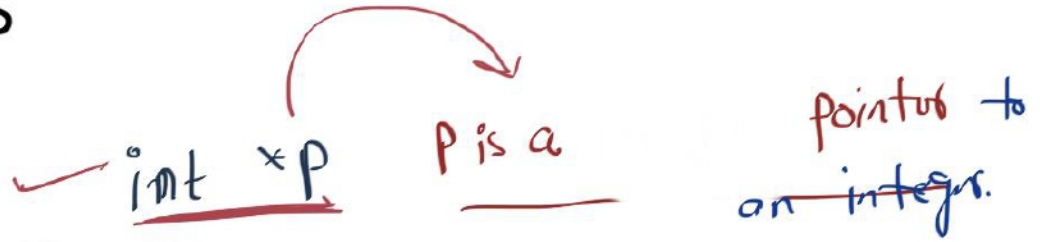
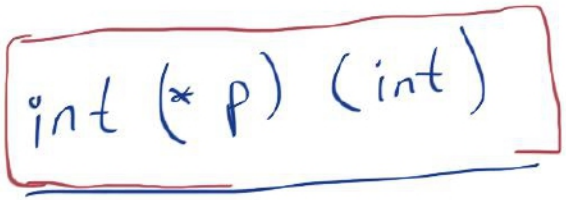
Complex Declarations



Translation to English -

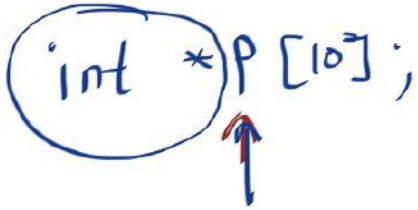
```
*      as "pointer to"
[ ]    as "array of"
( )    as "function returning"
```



STEP 1

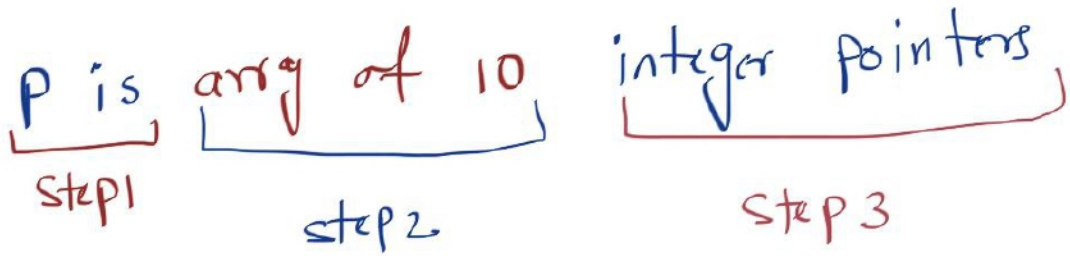
Find the identifier. This is your starting point. Then say to yourself, "identifier is." You've started your declaration.

p is



STEP 2

Look at the symbols on the right of the identifier. If, say, you find a "[ ]" there, you would say, "identifier is array of". Continue right until you run out of symbols *OR* hit a *right* parenthesis ")". In case If you find ")" - see STEP 3.



STEP 3

Keep going left (and keep translating to english) until you run out of symbols *OR* hit a *left* parenthesis "(".

STEP 1

Find the identifier. This is your starting point. Then say to yourself, "identifier is." You've started your declaration.

p is

int (*p) [10]

STEP 2

Look at the symbols on the right of the identifier. If, say, you find a "[ ]" there, you would say, "identifier is array of". Continue right until you run out of symbols *OR* hit a *right* parenthesis ")". In case If you find ")" - see STEP 3.

STEP 3

p is pointer to array of 10 integers

step 1

Keep going left (and keep translating to english) until you run out of symbols *OR* hit a *left* parenthesis "(".

in short -

Keep going right, if you find ")" then resolve it to corresponding "(" and after resolving keep going right.

```
    int *a[10];
Applying rule:


    int *a[10];       "a is"
         ^

    int *a[10];       "a is an array" of 10
          ^^^^

    int *a[10];       "a is an array of pointers"
        ^

    int *a[10];       "a is an array of pointers to `int`".
    ^^^
```

```
int (*p)[ 5 ];
```

```
int* f()
```

f is function returning int pointers.

```
int (*pf) ();
```

pf is a pointer to function returning int

```
int (*pf) (int);
```

pf is pointer to function taking int argument and returning int.

Question: What does the following C-statement declare?

```
int (*f) (int * );
```

GATE PYQ

A) A function that takes an integer pointer as argument and returns an integer

B) A function that takes an integer as argument and returns an integer pointer

C) A pointer to a function that takes an integer pointer as argument and returns an integer

D) A function that takes an integer pointer as argument and returns a function pointer

```
int * (* (*fp1) (int) ) [10];
```

fp1 is Pointer to function takes int as argument and returns pointer to array of 10 int pointers

```
int * (* (*fp1) (int) ) [10];
```

fp1 is a pointer to a function taking an int and returning a pointer to an array of 10 pointers to int

```
int (*apa[5])[10];
```

apa is an array of 5 pointers to array of 10 integers

```
int (*apa[ ])[ ];
```

an array of pointers to arrays of int

Pointer to function (Not required for GATE)

```
int my_fun(int x){
    return x+1;
}

main()
{
    int (*p)(int);

    printf("%d", p(2));
}
```

my_fun is called a pointer to a function.

p = my_fun

my_fun(2)

Pointer to function (Not required for GATE)

```c
int my_fun(int x){
    return x+1;
}


main()
{
    int (*p)(int) = my_fun;

    printf("%d", p(2));
}
```

my_fun is called a pointer to a function.

GO CLASSES

Void Pointer

Void Pointer

```
int main()

{
    int a = 7;
    void *p;
```

int *x = ← the address of an integer

char *c = ← the address of an character

void *p = ~~~~ ↑ we can supply any address here

Void Pointer

```
int main()

{
    int a = 7;
    void *p;
    p = &a;
```

(int)p

We can not directly dereference the void ptr.

Void Pointer

(int *) *p wrong

```
int main()

{
    int a = 7;
    void *p;
    p = &a;
    printf("Integer variable is = %d", *( (int*) p) );

}
```

(int*) p

↓

first type cast

→ void ptr can hold any type of address.

→ We can not directly dereference it

→ Before dereference, we need to typecast.

GO CLASSES

```
int n = 5;
char c = 'a';
void *p;
```

p = &n

p = &c

(char)p

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

=

sizeof(int*)

=

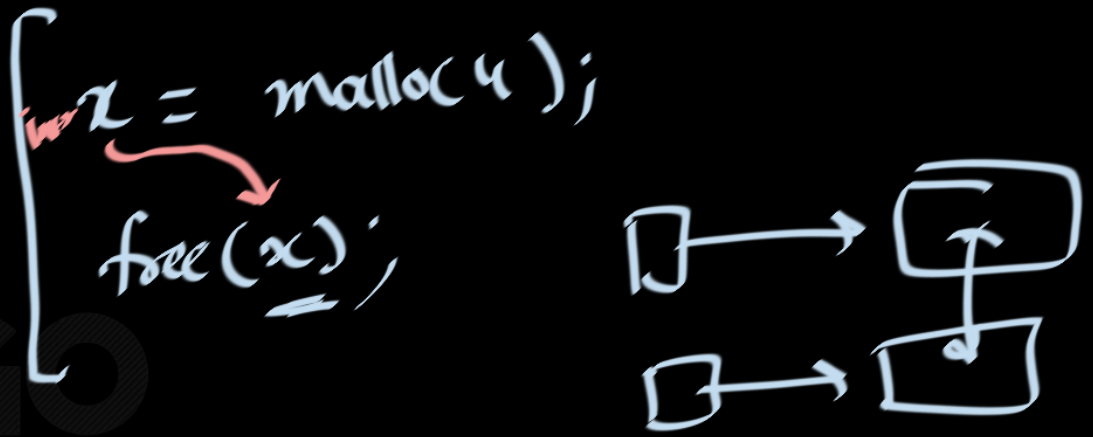
sizeof(char*)

=

8

OPTIONAL Examples -

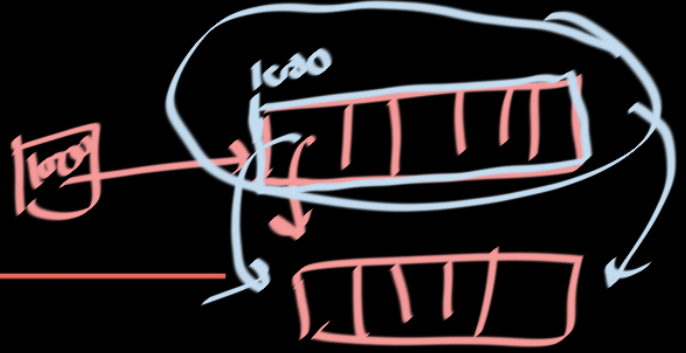
1. `void free(void *ptr);`



2.
```
void memcpy(void *dst, void *src, int size)
{
    char *d = dst;
    char *s = srcs;
    int i = 0;

    while (i < size) {
        d[i] = s[i];
        i++;
    }
}
```

No. of bytes

3.
```
void Swap(void *one, void *two, int elemSize) {
   char temp[elemSize];
   memcpy(temp, one, elemSize);
   memcpy(one, two, elemSize);
   memcpy(two, temp, elemSize);
}
```



GO CLASSES

You think you know pointers?

You think you know pointers?

```
int (*t)[10];

What is t ?
What is *t ?
What is (*t)[1] ?
What is *t[5] ?
What is *(t+1) ?
What is t[5] ?
What is *(t[1]+3) ?
What is t[1][2] ?
What is **(t+1) ?
```

Pointer to 1D array of integers

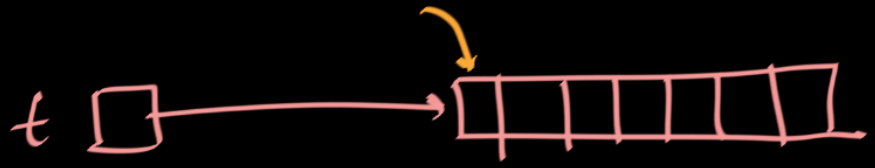
Points to the first element

2nd int of array

first element of 6th array

Pointer to the first element of 2nd array

```
int t[10][20];

What is t ?
What is *t ?
What is (*t)[1] ?
What is *t[5] ?
What is *(t+1) ?
What is t[5] ?
What is *(t[1]+3) ?
What is t[1][2] ?
What is **(t+1) ?
```



t : Pointer to 1D array

int a[4]

*t : 1D array

int (*t) [10];

* *

* []

[] []


*t[5]
t
*(t+5)

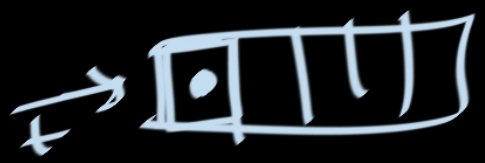
You think you know pointers? (Cond..)

```
int *t[10];
What is t ?
What is *t ?
What is (*t)[1] ?
What is *(t+1) ?
What is **(t+1) ?
What is t[5] ?
What is *(t[1]+2) ?
What is t[1][2] ?
```

t is an array of pointers

first element of the array

```
int t[10];
What is t ?
What is *t ?
What is t[1] ?
```



You think you know pointers? (Cond..)

```
int *t;

What is t ?

What is *t ?
```

```
int **t;

What is t ?

What is *t ?

What is **t ?
```

Easy questions on draw a diagram

Question 1: Draw Diagram

```
double d = 5.0;  /* a double                    */
double *pd = &d; /* a pointer to a double */
```

Answer

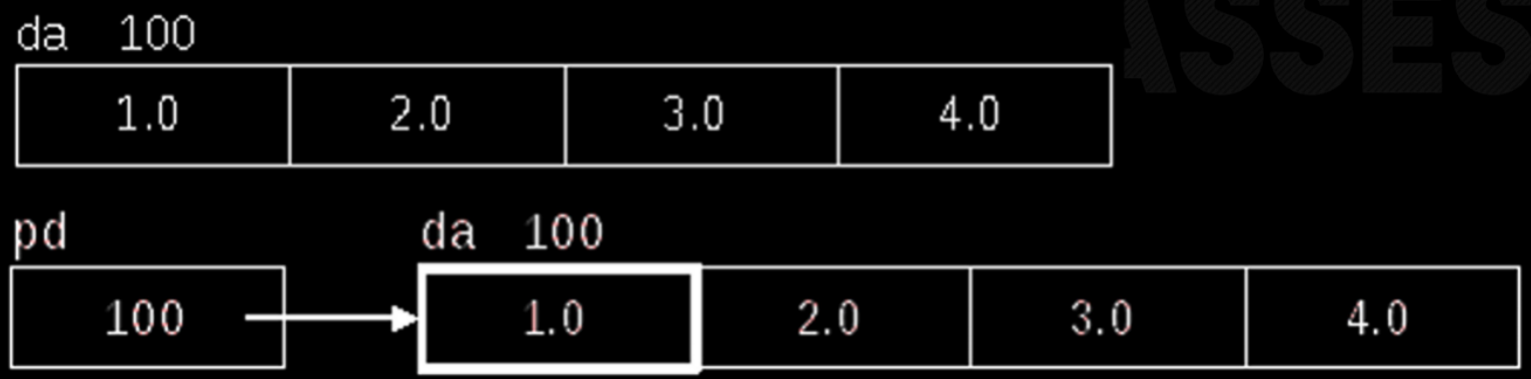
```
double d = 5.0;  /* a double                  */
double *pd = &d; /* a pointer to a double */
```

Question 2: Draw Diagram

```
double da[4] = {1.0, 2.0, 3.0, 4.0};
double *pd = da;
```

Answer

```
double da[4] = {1.0, 2.0, 3.0, 4.0};
double *pd = da;
```

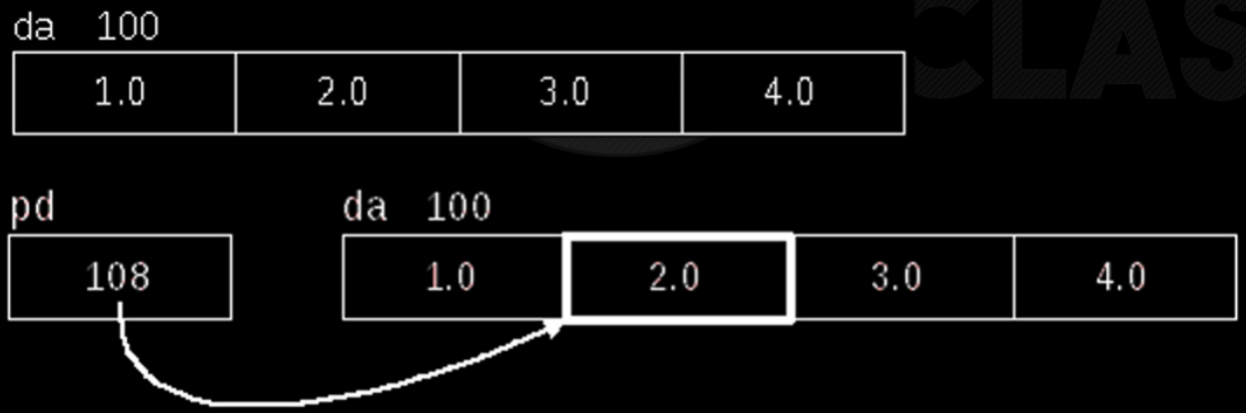


Question 3: Draw Diagram

```
double da[4] = {1.0, 2.0, 3.0, 4.0};
double *pd = &da[1];
```

Answer

```
double da[4] = {1.0, 2.0, 3.0, 4.0};
double *pd = &da[1];
```

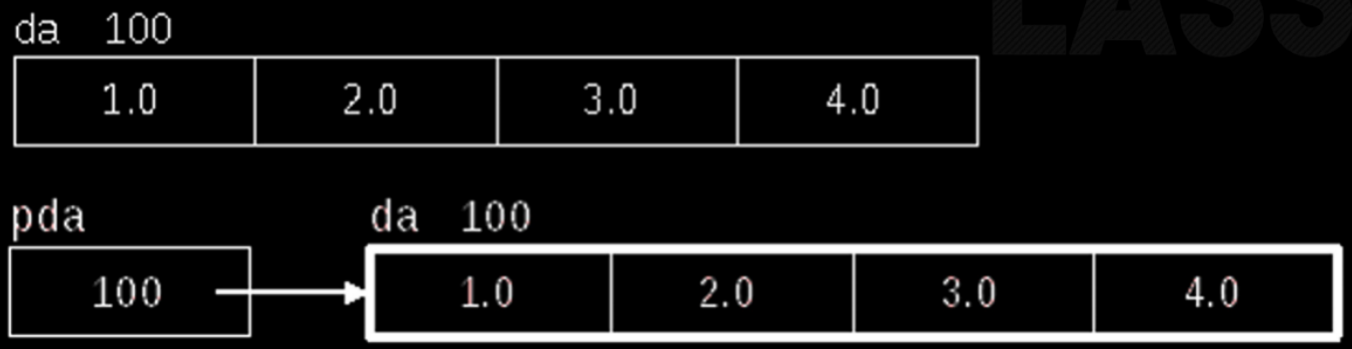


Question 4: Draw Diagram

```
double da[4] = {1.0, 2.0, 3.0, 4.0};
double (*pda)[4] = &da;
```

Answer

```
double da[4] = {1.0, 2.0, 3.0, 4.0};
double (*pda)[4] = &da;
```

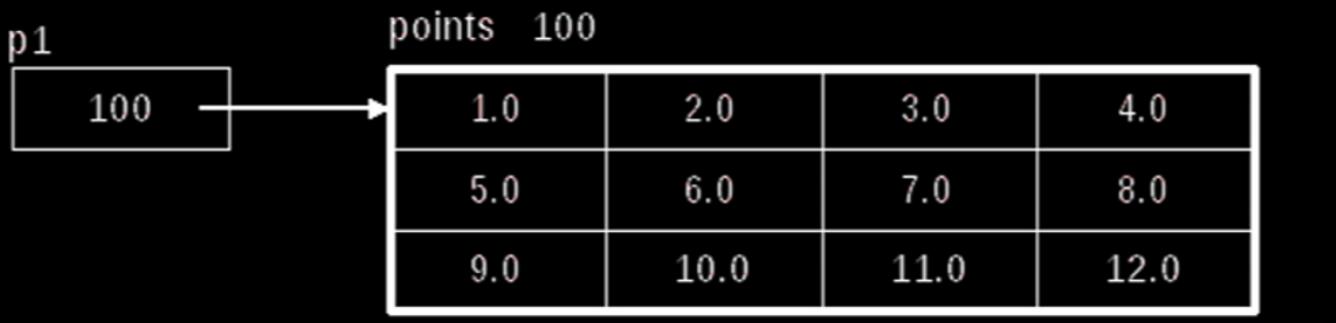


Question 5: Draw Diagram

```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
double (*p1)[3][4] = &points; /* pointer to an array of 3 arrays of 4 doubles */
```

Answer

```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
double (*p1)[3][4] = &points; /* pointer to an array of 3 arrays of 4 doubles */
```

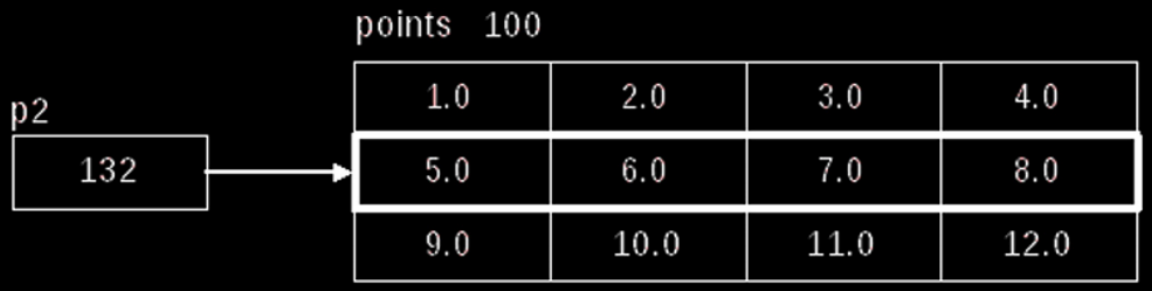


Question 6: Draw Diagram

```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
double (*p2)[4] = &points[1]; /* pointer to an array of 4 doubles (2nd element/row) */
```

Answer

```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
double (*p2)[4] = &points[1]; /* pointer to an array of 4 doubles (2nd element/row) */
```

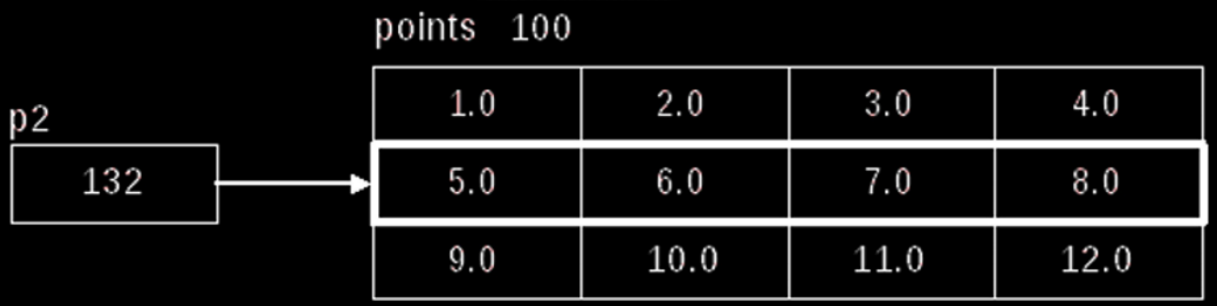


Question 7: Draw Diagram

```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
double (*p2)[4] = points+1; /* pointer to an array of 4 doubles (2nd element/row) */
```

Answer

```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
double (*p2)[4] = points+1; /* pointer to an array of 4 doubles (2nd element/row) */
```

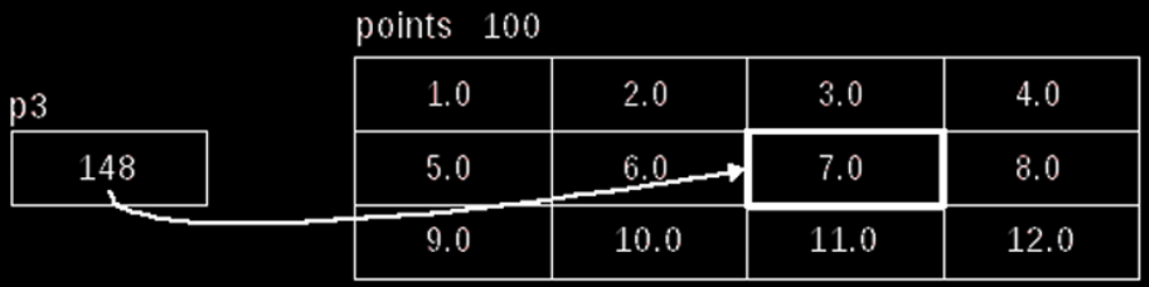


Question 8: Draw Diagram

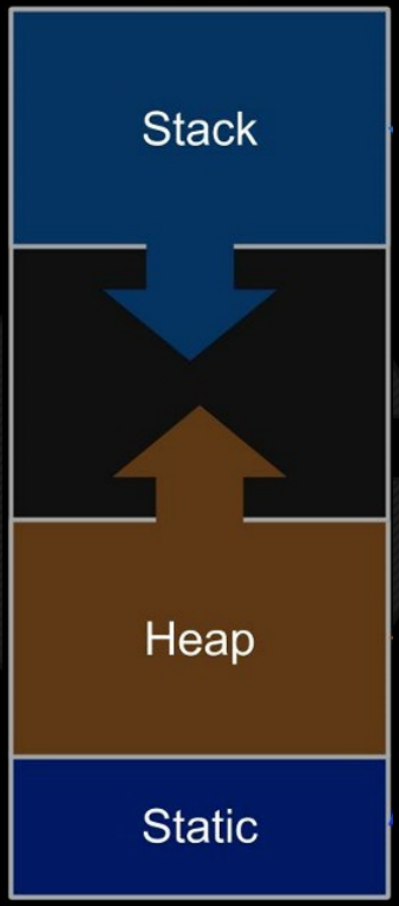
```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
double (*p2)[4] = &points[1]; /* pointer to an array of 4 doubles (2nd element/row) */
double *p3 = &(*p2)[2]; /* a pointer to a double (3rd element of 2nd row) */
```

Answer

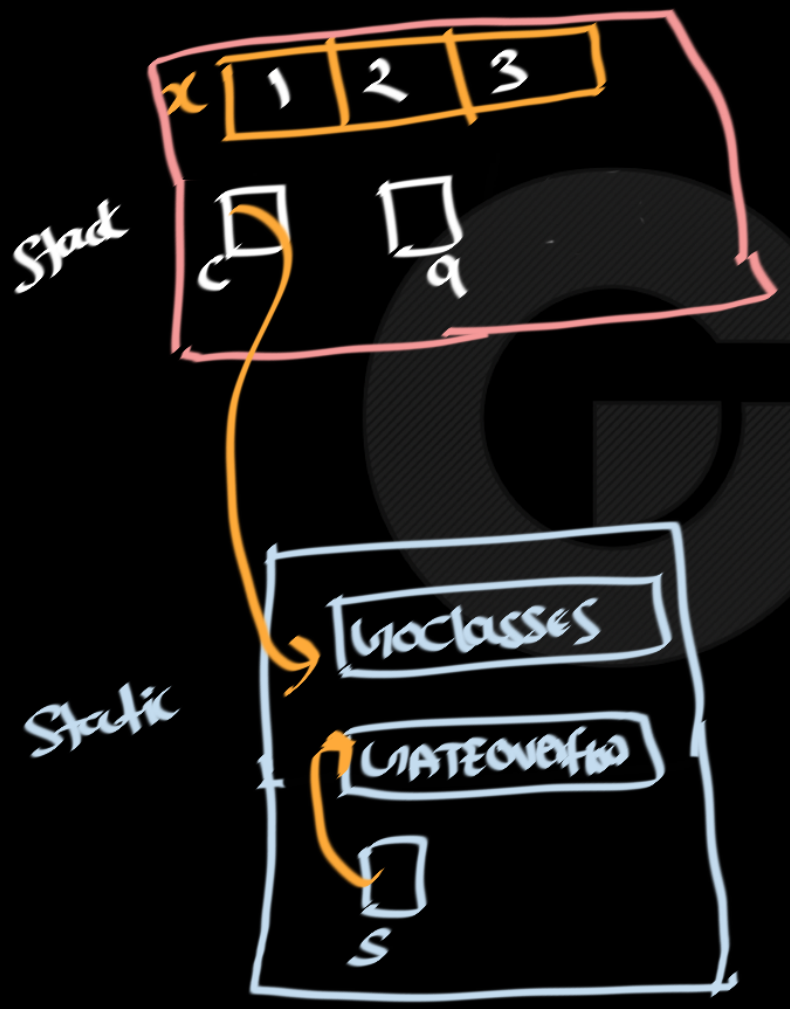
```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
double (*p2)[4] = &points[1]; /* pointer to an array of 4 doubles (2nd element/row) */
double *p3 = &(*p2)[2]; /* a pointer to a double (3rd element of 2nd row) */
```



Malloc and Free



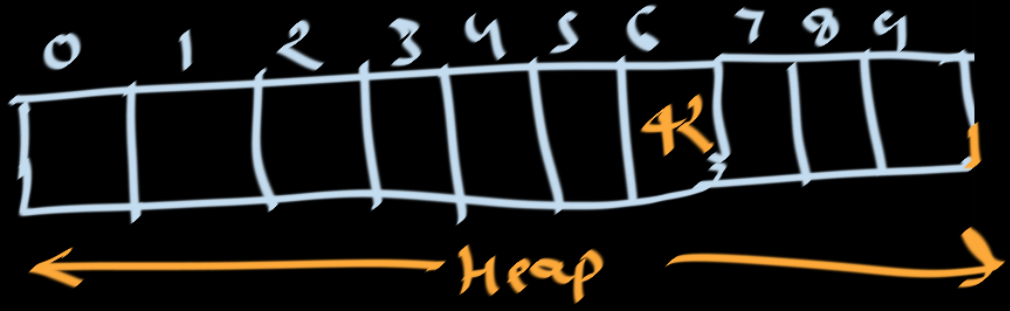

```
main( )
{
    char *c = "GOclasses"
    int a;
    static char *s = "GATEOverflow"
    int x[3] = {1,2,3}
}
```

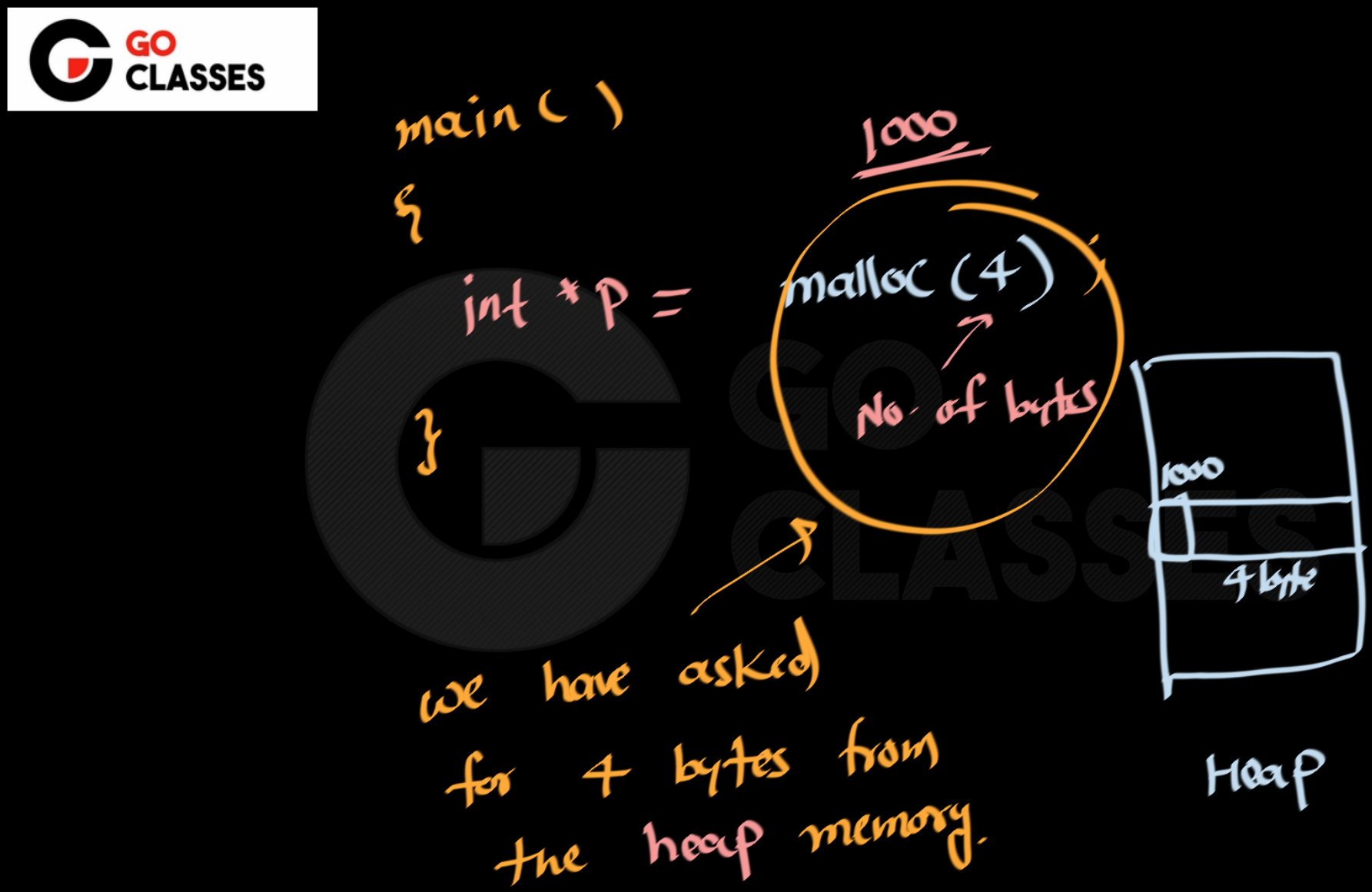


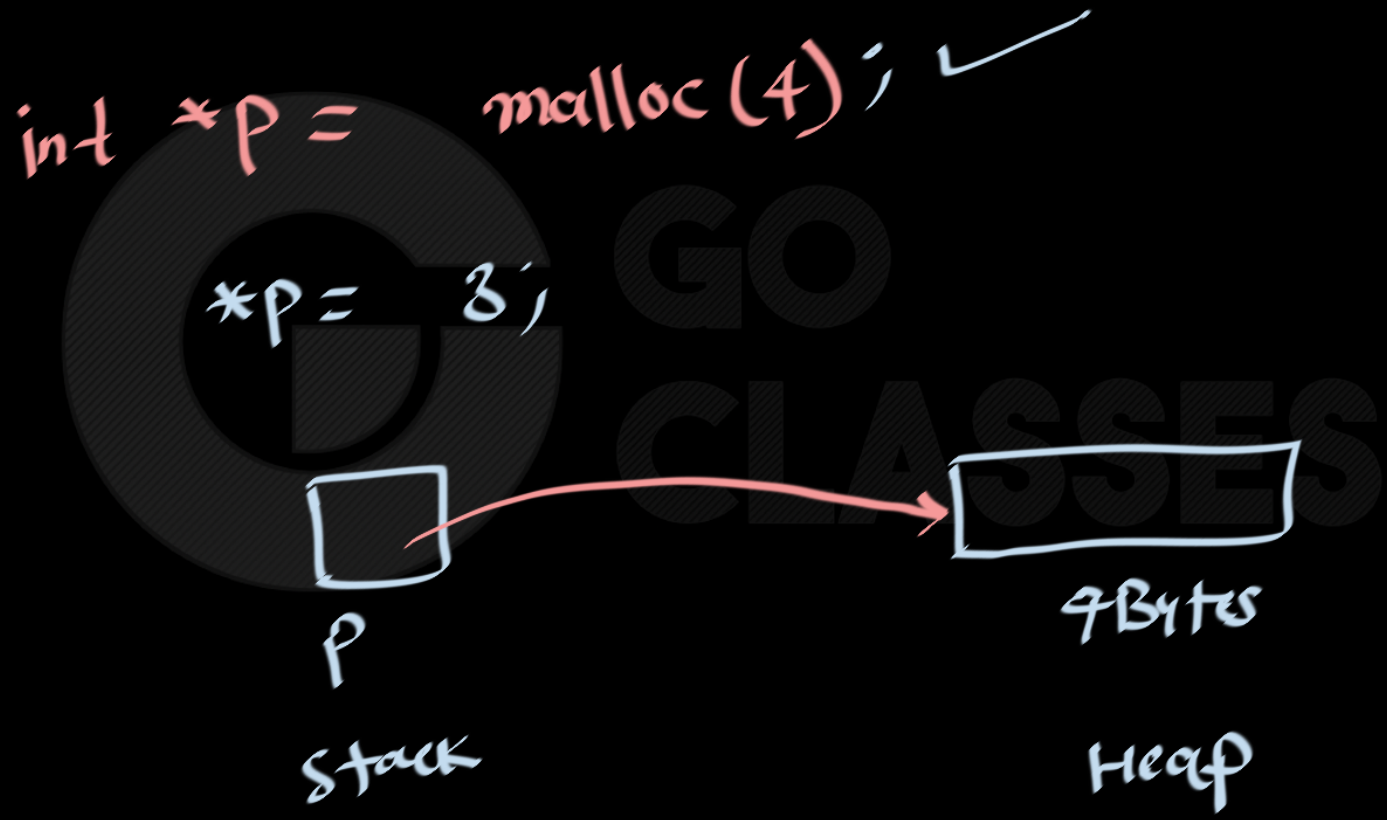

```
main( )
{
    char *c = "GOclasses"
    int a;
    static char *s = "GATEOverflow"
    int x[3] = {1,2,3}
}
```

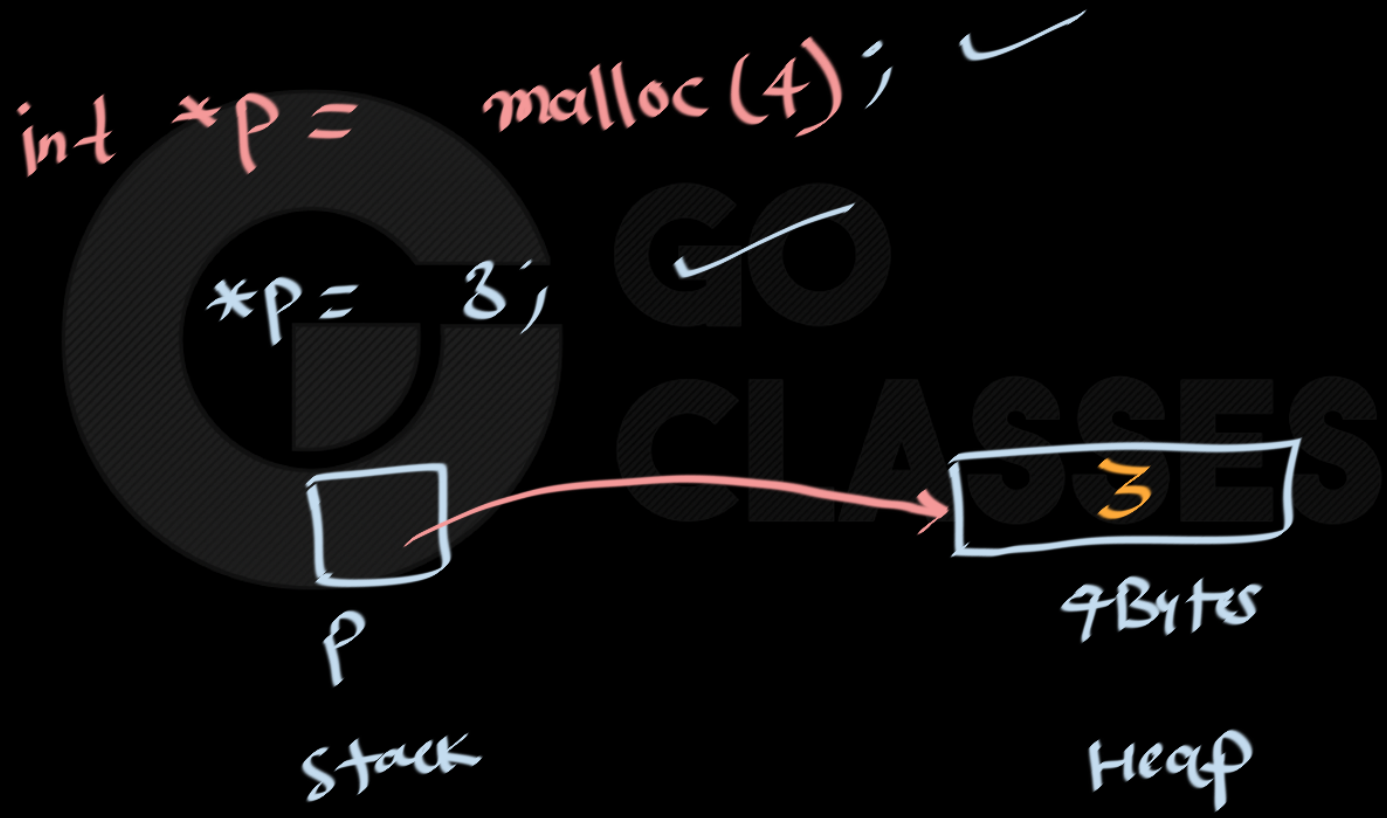
malloc function in c

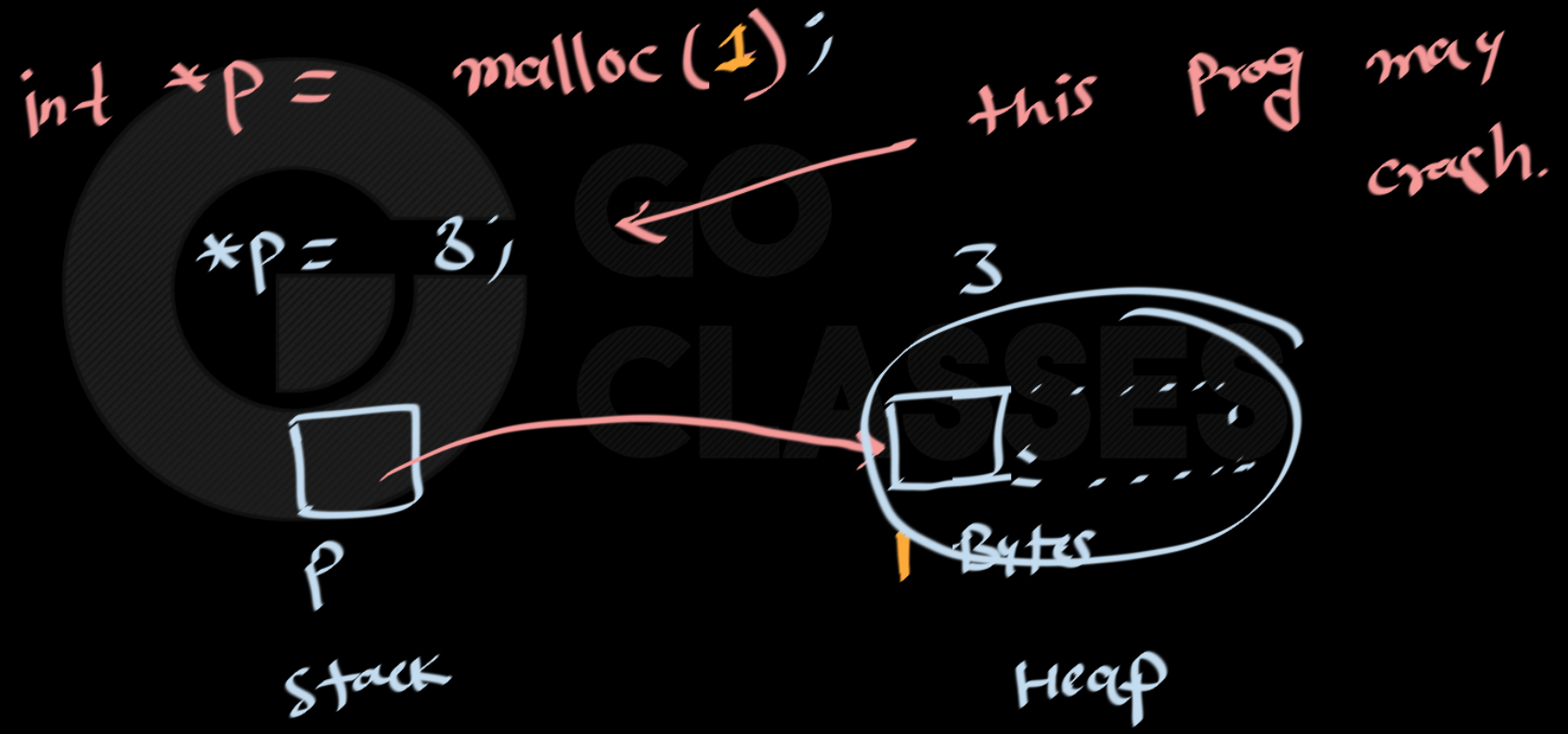
```c
int *ip;
ip = (int *) malloc( sizeof(int)*10 ); // allocate 10 ints
ip[6] = 42; // set the 7th element to 42
```

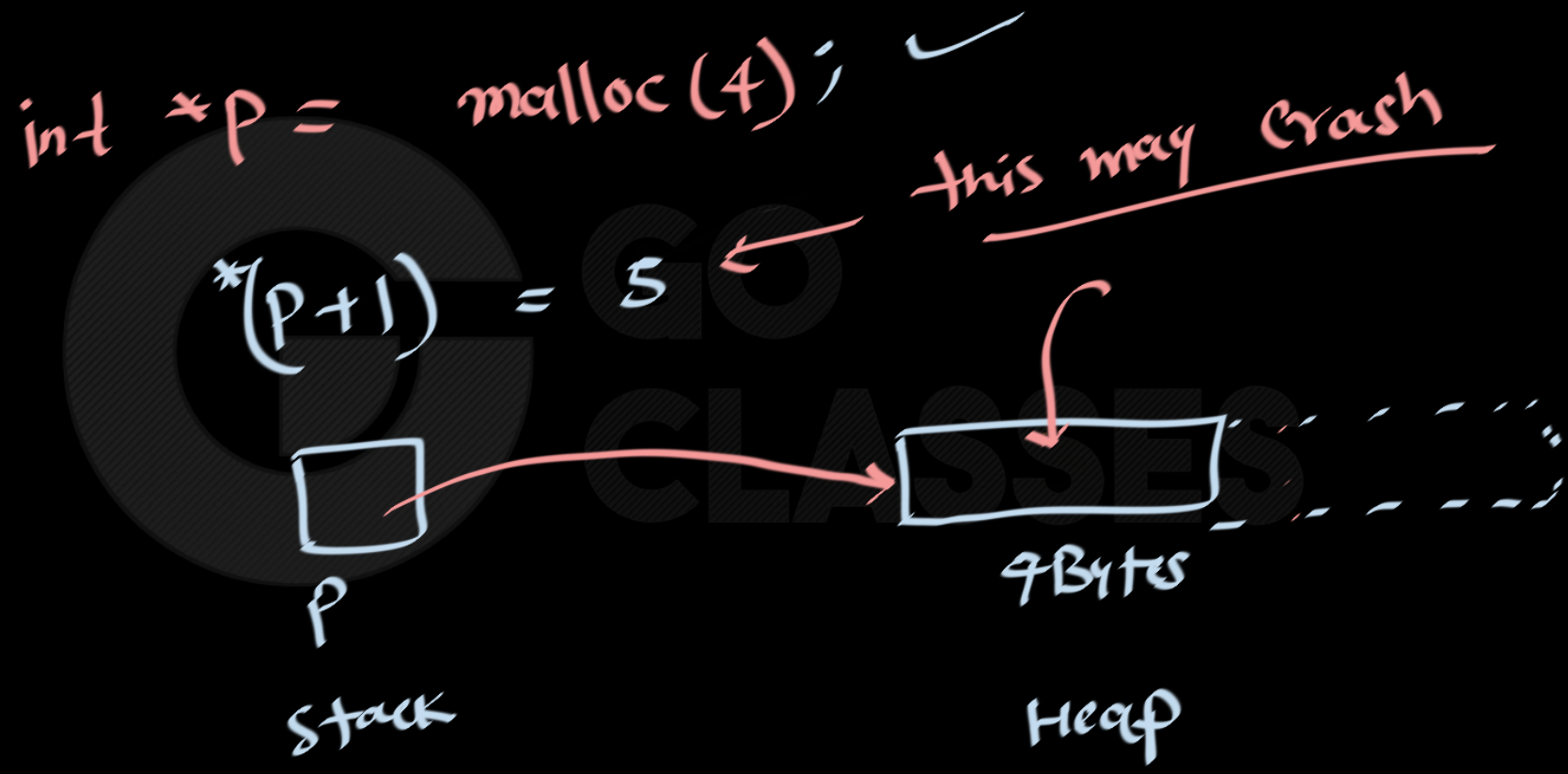



main ( )
{
int *p =
malloc (4) ;
}
1000
No. of bytes
we have asked for 4 bytes from the heap memory.
1000
4 byte
Heap


int *p = malloc(4);
*p = 3;
p
4Bytes
Stack
Heap


int *p = malloc(4);
*p = 3;
3
p
4Bytes
Stack
Heap


int *p = malloc(1);
this prog may crash.
*p = 3;
3
p
Bytes
stack
Heap


int *p = malloc(4);
this may crash
*(p+1) = 5
p
4Bytes
Stack
Heap

GO CLASSES

malloc(40);

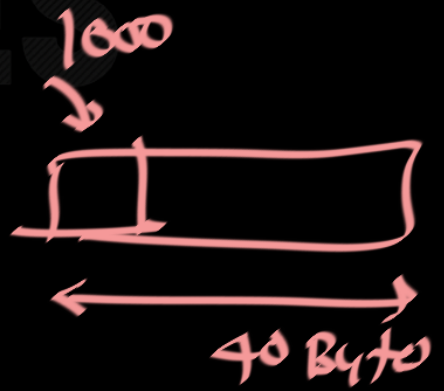
- malloc() doesn't know how you are going to use this memory.
- for example if we ask for 40 Bytes then you may store
  1) 40 Characters
  2) 10 integers
  3) 5 integers 20 chars
  4) 5 doubles

GO CLASSES

malloc(40);

Return type of malloc is void *.

1000
malloc(40);



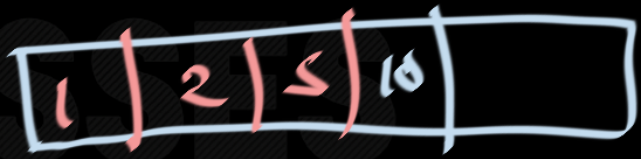
GO CLASSES

asking memory from heap

```
int *p = malloc(10 x sizeof(int))
```

using the memory

```
*p = 1
*(p+1) = 2
*(p+3) = 5
p[4] = 10
```

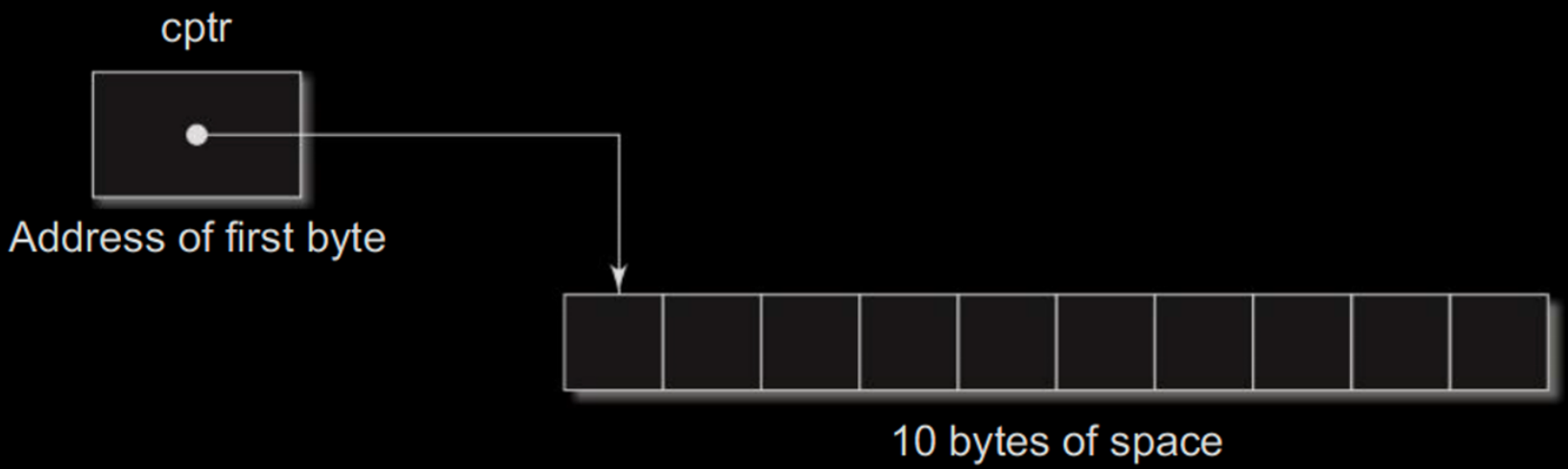


Heap

40 Bytes

char *cptr;

```
cptr = (char*) malloc(10);
```



Question 1

Which of the following represents the most correct use of malloc?

- `int x = malloc(sizeof(int));`
- `int *x = (void *)malloc(sizeof(char));`
- `int *x = (int *)malloc(sizeof(int));`
- `int *x = (int *)malloc(4);`

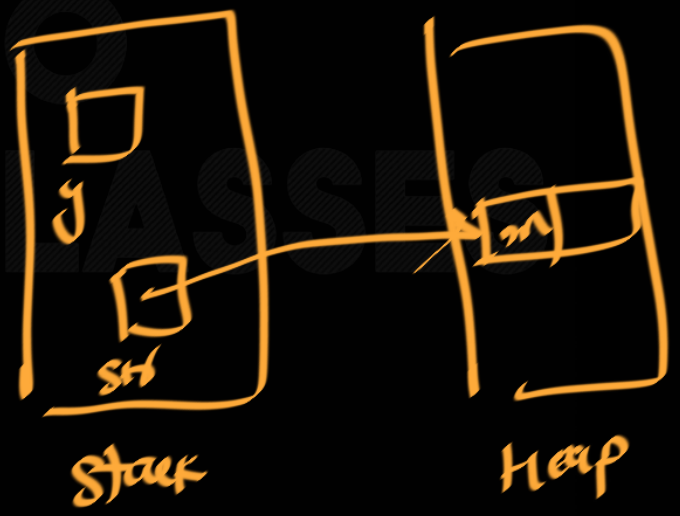
https://www.cs.virginia.edu/~jh2jf/courses/cs2130/spring2023/exams/s2022e2key.pdf

malloc function in c (Contd...)

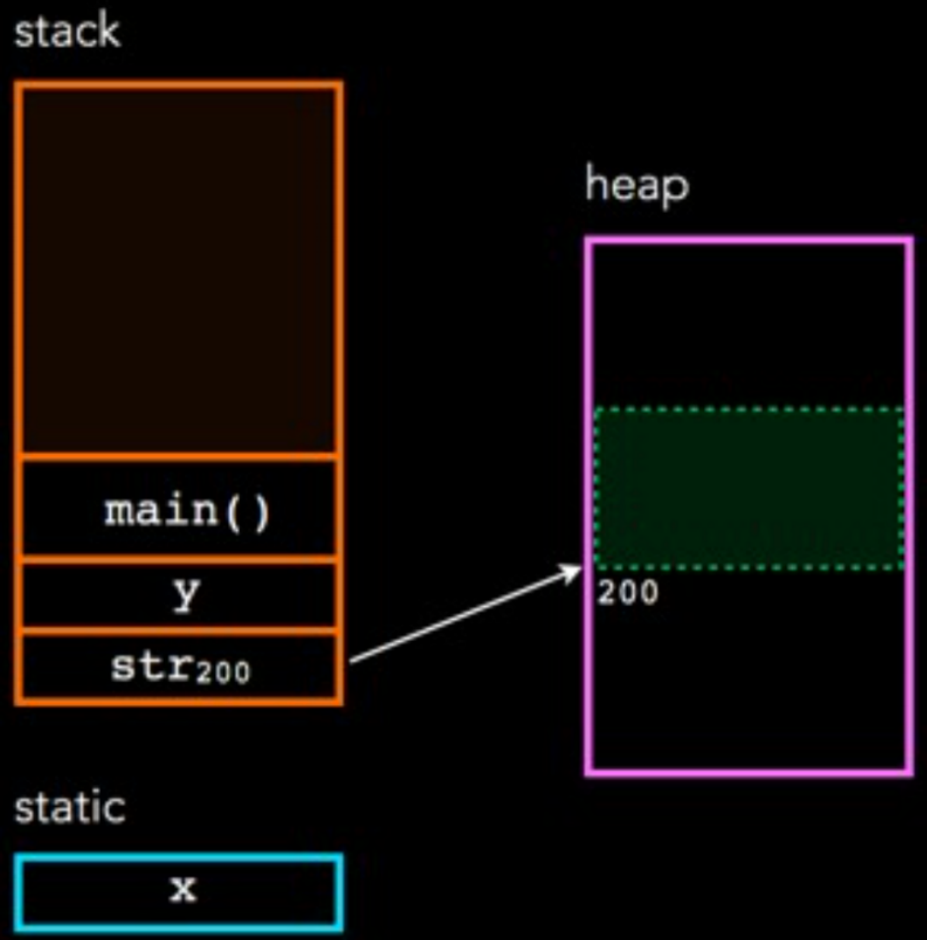
- malloc returns the void type pointer, we can use that without typecast.
- malloc returns the void* because, malloc doesn't know for what purpose this memory is being used.
- On fail, malloc returns NULL

Example

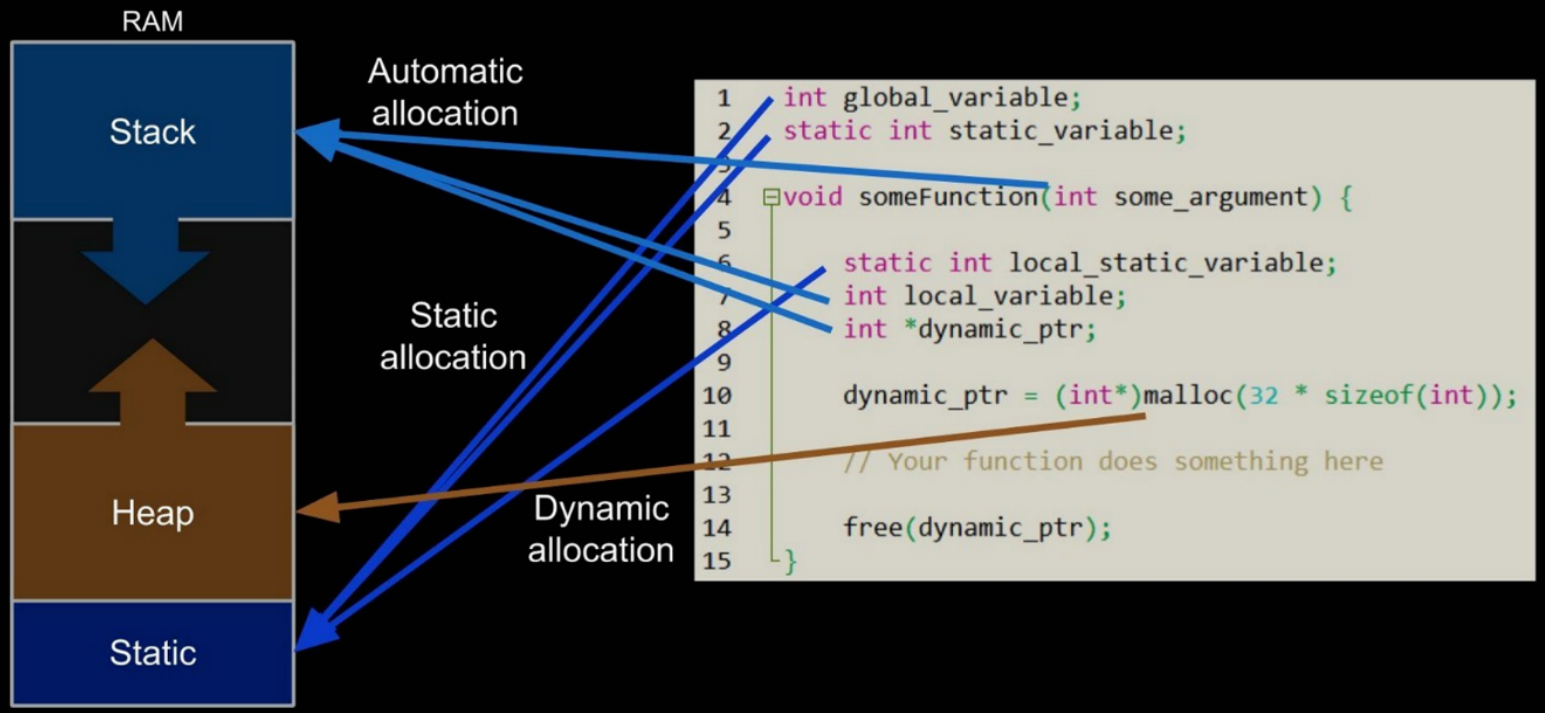
```c
#include <stdio.h>


int x;
int main(void)
{
    int y;
    char *str

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c", str[0]);
    free(str);
    return 0;

}
```



```c
#include <stdio.h>


int x;
int main(void)
{
    int y;
    char *str

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c", str[0]);
    free(str);
    return 0;

}
```



Example


RAM
Stack
Heap
Static
Automatic allocation
Static allocation
Dynamic allocation
1 int global_variable;
2 static int static_variable;
4 void someFunction(int some_argument) {
6 static int local_static_variable;
7 int local_variable;
8 int *dynamic_ptr;
10 dynamic_ptr = (int*)malloc(32 * sizeof(int));
12 // Your function does something here
14 free(dynamic_ptr);
15 }

Question 2

In the following code snippet, where are a, b, and c stored in memory?

```
int foo() {
  int a = 9;
  int b[3] = {2, 7, 8};
  char *c = (char *) malloc(100);
}
```

- b and c are stored on the stack, a is stored on the heap
- b is stored on the stack, a and c are stored on the heap
- ✓ All are stored in the stack
- a and b are stored on the stack, c is stored on the heap

Question 3

6. [2 points] Assuming the following line of code is inside the main( ) function, in which part of memory is the pointer variable `parray` allocated and in which part of memory is the 10 element integer array allocated?

```
int *parray = malloc (sizeof (int) * 10);
```

https://pages.cs.wisc.edu/~remzi/Classes/354/Spring2017/OldExams/midterm-spring-16.pdf

Solution

6. [2 points] Assuming the following line of code is inside the main( ) function, in which part of memory is the pointer variable `parray` allocated and in which part of memory is the 10 element integer array allocated?

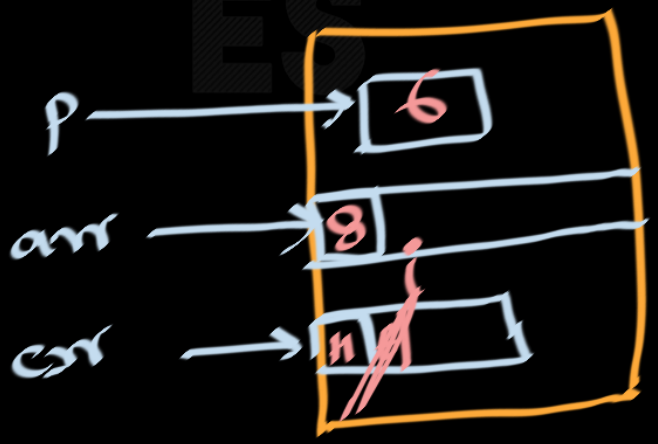
```
int *parray = malloc (sizeof (int) * 10);
```

ANSWER:

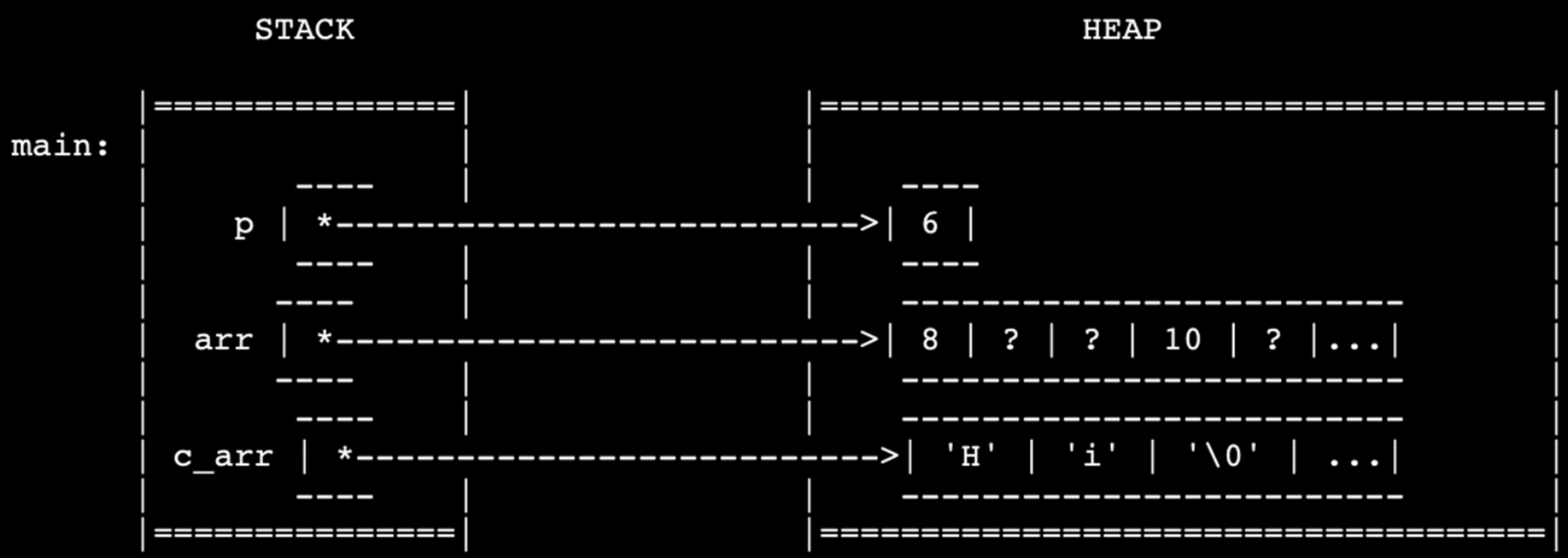
`parray` is allocated in **stack** memory.

10 element integer array is allocated in **heap** memory.

```c
int *p, *arr;
char *c_arr;

p = (int *)malloc(sizeof(int));
arr = (int *)malloc(sizeof(int)*20);
c_arr = (char *)malloc(sizeof(char)*10);

if((p == NULL) || (arr == NULL) || (c_arr == NULL)) {
   printf("malloc error\n");
}

*p = 6;
*arr = 8;
arr[3] = 10;
strcpy(c_arr, "He");
c_arr[1] = 'i';
```



The contents of memory will look like this:



Releasing the used space: Free

```
free (ptr);
```

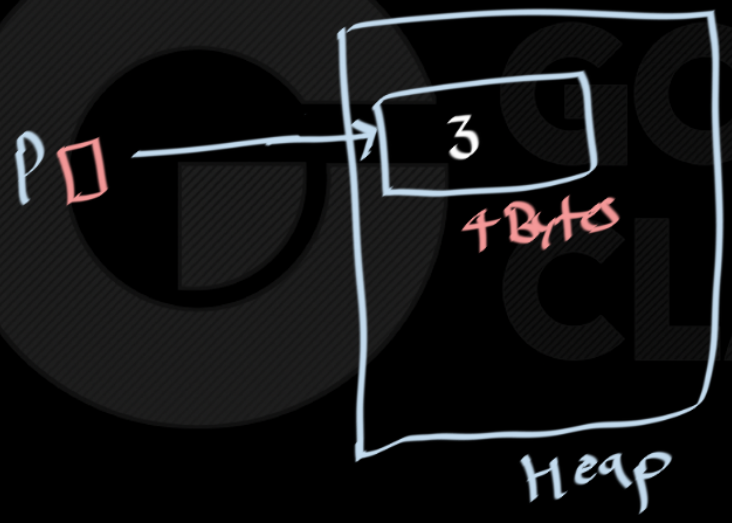
GO CLASSES

```
int * p = malloc ( sizeof(int));
*p = 3;
int n = *p x 2;
free(p)
```

GO CLASSES

What free(p) will do ?

```
int *p = malloc(4);
```




```
*p = 3;
.....
free(p);
```

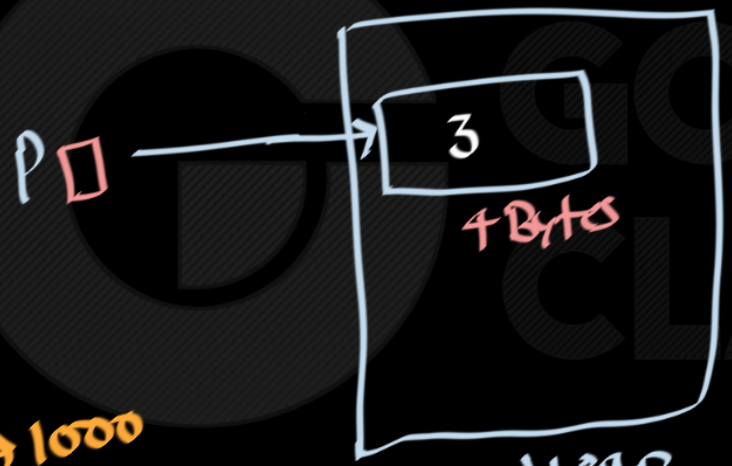
option1 : it will replace 3 with 0 `*p=0;`

option2: it will set p to null `p= null`

option3 : `p= null` ✗ OS will reclaim the memory `*p= 3` ✗

GO CLASSES

```
int *p = malloc(4);
```

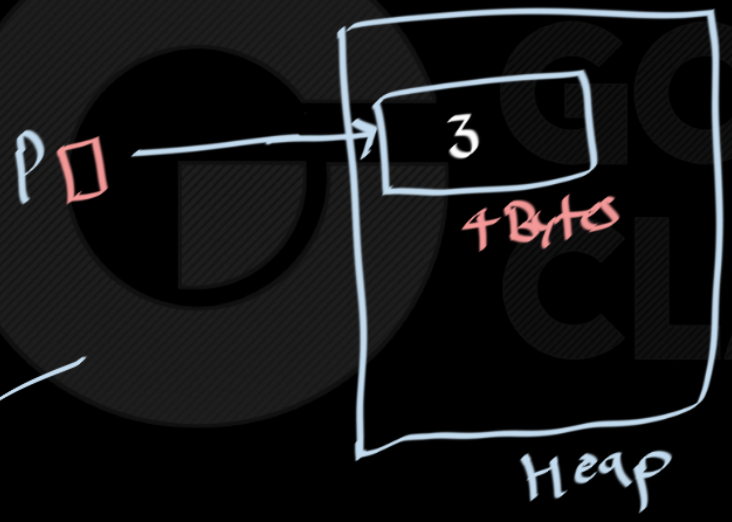



```
*p = 3;
.....
Pf(p) → 1000
free(p);
Pf(p) → 1000
```

OS now will reclaim the memory

GO CLASSES

```
int *p = malloc(4);
```




```
*p = 3;
.....
*p = 5;
free(p);
```

*p ← Program may crash

GO CLASSES

```
int *t ;
t= malloc (12) ;
*t = 5;
t[1] = 6;
t[2] = 7;
free (t);
```

*t ← may crash

GO CLASSES

free(p)

it is a way to tell OS that we (Programmer) don't need the memory Pointed by p.

*p → Prog may crash.

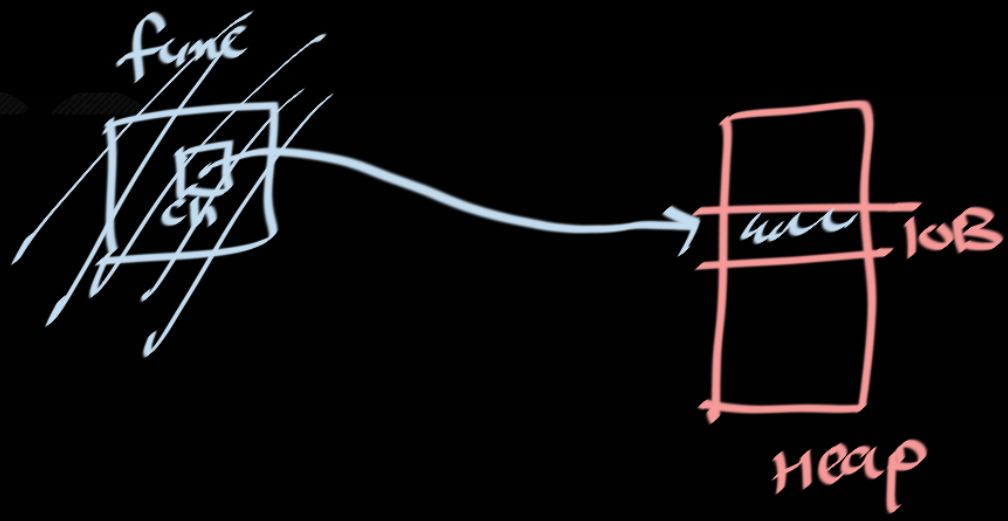
→ after free(p): what OS does with the memory? ⇒ None of your business.

GO CLASSES

Memory Leak and Dangling Pointer

Memory Leak

```
void func(){
    char *ch = malloc(10);
}


main(){

    func();
    //ch not valid outside, no way to access malloc-ed memory
}
```



Memory Leak

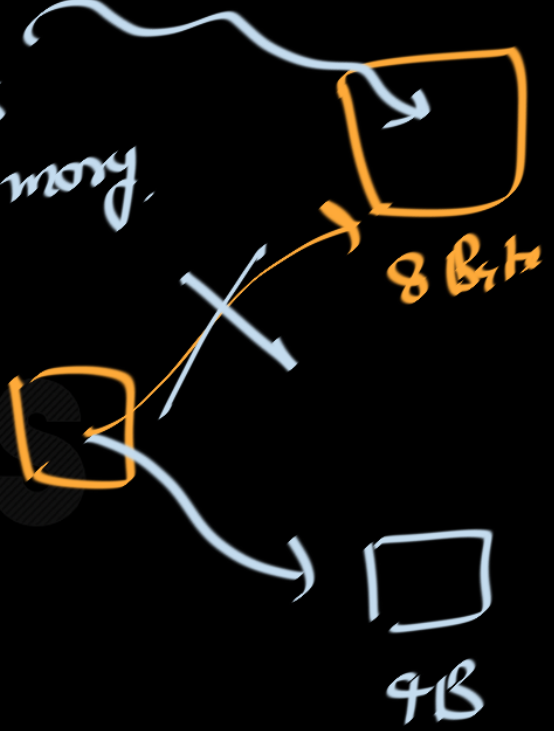
- A **memory leak** is memory which hasn't been freed, there is no way to access (or free it) now

```
void func(){
    char *ch = malloc(10);
}


main(){

    func();
    //ch not valid outside, no way to access malloc-ed memory
}
```

GO CLASSES

there is no way to access this memory.

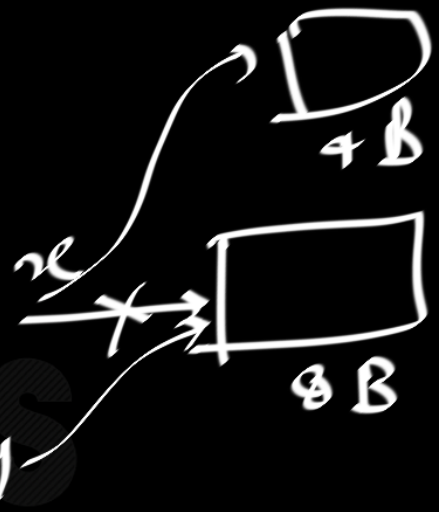
```
main( )
{
    int **x;
    x = malloc(8);
    x = malloc(4);
}
```



GO CLASSES

is this a memory leak?

```
main( )
{
    int **x, *y;
    x = malloc(8);
    y = x;
    x = malloc(4);
}
```



NO

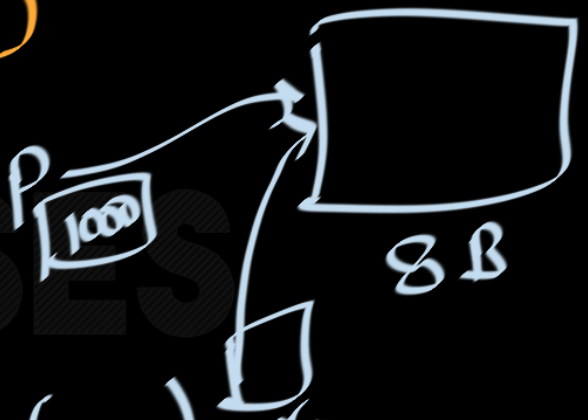
GO CLASSES

```
main ( )
{
    malloc (8);
    - - -
    - -
}
```

Yes
memory
leak

GO CLASSES

```
int *  fun( )
{
    int  *p = malloc(8);
    return p;
}
```



No, it is not a memory leak

```
main( )
{
    int *x = fun( );
}
```

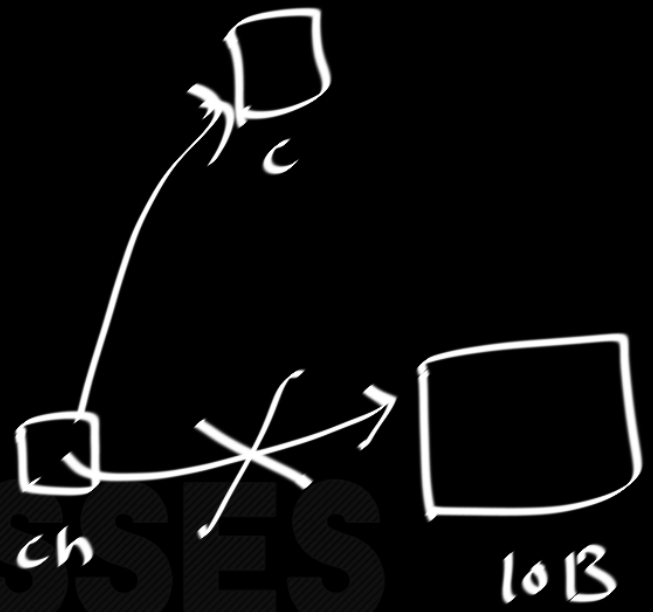
GO CLASSES

```
void func(){
    char *ch = malloc(10);
    free(ch);
}

main(){
    func();
}
```

No memory leak.

GO CLASSES

```
void func(){
    char c = 'a'
    char *ch = malloc(10);
    Ch = &c;
}

main(){

    func();
    //ch not valid outside,
}
```

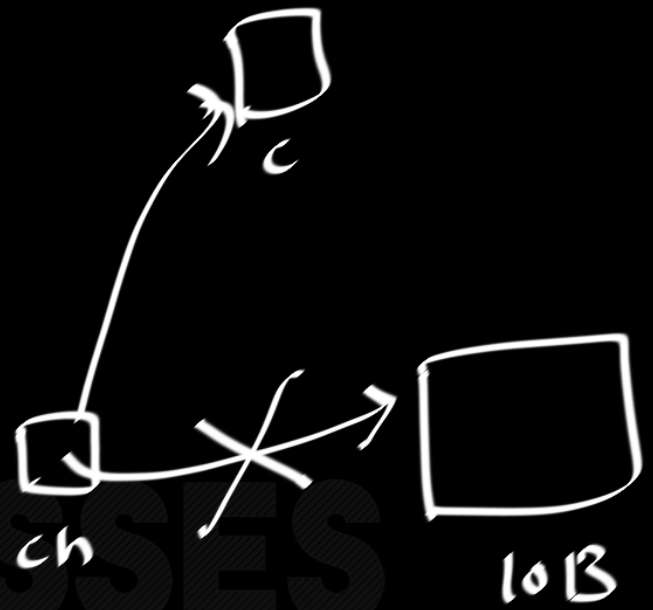


This is memory leak

GO CLASSES

```
void func(){
    char c = 'a'
    char *ch = malloc(10);
    ch = &c;
}

main(){

    func();
    //ch not valid outside,
}
```

free(ch);



Memory Leak

- A **memory leak** is memory which hasn't been freed, there is no way to access (or free it) now

```
void func(){
    char *ch = malloc(10);
}


main(){

    func();
    //ch not valid outside, no way to access malloc-ed memory
}
```

Memory leak is crime

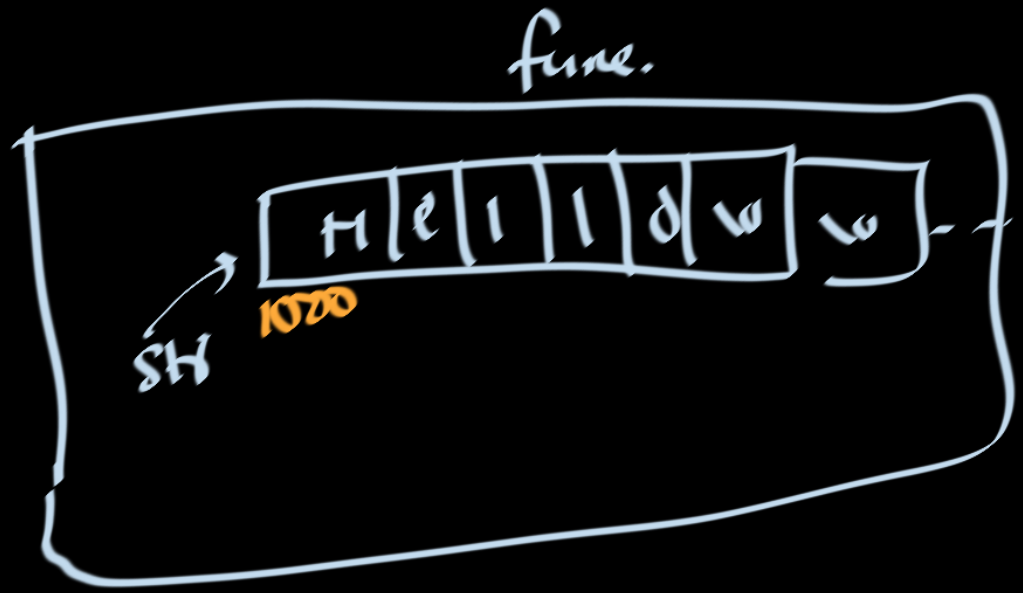
"I promise to always free each chunk of memory that I allocate."

https://ocw.mit.edu/courses/6-s096-effective-programming-in-c-and-c-january-iap-2014/resources/mit6_s096iap14_lecture2/

Dangling Pointer

- A **dangling pointer** points to memory that has already been freed

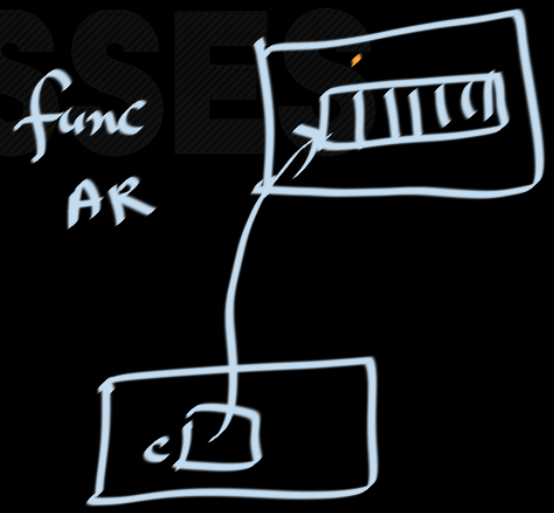
```
char *func()
{
    char str[10]={'H', 'e', 'l', 'l', 'o'};
   return str;
}

main(){

    char *c;
    c= func();
}
```



Dangling Pointer

- A **dangling pointer** points to memory that has already been freed

```
char *func()
{
    char str[10]={'H', 'e', 'l', 'l', 'o'};
   return str;
}

main(){

    char *c;
    c= func();

}
```



GO CLASSES

```
main( )
{
    int *x = malloc(8);
    ....
    free(x);
    *x = 5;
}
```

dangling pointer issue

Dangling Pointer (Contd..)

```
int *c = malloc(sizeof(int));
free(c);
*c = 3; //writing to freed location!
```

Question 4

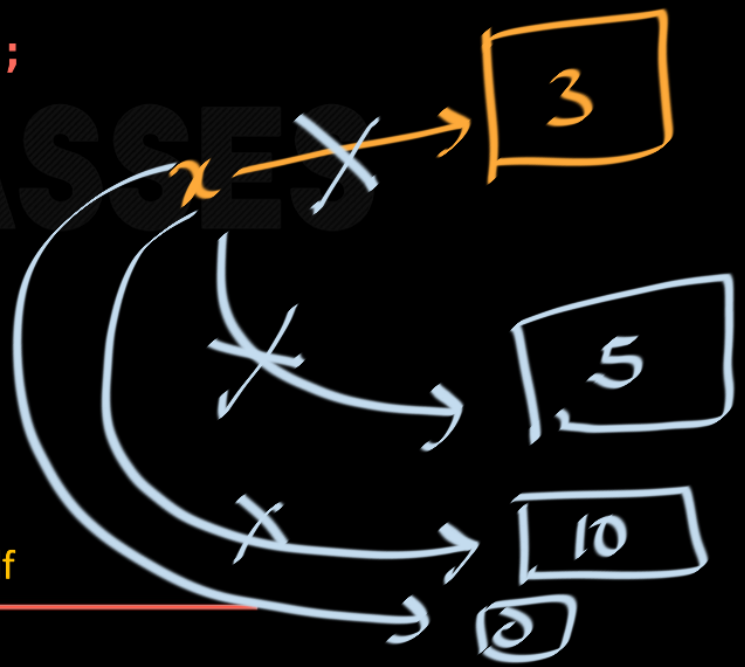
```c
int main() {
    char *p;
    p = (char *) malloc(sizeof(char));
    *p = 'a';

    p = (char *) malloc(sizeof(char));
    *p = 'c';

    return 0;
}
```

A. Syntax error in `*p = 'a';`
B. Memory leak
C. Dangling pointer
D. No issue with code

Question 5

```c
int main() {
    int *x;

    do {

        x = (int *)malloc(sizeof(int));
        printf("Enter an integer (0 to stop): ");
        scanf("%d", x);
        printf("You entered %d\n", *x);

    } while (*x != 0);

    free(x);

    return 0;
}
```

Memory leak

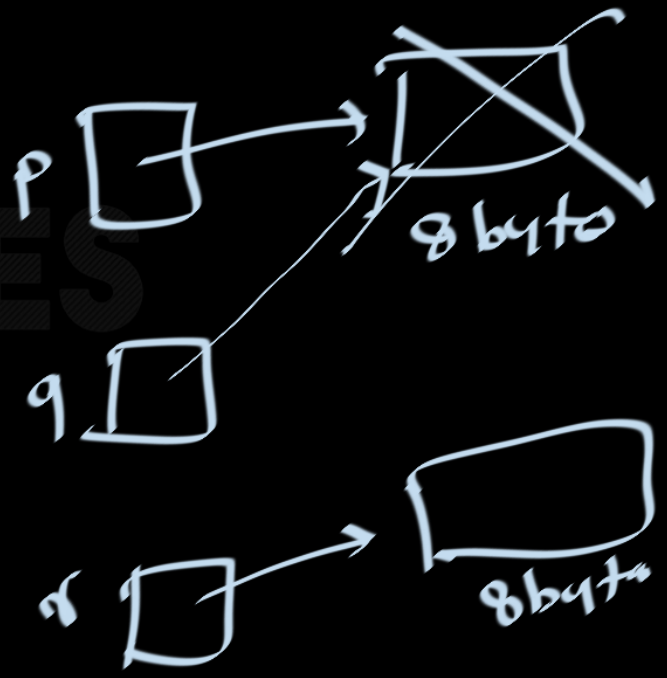


https://people.cs.pitt.edu/~aus/cs449/ts-midterm1-sample-solution.pdf

Question 6

```
int *p, *q, *r;

p = malloc(8);

...
q = p;

...

free(p);
r = malloc(8);

...
*q=5;
```

Dangling Pointer



Question 7

```
int *q;

void foo() {
  int a;
  q = &a;
}

int main() {
  foo();
  /* ... */
  *q = 5;
}
```

Question 8

Consider the C program fragment which uses a function foo.

```
main(){
    int *p = foo();
    p = NULL;
    //some other stuff
}
```

foo can be implemented in various following ways, all pieces of code are identical except for their use of free(). Each of them may be correct or they may have a memory leak, dangling pointer or both.

You should provide an answer for the complete program, which includes combining the main function with a specific implementation of the function foo.

https://ubccsss.org/files/213-2015-mt-soln.pdf

P1:

```
int *copy(int *src) {
    int *dst = malloc(sizeof(int));
    *dst = *src;
    return dst;
}
int foo() {
    int a = 3;
    int *b = copy(&a);
    return * b;
}
```

P2:

```
int *copy(int *src) {
    int *dst = malloc(sizeof(int));
    *dst = *src;
    free (dst);
    return dst;
}
int foo() {
    int a = 3;
    int *b = copy( &a);
    return * b;
}
```

P3:

```
int *copy(int *src) {
    int *dst = malloc(sizeof(int));
    *dst = *src;
    return dst;
}
int foo() {
    int a = 3;
    int *b = copy( &a);
    free (b);
    return * b;
}
```

P4:

```
int *copy(int *src) {
    int *dst = malloc(sizeof(int));
    *dst = *src;
    free (dst);
    return dst;
}
int foo() {
    int a = 3;
    int *b = copy( &a);
    free (b);
    return * b;
}
```

6 (6 marks) **Dynamic Allocation.** The following four pieces of code are identical except for the their use of `free()`. Each of them may be correct or they may have a memory leak, dangling pointer or both. In each case, determine whether these bugs exists and if so, briefly describe the bug(s); do not describe how to fix the bug.

6a

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  return *b;
}
```

Memory leak, because object allocated in `copy` is not freed in the shown code and when `foo` returns it is unreachable.

6b

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  free (dst);
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  return *b;
}
```

Dangling pointer. After `free` in `copy`, `dst` is a dangling pointer. This value is returned by copy and so `b` in `foo` is also a dangling pointer. The last statement of `foo`, `return *b` dereferences this dangling pointer.

6c

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  free (b);
  return *b;
}
```

Dangling pointer. After `free` in `foo`, `b` becomes and dangling pointer and it is then dereferenced in the last statement.

6d

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  free (dst);
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  free (b);
  return *b;
}
```

Dangling pointer. After `free` in `copy`, `dst` becomes a dangling pointer. This value is returned by `copy` and so `b` in `foo` is also a dangling pointer. The third statement of `foo` then calls `free` again on this value, attempting to free an object that has already been freed, which results in an error. If the program where to proceed it would then dereference the dandling pointer in the `return` statement.

Solution

https://ubccsss.org/files/213-2015-mt-soln.pdf

Question 9

2. Given the declarations

```
int* ptrA;
int* ptrB;
```

Which of the following does NOT have memory leak ?

Note: Each code segment denoted by (a), (b), (c) and (d) is independent.

A

```
ptrA = malloc(4);
ptrB = malloc(4);

*ptrA = 345;
ptrB = ptrA;
free(ptrA);
free(ptrB);
```

B

```
ptrA = malloc(4);
ptrA = 345;

ptrB = ptrA;
free(ptrA);
```

C

```
ptrA = malloc(4);
ptrB = malloc(4);


*ptrA = 345;
*ptrB = *ptrA;
free(ptrA);
free(ptrB);
```

D

```
ptrA = malloc(4);
ptrB = malloc(4);

*ptrA = 345;
ptrB = malloc(4);
*ptrB = *ptrA;
free(ptrA);
free(ptrB);
```

Question 10

```c
#include <stdio.h>

void we(void) {
    int *ptr;
    {
        int x;
        ptr = &x;
    }
    *ptr = 3;
}

void main() {
    we();
}
```

Question 11

Which code fragment produces a dangling pointer?

a

```
int* f(void)
{   int d;
    return &d;
}
int main()
{   int* p=f();
    return 0;
}
```

b

```
int* f(void){return  malloc(4); }
int main()
{   int* p=f();
    int  q;
    p=&q;
    return 0;
}
```

c

```
int main()
{   int* p= malloc(4);
    int* q=p;
    free(q);
    return 0;
}
```

d

```
int* f(void){return  malloc(4);  }
int main()
{   int* p=f();
    p= malloc(4);
    return 0;
}
```

Question 12

```
int *p = (int*)malloc(sizeof(int));
int *q = (int*)malloc(sizeof(int));
int *r;


*p = 17;
r = q;
*q = 42;
p = q;
free(r);
```

Question 13

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

int main()
{
    int *p = malloc(sizeof(int));
    *p = 42;
    p = malloc(sizeof(int));
    free(p);
}
```

https://pages.cs.wisc.edu/~remzi/Classes/354/Spring2017/OldExams/midterm-spring-16.pdf

Solution

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

int main()
{
    int *p = malloc(sizeof(int));
    *p = 42;
    p = malloc(sizeof(int));
    free(p);
}
```

ANSWER:

There is a **memory leak** in this program since we overwrite the pointer p before freeing the memory location it points to.

Solution: add `free(p)` before the second malloc.

Question 14

```c
#include <stdio.h>


int main() {
    int *x;
    for (int i = 0; i <= 50; i++) {
        x = (int*)malloc(sizeof(int));
        *x = i;
        printf("%d\n", *x);
    }


    free(x);          // Free the memory allocated for x

    return 0;
}
```

https://www.ecb.torontomu.ca/~courses/coe808/Midterm.pdf

Question 15

Are there any dynamic memory management errors in the following code?

```
int *p = malloc(4);
int *q = malloc(4);
int *r;
*p = 17;
r = q;
*q = 42;
p = q;
free(r);
```

A. No, there are no errors
B. Yes, a memory leak
C. Yes, misuse of a dangling pointer
D. Yes, both a memory leak and misuse of a dangling pointer

https://courses.cs.washington.edu/courses/cse143/00au/exam_quiz/finalsol.pdf

```c
#include<stdio.h>

int *assignval (int *x, int val) {
    *x = val;
    return x;
}

void main () {
    int *x = malloc(sizeof(int));
    if (NULL == x) return;
    x = assignval (x,0);
    if (x) {
        x = (int *)malloc(sizeof(int));
        if (NULL == x) return;
        x = assignval (x,10);
    }
    printf("%d\n", *x);
    free(x);
}
```

GATE 2017

Options for previous question:

The code suffers from which one of the following problems:

A. compiler error as the return of $malloc$ is not typecast appropriately.
B. compiler error because the comparison should be made as $x == NULL$ and not as shown.
C. compiles successfully but execution may result in dangling pointer.
D. compiles successfully but execution may result in memory leak.

[P1]

```
int *g(void)
{
    int x = 10;
    return (&x);
}
```

[P2]

```
int *g(void)
{
    int *px;
    *px = 10;
    return px;
}
```

[P3]

```
int *g(void)
{
    int *px;
    px = (int*) malloc (sizeof(int));
    *px = 10;
    return px;
}
```

GATE 2001

Which one of the functions are likely to cause problems with pointers ?

A. Only $P3$

B. Only $P1$ and $P3$

C. Only $P1$ and $P2$

D. $P1, P2$ and $P3$