CSCI3150 – Tutorial 2

Tutorial 2 - C refresher: pointers II + Warmup Discussion

Calvin Kam <hckam@cse.cuhk.edu.hk>

Agenda

1. 2D Array vs Array of Character Pointers?
2. Malloc() and Free()
3. Warmup exercise explanation

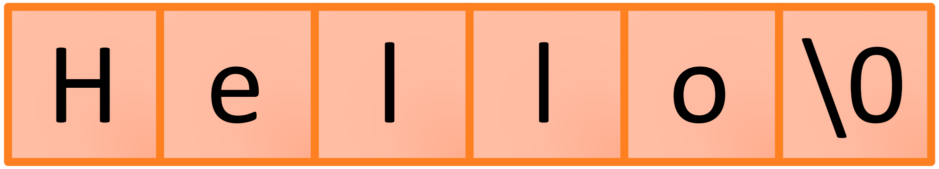
Remember Last Time? Array vs Array of Pointer?

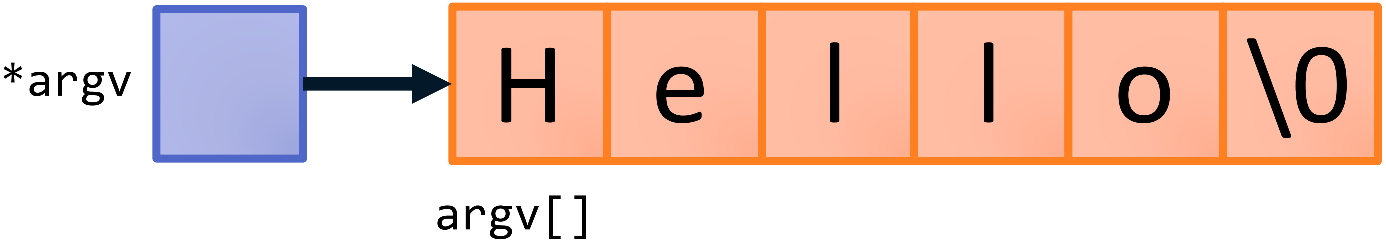
`char argv[]`

- Just a simple character array.

`char *argv[]`

- Array of Character Pointers





Let’s inspect the *argv[]

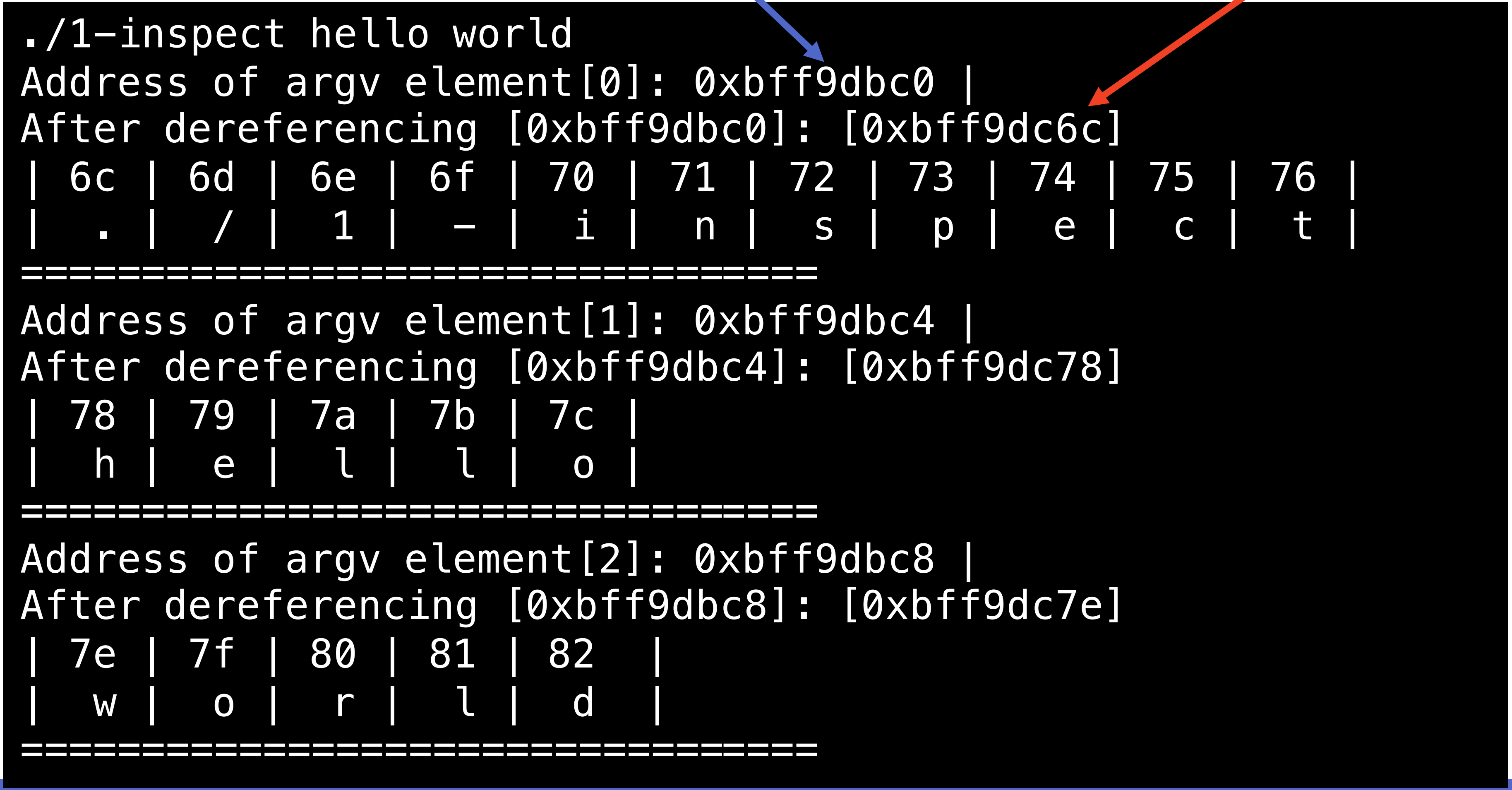
1-inspect.c

```
for(i = 0;i < argc;i++){
    printf("Address of argv element[%d]: %p |\n",i,tmpArgv);
    char *ptr = argv[i];
    printf("After derefercing [%p]: [%p]\n",tmpArgv,ptr);
    for (j = 0;j < strlen(argv[i]);j++)
      printf("|%3s ",printPtrAddr(ptr++));

    printf("|\n");
    ptr = argv[i];
    for (j = 0;j < strlen(argv[i]);j++)
      printf("|%3c ",*(ptr++));

    printf("|\n");
    printf("==================================\n");
    tmpArgv++;
  }
```

Result:

First Derefere cing

Second Derefere cing



```
./1-inspect hello world
Address of argv element[0]: 0xbff9dbc0 |
After dereferencing [0xbff9dbc0]: [0xbff9dc6c]
| 6c | 6d | 6e | 6f | 70 | 71 | 72 | 73 | 74 | 75 | 76 |
|  . |  / |  1 |  - |  i |  n |  s |  p |  e |  c |  t |
=================================
Address of argv element[1]: 0xbff9dbc4 |
After dereferencing [0xbff9dbc4]: [0xbff9dc78]
| 78 | 79 | 7a | 7b | 7c |
|  h |  e |  l |  l |  o |
=================================
Address of argv element[2]: 0xbff9dbc8 |
After dereferencing [0xbff9dbc8]: [0xbff9dc7e]
| 7e | 7f | 80 | 81 | 82  |
|  w |  o |  r |  l |  d  |
=================================
```

2D Array

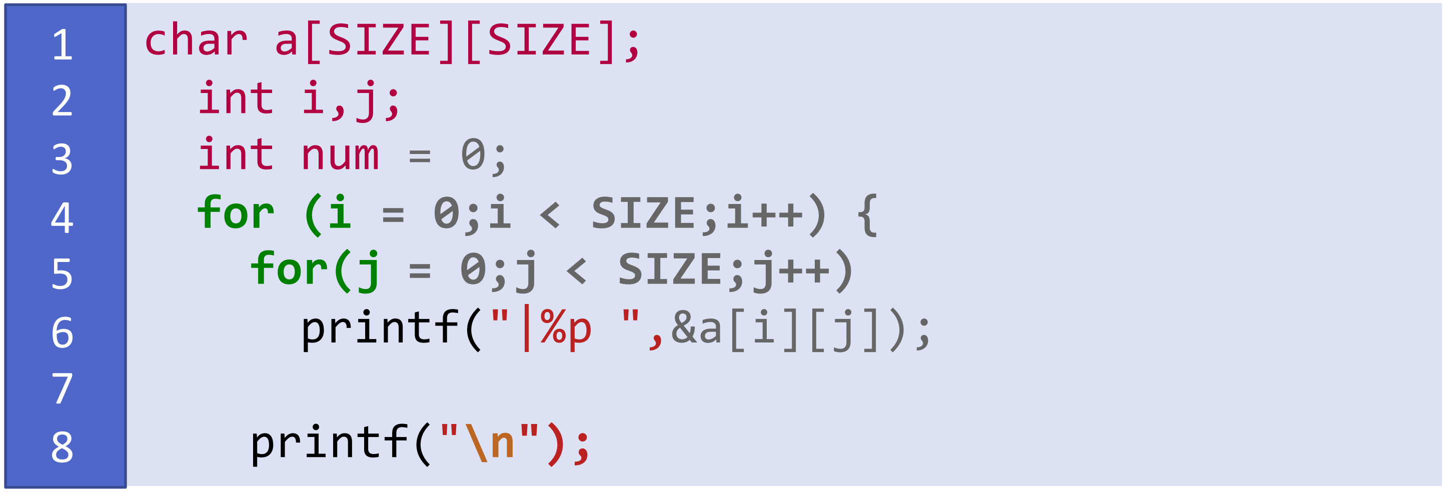
In C, we can declare a two dimension array like this:

```
int mark[10][10];
```

Can we say array of character pointers and 2-D array are the same

- No

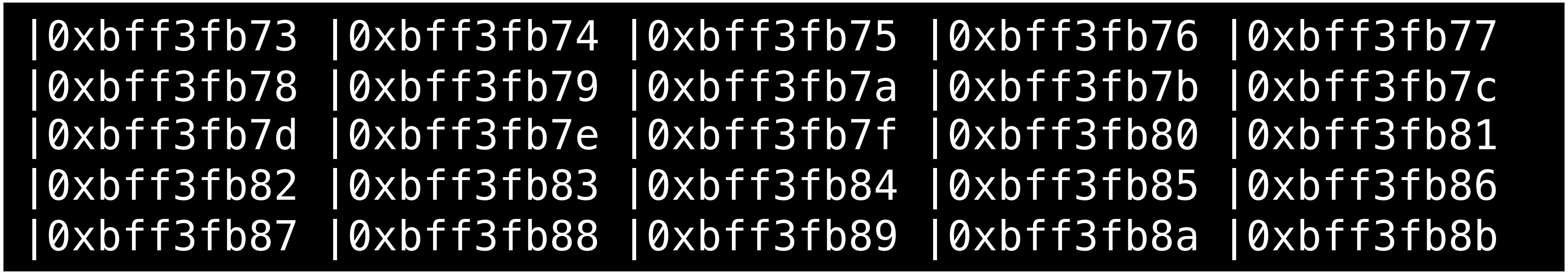
Let’s check



```
char a[SIZE][SIZE];
  int i,j;
  int num = 0;
  for (i = 0;i < SIZE;i++) {
    for(j = 0;j < SIZE;j++)
      printf("|%p ",&a[i][j]);

    printf("\n");
```

2-2DArray.c

Results



```
|0xbff3fb73 |0xbff3fb74 |0xbff3fb75 |0xbff3fb76 |0xbff3fb77
|0xbff3fb78 |0xbff3fb79 |0xbff3fb7a |0xbff3fb7b |0xbff3fb7c
|0xbff3fb7d |0xbff3fb7e |0xbff3fb7f |0xbff3fb80 |0xbff3fb81
|0xbff3fb82 |0xbff3fb83 |0xbff3fb84 |0xbff3fb85 |0xbff3fb86
|0xbff3fb87 |0xbff3fb88 |0xbff3fb89 |0xbff3fb8a |0xbff3fb8b
```

- Actually it consists of 25 consecutive memory spaces.

- `a[i][j] = *(a + i*SIZE + j)`

Malloc() and Free()

Problem 1: Empty Pointers?

What happens if we are going to access/modify the pointer pointing to nothing?

```
        int *jPtr;
        printf("The value of jPtr(%p) is [%d].\n",jPtr,*jPtr);
        printf("-------- We are putting 1234 to *jPtr...--------\n");
        *jPtr = 1234;
        return 0;
}
```

Crash?

Problem 2: Dynamic Allocation?

Can we allocate a dynamically array like this?

```
#include <stdio.h>

int main(int argc,char *argv[]){
        int a = 5;
        int numArray[a];
        return 0;
}
```

🚫

The Compiler may let you go but it is not a Standard C method!

malloc() & free()

How can we allocate an new memory space dynamically?

C has no new , but it has a function malloc()

malloc()asks the OS to allocate n bytes of memory.

Then it returns the pointer (address) of that allocated memory!

malloc() and free()

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

int main(int argc,char *argv[]) {
// Here We declare an empty Int pointer...
        int *nPtr;
// We ask the OS to allocate some memory for us
        nPtr = malloc(sizeof(int));
        if(nPtr == NULL)
                printf("Cannot malloc()!!\n");
        printf("Memory Allocated. nPtr(%p):
[%d]\n",nPtr,*nPtr);
        printf("Success..We are going to put
3150 there...\n\n");
        *nPtr = 3150;
        printf("Now: nPtr(%p):
[%d]\n",nPtr,*nPtr);
........
```

malloc() Requires stdlib.h
(Standard LIBrary)

malloc() will return NULL if
memory cannot be allocated
(eg: Memory FULL)

malloc() and free()

Remember to free it when you are not using it anymore.

For example, deletion in linked list.

free()

```
// Here We declare an empty Int pointer...
        int *nPtr;
// We ask the OS to allocate some memory for us
        nPtr = malloc(sizeof(int));
        if(nPtr == NULL)
                printf("Cannot malloc()!!\n");
        *nPtr = 3150;

// Remember to Free it after use
        free(nPtr);
        printf("\n\nAfter we free it...\n");
        printf("Now: nPtr(%p): [%d]\n",nPtr,*nPtr);
// We Declare a new Pointer
        char *newPtr = malloc(sizeof(char));
        printf("\n\nThe New Pointer located at 
(%p)\n",newPtr);
        *newPtr = 'a';
        printf("After putting sth, Value of 
newPtr:%c\n\n",*newPtr);
        printf("Now: nPtr(%p): [%d]\n",nPtr,*nPtr);
```

Trying to access the memory space already freed...
What will happen?

free()

After freeing the pointers, the memory will be returned to the program for further allocation.

If you continue to access using the old pointer, UNDEFINED ACTION will be occurred.

We call that : **dangling pointer**

Create Array Using malloc

Can we dynamically create an array? YES!

By multiplying the size to the total element, we can create an array dynamically (on-the-fly).

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

int main(int argc,char *argv[])
{
        int *array;
        //Creating a pointer with Size
        array = malloc(sizeof(int)*SIZE);
        int *ptr = array;

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

        printf("Printing the Array....\n");
        for(i=0;i<SIZE;i++){
                printf("Element %d: [%d]\n",i,array[i]);
        }
        return 0;
}
```

Give malloc() the number of elements for the array

Access it in array style! Okay!

Warmup Exercise 1

Question 1- Interchangable printf()?

```
#include <stdio.h>

int * addition(int a, int b) {
        int c = a + b;
        int *d = &c;
        return d;
}

int main(void) {
        int result = *(addition(1, 2));
        int *result_ptr = addition(1, 2);
        printf("result = %d\n", *result_ptr);
        printf("result = %d\n", result);
        return 0;
}
```

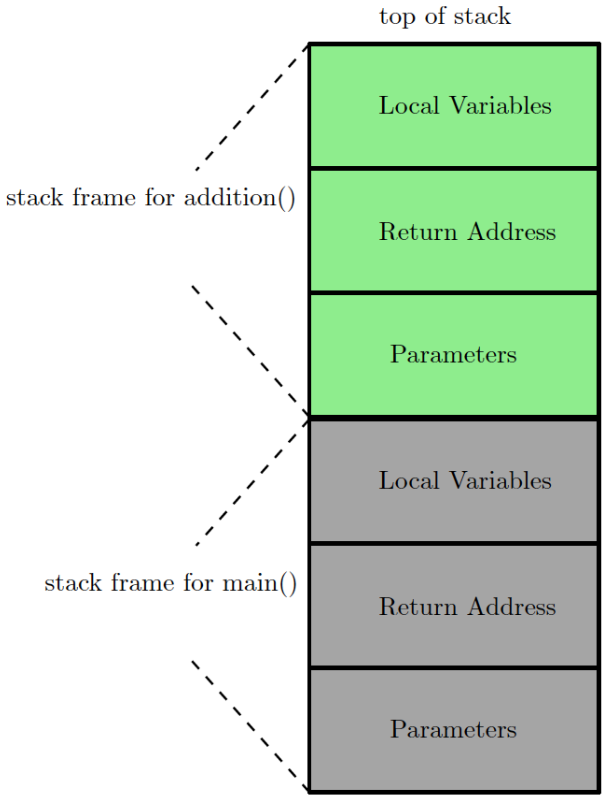
Question 1 - Result

```
./q1
result = 3
result = 3
```

After interchanging the statement:

```
./q1
result = 3
result = 0
```

Call Stack



When the function returns, the related space will be recycled and it can be used by other things.

printf() also occupies memory spaces. After printing once, the function has already occupied the block.

When the 2nd printf() tries to access, it retrieves some weird content.

NEVER RETURNS THE ADDRESS OF LOCAL VARIABLE

If you really want to return a pointer…

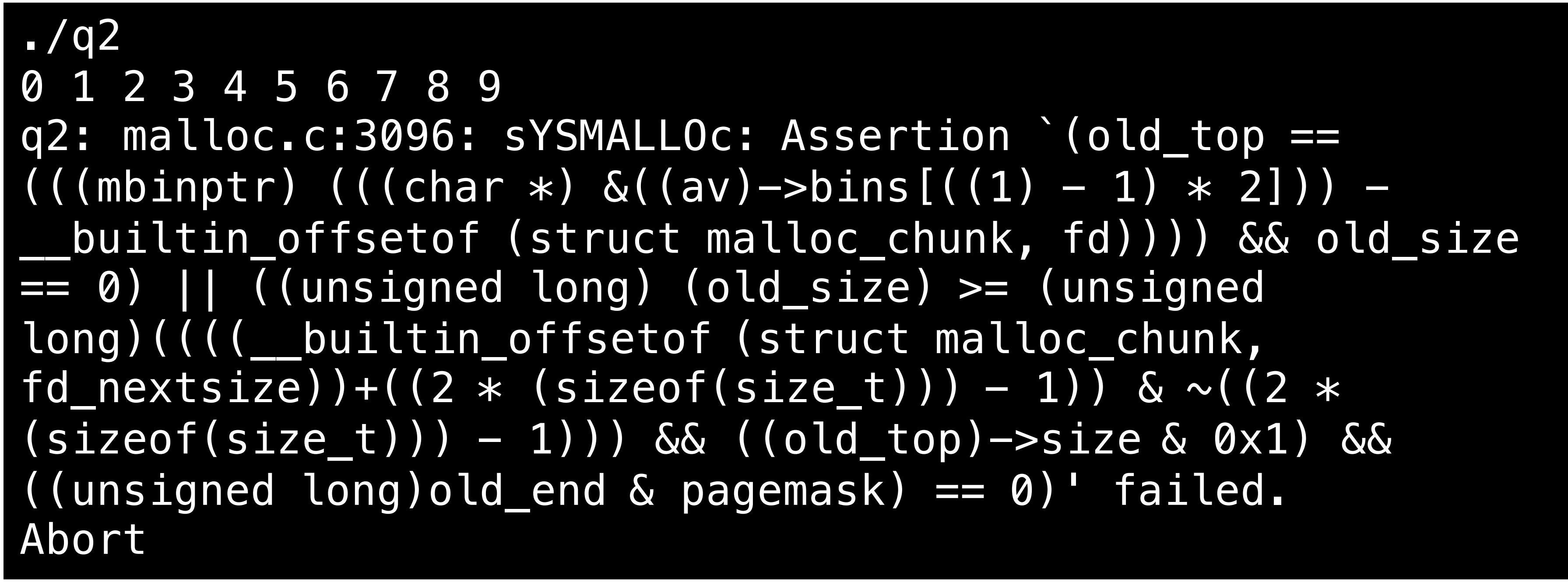
```
int * addition(int a, int b) {
        int c = a + b;
        int *d = malloc(sizeof(int));
        *d = c;
        return d;
}
```

Use malloc() instead

Question 2

```
int count = 0;

int * new_array() {
       int i, *array = (int *) malloc(sizeof(int) * 9);
       for(i = 0; i <= 9; i++)
              array[i] = count++;
       for(i = 0; i <= 9; i++)
              printf("%d ", array[i]);
       printf("\n");
       return array;
}
int main(void) {
       int i;
       int *a;
       for(i = 0; i < 10; i++) {
              a = new_array();
       }
       return 0;
}
```

Question 2 - Result



```
./q2
0 1 2 3 4 5 6 7 8 9
q2: malloc.c:3096: sYSMALLOc: Assertion `(old_top ==
(((mbinptr) (((char *) &((av)->bins[((1) - 1) * 2])) -
__builtin_offsetof (struct malloc_chunk, fd)))) && old_size
== 0) || ((unsigned long) (old_size) >= (unsigned
long)((((__builtin_offsetof (struct malloc_chunk,
fd_nextsize))+((2 * (sizeof(size_t))) - 1)) & ~((2 *
(sizeof(size_t))) - 1))) && ((old_top)->size & 0x1) &&
((unsigned long)old_end & pagemask) == 0)' failed.
Abort
```

Segmentation fault ☹

Undefined Action

In the program, we are trying to access the space beyond an array.

→Undefined action will occur in this case.

EXTRA: Why in 64-bit this program runs without a problem?

→Related to the behavior of malloc().

→ malloc() always allocates more spaces than you requested, for storing metadata.

→ Larger in 64-bit, and you are luckily to store the "extra" one int in those area.

Question 3 -

```
void process_array(int array[ROWS][COLS]) {
         int i, j, count = 0;
         for(i = 0; i < ROWS; i++)
                  for(j = 0; j < COLS; j++)
                           array[i][j] = count++;
}
int main(void) {
         int **array = malloc(sizeof(int) * ROWS * COLS);
         process_array(array);

         int i, j;
         for(i = 0; i < ROWS; i++) {
                  for(j = 0; j < COLS; j++) {
                           printf("%d ", array[i][j]);
                  }
                  printf("\n");
         }

         return 0;
}
```

Question 3 - Result

```
./q3
Segmentation fault
```

Segmentation Fault Again ☹

Wrong Types!

2-D Array!

```
void process_array(int array[ROWS][COLS]) {
        int i, j, count = 0;
        for(i = 0; i < ROWS; i++)
                for(j = 0; j < COLS; j++)
                        array[i][j] = count++;
}
int main(void) {
        int **array = malloc(sizeof(int) * ROWS * COLS);
        process_array(array);

        int i, j;
        for(i = 0; i < ROWS; i++) {
                for(j = 0; j < COLS; j++) {
                        printf("%d ", array[i][j]);
                }
                printf("\n");
        }

        return 0;
}
```

Array of int pointers!!

Solution

First Solution:

```
int array[ROWS][COLS];
process_array(array);
```

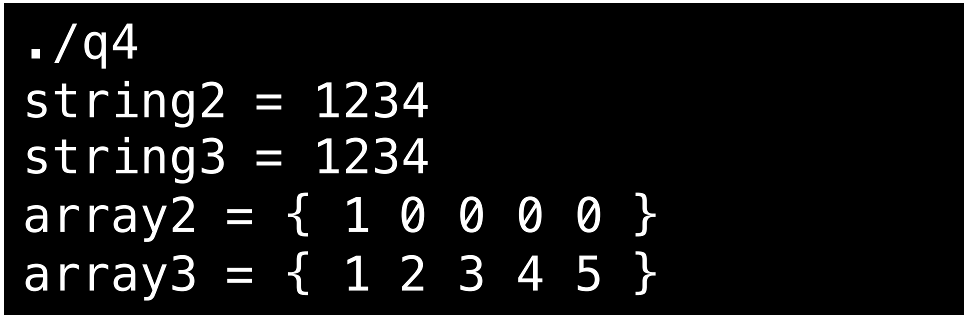
Second Solution:

```
int **array = malloc(sizeof(int) * ROWS);
    int k;
    for(k = 0;k < ROWS;k++)
        array[k] = malloc(sizeof(int) * COLS);
    process_array(array);
```

Question 4 – strncpy and memcpy?

```
char string1[SIZE] = { '1','2','3','4','\0' };
char string2[SIZE], string3[SIZE];
int array1[SIZE] = { 1, 2, 3, 4, 5 };
int array2[SIZE], array3[SIZE];

strncpy(string2, string1, sizeof(string1));
memcpy (string3, string1, sizeof(string1));

printf("string2 = %s\n", string2);
printf("string3 = %s\n", string3);

strncpy((char *) array2, (char *) array1, sizeof(array1));
memcpy(array3, array1, sizeof(array1));

print_array(array2, "array2", SIZE);
print_array(array3, "array3", SIZE);
```

Question 4 - Result



```
./q4
string2 = 1234
string3 = 1234
array2 = { 1 0 0 0 0 }
array3 = { 1 2 3 4 5 }
```

Why?_____?

String - revised

How is a string stored in C?

H	E	L	L	O	\0

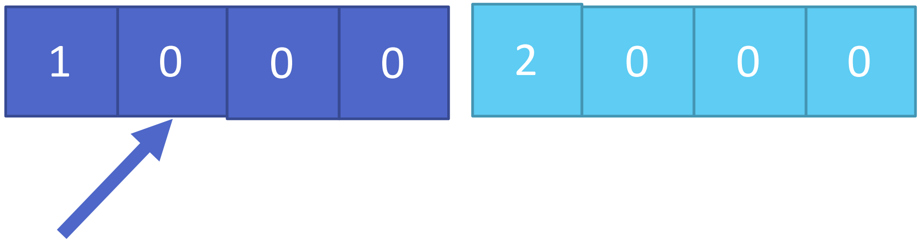
When strncpy() encountered a '\0' (Null Character), it will stop.

But… Why it stops in the example?

```
int array1[SIZE] = { 1, 2, 3, 4, 5 };
```

Forced Casting

When doing this kind of “casting”:



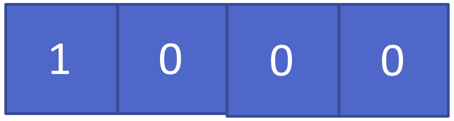
1	0	0	0	2	0	0	0

The integer occurs 4 bytes, so when it is casted to char*, it becomes “1000”.

Then when strncpy() encounters the zero after 1, it stops!

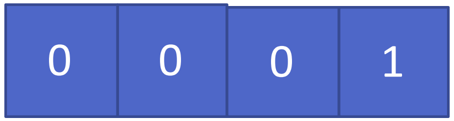
Extra: Endianness

Little Endian: (General Linux)



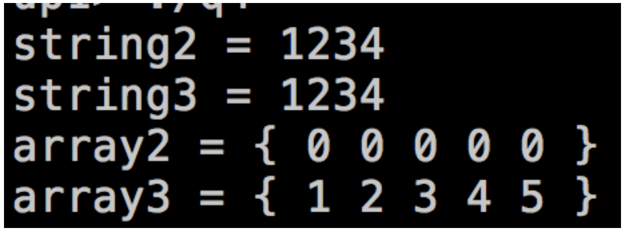
1 0 0 0

Big Endian: (Sun SPARC)



0 0 0 1

In some machine, the result will be:



```
string2 = 1234
string3 = 1234
array2 = { 0 0 0 0 0 }
array3 = { 1 2 3 4 5 }
```

END

Keep Warm and see you in next tutorial :)