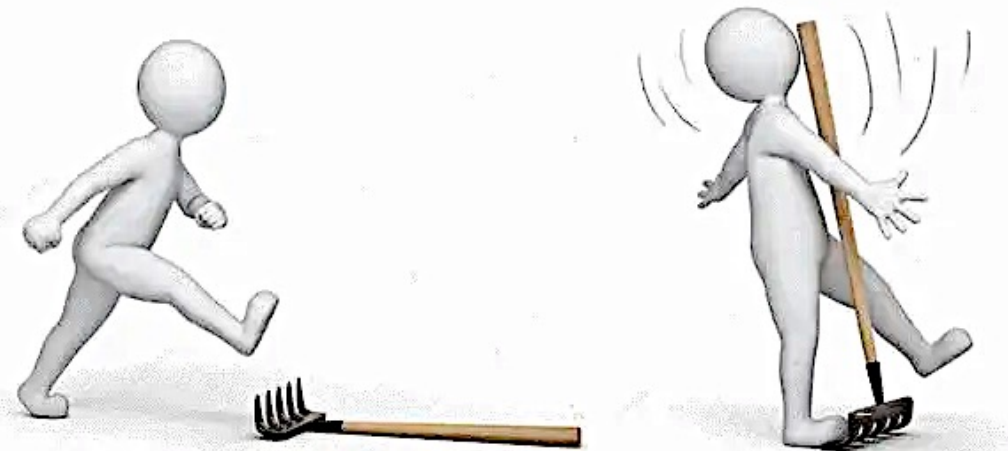
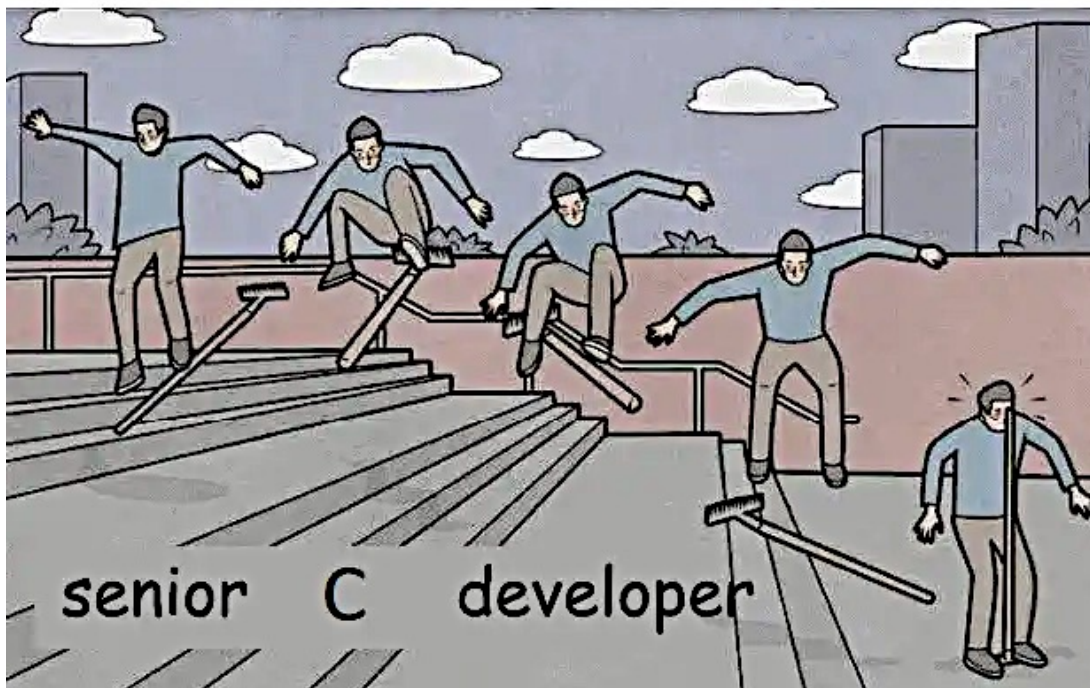
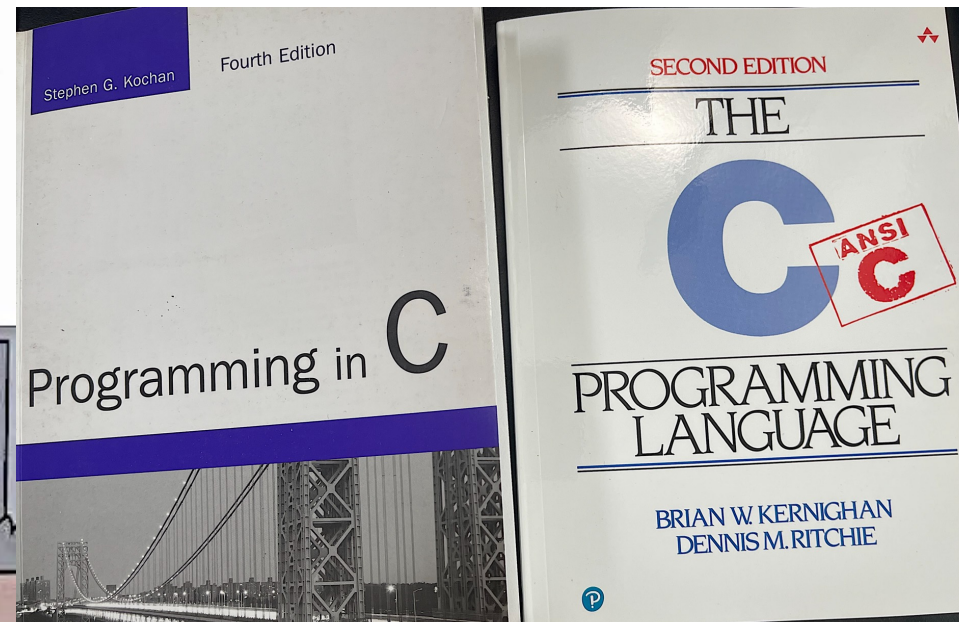


Lecture 12: C part V



junior C developer

Suggested Literature



senior C developer

RECAP: Pointers – Common Mistakes

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
```

RECAP: Pointers – Common Mistakes

```
#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
```

RECAP: generate a stack overflow?

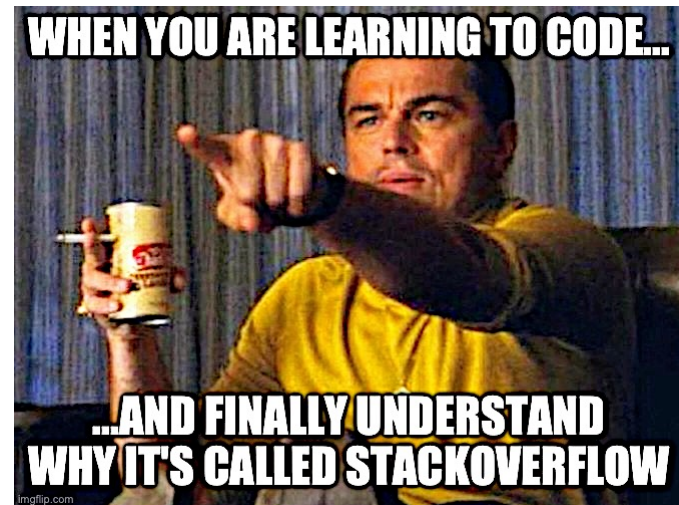
Stack size is ~8 Megabytes of memory.

```
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
```

RECAP: Dynamically allocated memory

```
#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

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
```

RECAP: Pointers – Common Mistakes

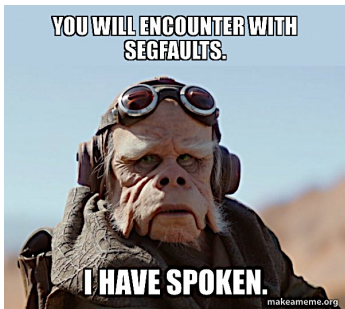
```
#include <stdio.h>
#include <stdlib.h>

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 X

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

int main()
{
    int* p = (int*) malloc(sizeof(int));
    *p = 5;

    p = (int*) malloc(sizeof(int)); // memory leak

    return 0;
}
```

```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p13.c -o p13
whovian@phy504:~/host/Lecture11$ ./p13
whovian@phy504:~/host/Lecture11$
```

Not even **--pedantic -Wall**
warns you

If you lose the
pointers to an
allocated array,
you create
garbage objects
(memory can't
be reached but is
not freed)

Memory leak

Pointers – Common Mistakes Part XI

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

typedef struct {
    char* model; int capacity;
} Aircraft;

int main()
{
    Aircraft af1, af2;

    af1.model =
        (char*) malloc(strlen("Thunderbird") + 1);
    strcpy(af1.model, "Thunderbird");
    af1.capacity = 320;

    af2 = af1; // Shallow copy

    // Modifying af2 will affect af1
    strcpy(af2.model, "BlackHawk");
    printf("%s\n", af1.model);
    return 0;
}
```

If you define a struct
(user-defined type) with
pointers, be careful with
shallow vs deep copy

That is why in C++, we
have **copy constructors**

```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p14.c -o p14
whovian@phy504:~/host/Lecture11$ ./p14
BlackHawk
```

In C, you can create functions
that act more-or-less as copy
constructors (does deep copy)

Pointers – Common Mistakes Part XII

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

int main()
{
    char* str1 =
        (char*) malloc(strlen("Thunderbird") + 1);
    strcpy(str1, "Thunderbird");

    char* str2 = str1;
    printf("%s\n", str1);

    free(str1);
    printf("%s\n", str2); // ERROR
    return 0;
}
```

```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p15.c -o p15
p15.c: In function 'main':
p15.c:16:3: warning: pointer 'str2' used after 'free' [-Wuse-after-free]
   16 |     printf("%s\n", str2); // ERROR
      |     ~~~~~^~~~~~
p15.c:15:3: note: call to 'free' here
   15 |     free(str1);
      |     ~~~~~^~~~~
whovian@phy504:~/host/Lecture11$ ./p15
Thunderbird
[UUU
```

If you have multiple pointers pointing to the same memory, be careful when you free that memory.

That is one reason
why in C++, we have
unique pointers

Pointers – Common Mistakes Part XIII

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

int main()
{
    const int SIZE = 10;
    double* x =
        (double*) malloc(sizeof(double)*SIZE);

    x[SIZE - 1] = 20.21;
    printf("%lf\n", x[SIZE - 1]);

    x[SIZE] = 25.99; // Error
    printf("%lf\n", x[SIZE]);

    return 0;
}
```

Users must be careful
when accessing arrays.

Array indexes go from
zero to **size - 1**



```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p16.c -o p16
whovian@phy504:~/host/Lecture11$ ./p16
20.210000
25.990000
```

Pointers – Common Mistakes Part XIV

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

int main()
{
    const int SIZE = 10;
    double *p =
        (double*) malloc(sizeof(double)*SIZE);

    for (int i=0; i<SIZE; i++)
        *(p + i) = i;

    double *q = p;

    double lastVal = *(q + SIZE); // Error
    printf("%lf\n", lastVal);

    lastVal = *(q + SIZE - 1);    // Good job!
    printf("%lf\n", lastVal);

    return 0;
}
```

Users must be careful
when accessing arrays.

Array indexes go from
zero to **size - 1**



```
whovian@phy504:~/host/Lecture11$ gcc --pedantic -Wall p17.c -o p17
```

```
whovian@phy504:~/host/Lecture11$ ./p17
```

```
0.000000
```

```
9.000000
```

Pointers – Common Mistakes Part XV

Pointers are not a magic wand that allows you not to respect the memory boundaries of types

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

int main()
{
    int num = 2147483647;
    int *pi = &num;
    short *ps = (short*) pi; // max short 32,767

    printf("pi: %p Val(16): %x Val(10): %d\n", pi, *pi, *pi);

    printf("ps: %p Val(16): %hx Val(10): %hd\n", ps,
        (unsigned short) *ps,
        (unsigned short) *ps);
}
```

```
whovian@phy504:~/host/Lecture11$ gcc p18.c -o p18
whovian@phy504:~/host/Lecture11$ ./p18
pi: 0x2aaaab2ab3f4 Val(16): 7fffffff Val(10): 2147483647
ps: 0x2aaaab2ab3f4 Val(16): ffff Val(10): -1
```

Pointers – Common Mistakes Part XVI

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

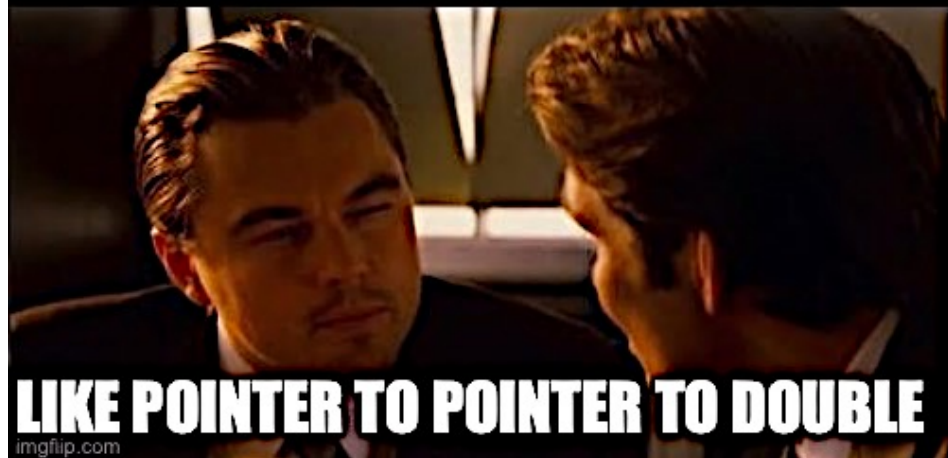
```
int main()
{
    char* str1 =
        (char*) malloc(strlen("Thunderbird")+1);
    strcpy(str1, "Thunderbird");

    char* str2 =
        (char*) malloc(strlen("Thunderbird")+1);
    strcpy(str2, "Thunderbird");

    if (str1 == str2)
        printf("Two strings are equal\n");
    else
        printf("Two strings are NOT equal\n");
}
```

```
whovian@phy504:~/host/Lecture11$ gcc p19.c -o p19
whovian@phy504:~/host/Lecture11$ ./p19
Two strings are NOT equal
```

Pointers
pointing to
different
locations
obviously do
not imply the
contents on
these locations
are different.



Double pointers in
C

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

```
int z = 10;
```

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

```
{
```

```
    *ptr = &z;
```

```
}
```

```
int main ()
```

```
{
```

```
    int x = 4;
```

```
    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);
```

```
}
```

The address of
the pointer to int
is passed by
value

```
whovian@phy504:~/host/Phy504Docker/C$ gcc poi
whovian@phy504:~/host/Phy504Docker/C$ ./point
x = 4
&x = 0x400180537c
 y = 0x400180537c
x = 4
&x = 0x400180537c
 y = 0x4000004010
&z = 0x4000004010
```

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

Double Pointers

```
#include <stdio.h>

int main()
{
    int a = 5;
    int* ptr = &a;
    int** d_ptr = &ptr;

    printf("Size of normal Pointer: %li \n",
        sizeof(ptr));

    printf("Size of Double Pointer: %li \n",
        sizeof(d_ptr));

    return 0;
}
```

int* and **int****
have the same size;
they are both
addresses

```
whovian@phy504:~/host/C$ gcc pointers18.c -o pointers18
whovian@phy504:~/host/C$ ./pointers18 8 bytes = 64 bits =
Size of normal Pointer: 8 up to 16 digits in
Size of Double Pointer: 8 Hex
```

Double Pointers

The trick to reading declarations is to read from right to left. Thus:

```
int const
```

A constant integer Yes! **const int x;** is equivalent to **int const x;**

```
int const *
```

A (variable) pointer to a constant integer

```
int * const
```

A constant pointer to a (variable) integer

```
int * const *
```

A pointer to a constant pointer to an integer

```
int const * *
```

A pointer to a pointer to a constant integer

```
int const * const *
```

A pointer to a constant pointer to a constant integer

You can see that the `const` always refers to the text to its left, just as the `*` does.

Source: 21st Century C,
2nd Edition. Author:
Ben Klemens

Constant correctness is involved

- The double pointer to an **int** is constant, i.e., it can't be assigned to another pointer to int address: **int** const ptr;**
- The pointer to **int** that the double pointer holds (***ptr**) is not a constant pointer to an integer. (***ptr**) can be assigned to another integer address
- What if we don't want the pointer (***ptr**) to **int** that the double pointer holds to be assigned to another address)?

int* const * const

- Even in this case, the **int** that ***(***ptr**)** holds can be modified? What if we don't want that? We must define the double pointer **ptr** as **const int* const * const** or **int const * const * const** (some people find that easier to remember. In this case, **const** is a promise for the operator on the left

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

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

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

```
{
```

```
    *ptr = &z;    // This fails!
```

```
    ptr = NULL;   // This also fails!
```

```
    *(*ptr) = 4;  // This also fails!
```

```
}
```

```
int main ()
```

```
{
```

```
    int x = 4;
```

```
    const 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);
```

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

```
}
```

Pointers

The double pointer
to an **int**, **ptr**, is
constant

The pointer **ptr** can't
be assigned to
another pointer to int
address

```
pointers11.c: In function 'test':
```

```
pointers11.c:7:10: error: assignment of read-only location '*ptr'
```

```
7 |     *ptr = &z;    // This fails!
  |     ^
```

```
pointers11.c:8:9: error: assignment of read-only parameter 'ptr'
```

```
8 |     ptr = NULL;   // This also fails!
  |     ^
```

```
pointers11.c:9:13: error: assignment of read-only location '**ptr'
```

```
9 |     *(*ptr) = 4;  // This also fails!
  |     ^
```

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

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

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

```
{
```

```
    *ptr = &z; // This fails!
```

```
    ptr = NULL; // This also fails!
```

```
    *(*ptr) = 4; // This also fails!
```

```
}
```

```
int main ()
```

```
{
```

```
    int x = 4;
```

```
    const 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);
```

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

```
}
```

Pointers

The pointer to **int** that the double pointer holds, **(*ptr)**, is constant

The pointer **(*ptr)** can't be assigned to another integer address

```
pointers11.c: In function 'test':
```

```
pointers11.c:7:10: error: assignment of read-only location '*ptr'
```

```
7 |     *ptr = &z; // This fails!
```

```
pointers11.c:8:9: error: assignment of read-only parameter 'ptr'
```

```
8 |     ptr = NULL; // This also fails!
```

```
pointers11.c:9:13: error: assignment of read-only location '**ptr'
```

```
9 |     *(*ptr) = 4; // This also fails!
```

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

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

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

```
{
```

```
    *ptr = &z;    // This fails!
```

```
    ptr = NULL;   // This also fails!
```

```
    *(*ptr) = 4;  // This also fails!
```

```
}
```

```
int main ()
```

```
{
```

```
    int x = 4;
```

```
    const 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);
```

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

```
}
```

Pointers

The integer that
****(**ptr*)*** holds can't
be modified

The integer variable
****(**ptr*)*** can't be
assigned to another
integer value

```
pointers11.c: In function 'test':
```

```
pointers11.c:7:10: error: assignment of read-only location '*ptr'
   7 |     *ptr = &z;    // This fails!
     |     ^
```

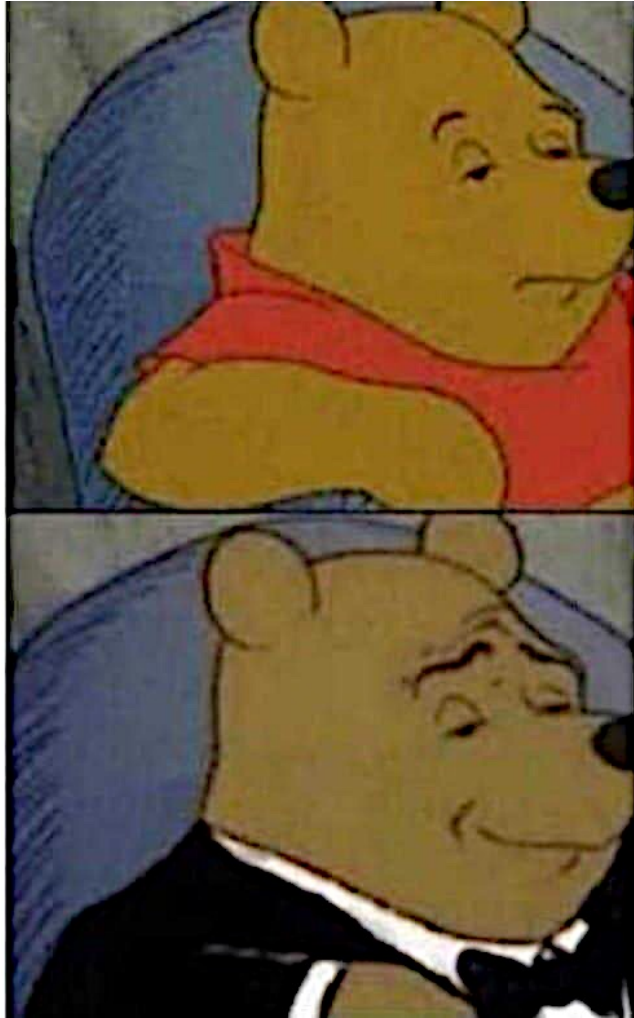
```
pointers11.c:8:9: error: assignment of read-only parameter 'ptr'
   8 |     ptr = NULL;   // This also fails!
     |     ^
```

```
pointers11.c:9:13: error: assignment of read-only location '**ptr'
   9 |     *(*ptr) = 4;  // This also fails!
```

What about triple pointers in C?



Dynamically allocated memory (2D array)



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

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

malloc & calloc on double pointers

In this common way:
we need to **malloc** each
dimension separately
(with the **for loop**)

When deallocating
memory with **free**, we
also need to free each
dimension separately

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

int main ()
{
    const int nrows = 10;
    const int ncols = 10;
    double** x = (double**) malloc(sizeof(double*)*nrows);
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for (int i=0; i<nrows; i++)
    {
        x[i] = (double*) calloc(ncols, sizeof(double));
        if (x[i] == NULL)
        {
            printf("Error in memory: memory exhaustion \n");
            exit(1);
        }
    }
    for (int i=0; i<nrows; i++)
    {
        for(int j=0; j<ncols; j++)
        {
            printf("x[%d,%d] = %.2f", i, j, x[i][j]);
        }
        printf("\n");
    }
    for (int i=0; i<nrows; i++)
    {
        free(x[i]);
    }
    free(x);
    return 0;
}
```

malloc & calloc on double pointers

The problem with that approach is the memory is not contiguous

In normal circumstances, that is totally ok

But there are more optimal ways in High-Performance Computing (contiguous memory)

Solution 1: 1D array such that **$a[i,j] = v[i + \text{nrows} * j]$**

Solution2: (ref: Ex4 - <https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/>)

malloc & calloc on double pointers

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

int main ()
{
    const int nrows = 3;
    const int ncols = 2;
    double* x = (double*) malloc(sizeof(double)*ncols*nrows);
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for (int i=0; i<nrows; i++)
    {
        for(int j=0; j<ncols; j++)
        {
            x[i + nrows*j] = i + j;
            printf("x[%d,%d] = %0.2f  ", i, j, x[i + nrows*j]);
        }
        printf("\n");
    }
    free(x);
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc pointers17.c -o pointers17
whovian@phy504:~/host/C$ nano pointers17.c
whovian@phy504:~/host/C$ ./pointers17
x[0,0] = 0.00  x[0,1] = 1.00
x[1,0] = 1.00  x[1,1] = 2.00
x[2,0] = 2.00  x[2,1] = 3.00
```

Solution 1: 1D
array such that

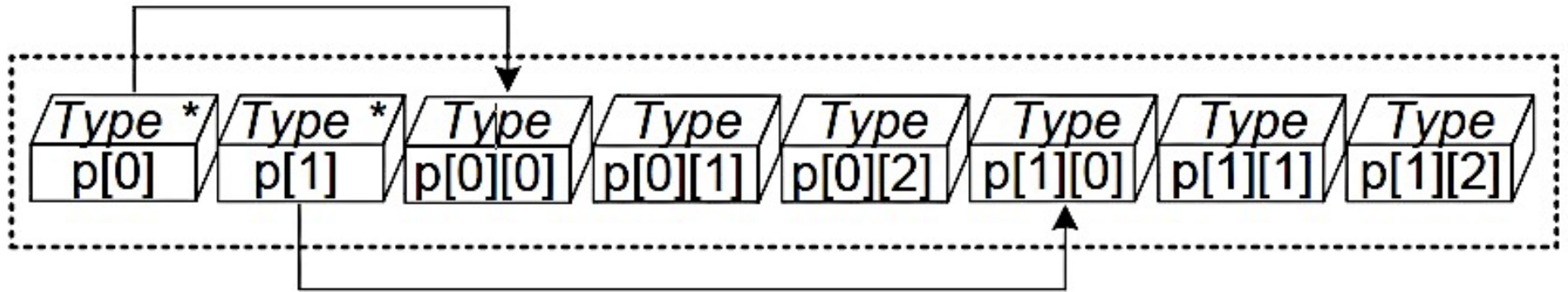
$$a[i,j] = v[i + \text{nrows} * j]$$

Cumbersome!

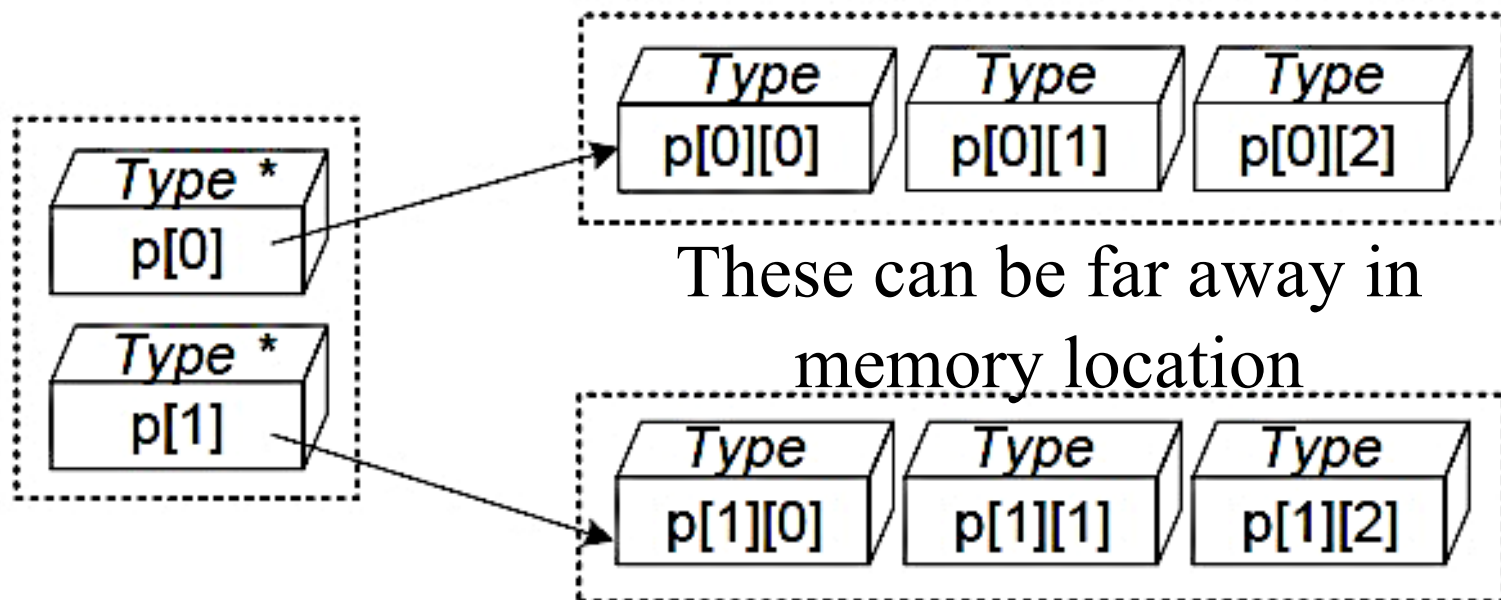
Good part:
memory is
contiguous

malloc & calloc on double pointers

good: Memory is contiguous



bad: Memory is non-contiguous. Integer arithmetic w/ pointers not possible, i.e., $a[i,j] \neq *(a + i + \text{nrows} * j)$



Homework I

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

```
int i, j, *p, *q;  
i = 10;  
p = &j;  
q = malloc(sizeof(int));  
*q = i;  
j = i;  
q = p;  
*q = 5;  
printf("%d %d %d %d", i, j, *p, *q);
```

Homework II

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, *ptr3, i = 10, j = 20, k = 30;
    ptr1 = &i;
    i = 100;
    ptr2 = &j;
    j = *ptr2 + *ptr1;
    ptr3 = &k;
    k = *ptr3 + *ptr2;

    printf("%d %d %d\n", *ptr1, *ptr2, *ptr3);

    return 0;
}
```

Homework III

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, **ptr, i = 10, j = 20;
    ptr1 = &i;
    ptr = &ptr1;
    **ptr += 100;
    ptr1 = &j;
    **ptr += 100;

    printf("Val = %d\n", i+j);

    return 0;
}
```

Homework IV

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

```
#include <stdio.h>

int main()
{
    int *ptr, i = 10, j = 20, k = 30;
    ptr = &i;
    *ptr = 40;
    ptr = &j;
    *ptr += i;
    ptr = &k;
    *ptr += i + j ;

    printf("i = %d j = %d k = %d\n", i, j, k);

    return 0;
}
```

Homework V

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;
    *ptr1 = 150;
    ptr2 = &j;
    *ptr2 = 50;
    ptr2 = ptr1;
    *ptr2 = 250;
    ptr2 = &j;
    *ptr2 += *ptr1;

    printf("Val = %d\n", j);

    return 0;
}
```

Homework VI

```
#include <stdio.h>

void test(int **tmp, int i)
{
    *(*tmp + i) += 50;
}

int main()
{
    int *ptr;
    int **tmp;
    int arr[] = {10, 20, 30};

    ptr = arr;
    tmp = &ptr;

    for(int i=0; i<3; i++)
    {
        test(tmp, i);
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

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

Why?

Homework VII

const correctness is a significant and important deal on pointers

Question:

Examine the following pointer declarations:

c

 Copy

```
const int *ptr1;  
int *const ptr2;  
const int *const ptr3;
```

- a) Explain the differences between these three declarations.
- b) Write a code snippet that demonstrates what operations (changing the pointer, changing the value pointed to) are allowed or disallowed for each.

Homework VIII

- Write a function that multiplies the matrix by a scalar
- Write a vector struct and then a function that evaluate a linear transformation Ax , where A is a matrix and x is a vector. The function should check if the dimensions are compatible
- Write a function that checks if the matrix is either 2×2 or 3×3 and in these two cases, return the determinant. In all other cases, it returns an error

Additional Reading

- GeeksforGeeks article on double pointer:
<https://www.geeksforgeeks.org/c-pointer-to-pointer-double-pointer/>