# CEng 140

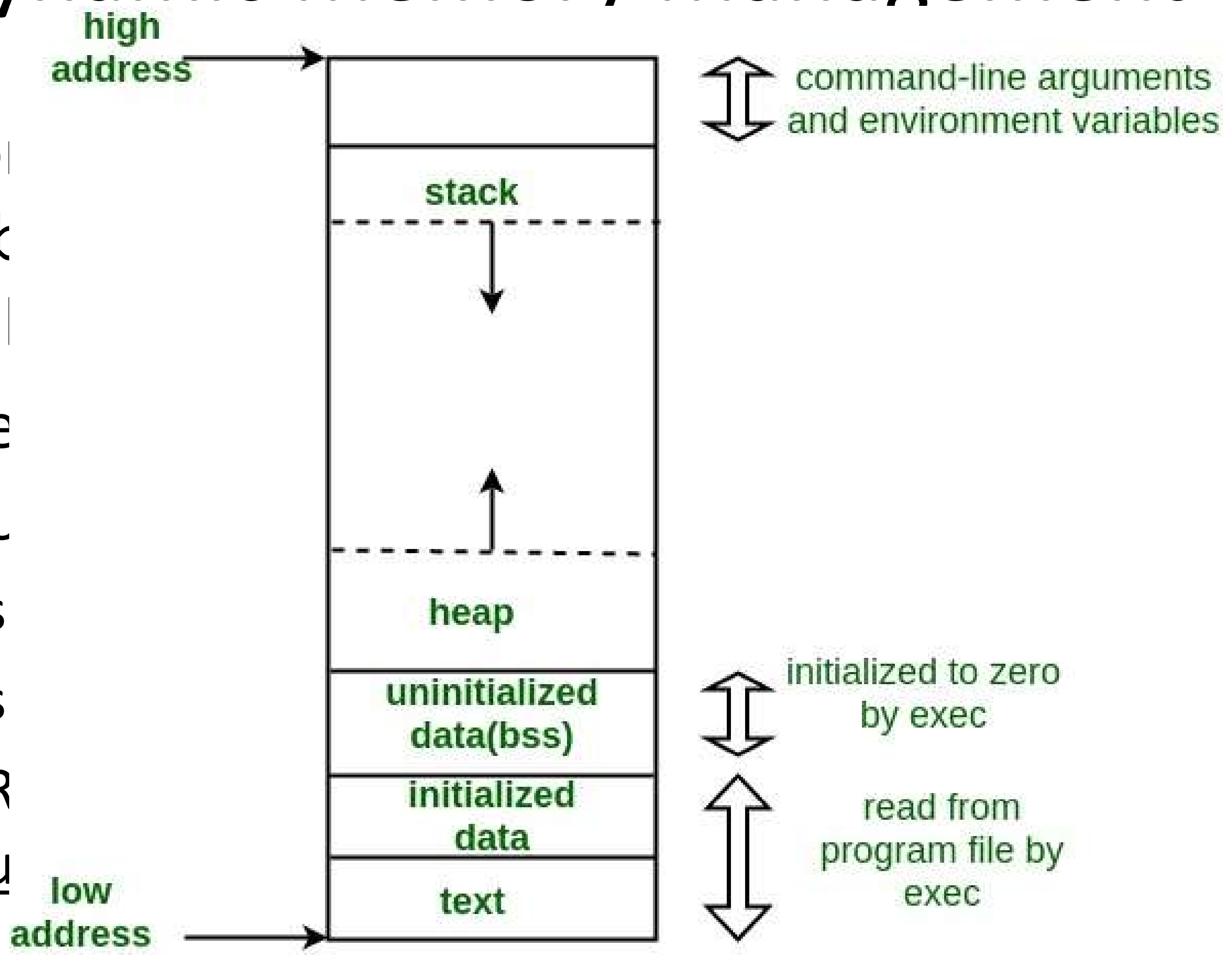
Dynamic Memory Management

# Dynamic Memory Management

- C provides DMM functions that enable storage to be **allocated** as needed & **released** when no longer required!
- sizeof() **operator**
  - Unary
  - sizeof(*type_name)*
  - sizeof(*expression*)
  - Result of sizeof() is of type **size_t** which is an unsigned integral type

# Dynamic Memory Management

- C p
  to
  no
- size
  - U
  - s
  - s
  - R
    u


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

# DMM: sizeof()

- sizeof(*type_name)*
  - When applied to a typename, sizeof() yields the **size in bytes** of an object of the type named
  - sizeof(int) → 2 (assuming your system has 2 byte integers)
  - sizeof(char) → 1
  - sizeof(float) → 4

# DMM: sizeof()

- sizeof(*expression*)
  - When applied to an expression, analyzes the expression at the compile time to determine its type, and yields the same result as if it had been applied to the type of the expression.

short s, *sp;

- sizeof(s) → sizeof(short)
- sizeof(sp) → sizeof(short *)
- sizeof(*sp) → sizeof(short)

# DMM: sizeof()

- If the operand to sizeof is an n-element array of type T, the result is: n x sizeof(T)

  int a[10]; → sizeof(a) → 2 x 10 = 20 bytes

- Size of a string constant is number of chars + 1
  - sizeof("computer") → 9 bytes

# DMM: sizeof()

- sizeof does not cause any of the usual type conversions in determining the type of the expression
  - E.g.: when applied to an array name, sizeof does **not** cause the array name to be converted to a pointer.

However, if the expression contains operators that do perform usual type conversions, these are taken into account while determining the expression's type (see example on the next slide)

# Example

char c;

sizeof(c) → same as sizeof(char)

sizeof(c+0) → same as sizeof(int)

# DMM: sizeof()

- When sizeof is applied to an expression, it is compiled to determine its type, but not compiled to the executable code!

```
int i, j;
i=1;
j= sizeof(--i);
```

What is the value of i afterwards?

# Example

```
char arr[] = "hello";
char *cp = arr;
int main()
{  printf("%lu \n", (unsigned long) sizeof(arr); }
```

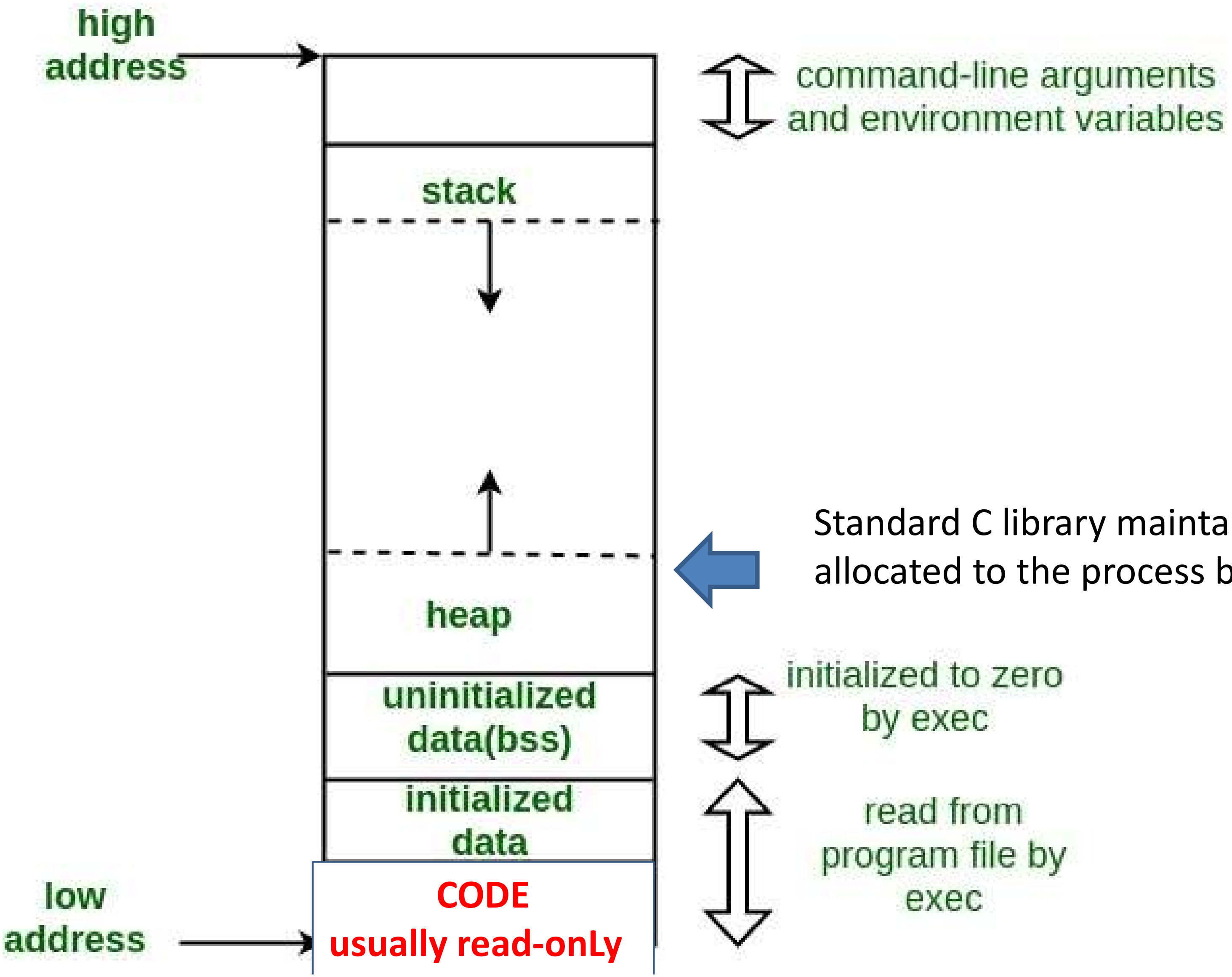
What is the output?

→ 6

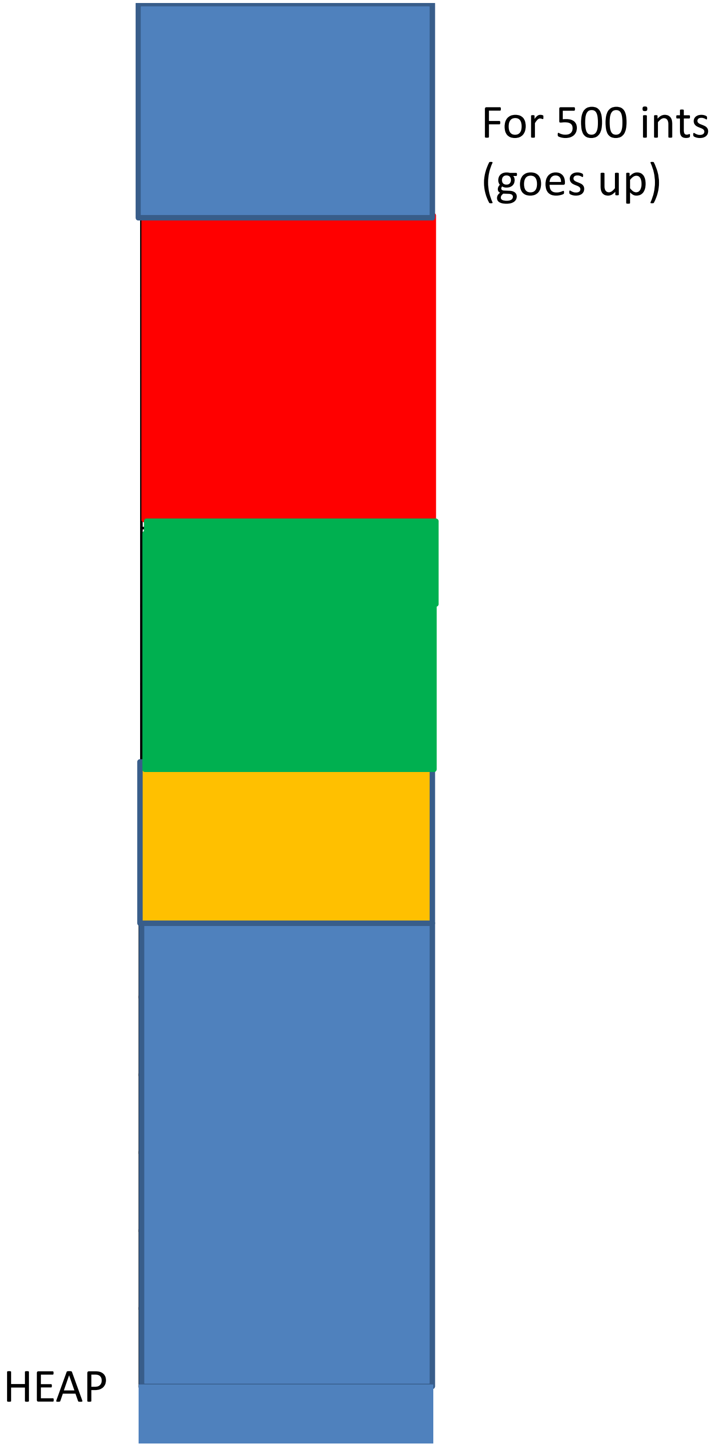
What if we print sizeof(*cp)

→ 1

# DMM


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
low address
CODE
usually read-onLy

Standard C library maintains the HEAP allocated to the process by the OS



- A program may ask for allocating memory (for its data objects) as it needs, and may release the memory when it does not need it anymore

```
int *p, *q;
double *w;
p = /* allocate mem to store 100 ints */
w= /* allocate mem to store 20 doubles */
q = /* allocate mem to store 30 ints */
/* release memoy for doubles */
w= /* allocate mem to store 50 doubles */
/* increase prev memory for 100 int to 120*/
/* reduce prev memory for 30 ints to 5*/
/* increase prev memory for 120 int to 500*/
```

# DMM Functions

- malloc, calloc, realloc, free (in <stdlib.h>)

- malloc, calloc: obtain storage for an object
- realloc: change the size of the storage allocated to an object
- All three functions allocate contiguous memory blocks

- free: releases the storage

# DMM Functions

- Calling malloc, calloc, realloc:
  - yields a pointer to the beginning of the storage allocated to an object, and
  - it is suitably aligned, so that it may be assigned to a pointer of any type of object
  - Returns a generic pointer **void *** that can be safely converted to a pointer of any type.

# void * malloc(size_t size);

- allocates storage for an object whose size is specified by size
- **if** there is available memory space (in HEAP)
  - returns a pointer to the alloctaed storage (which is NOT initialized in any way)
- **else** (not enough space)
  - returns NULL

# Example

```
float *fp, fa[10];
/* allocate storage dynamically to store an array
of 10 floats */
fp = (float *) malloc (sizeof(float)*10);
```

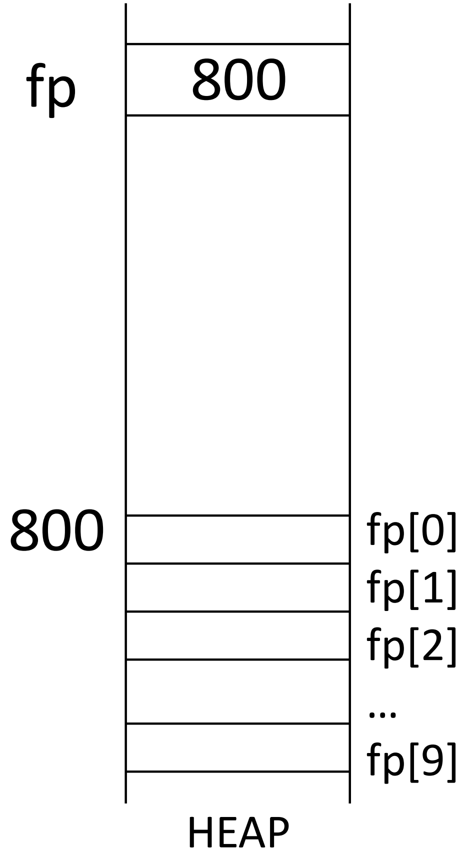
Or:

```
fp = (float *) malloc (sizeof(fa));
```

void * coerced to float *

# Example

```
float *fp;
fp = (float *) malloc (sizeof(float)*10);
```



The area allocated is size of 10 floats

- 10 x 4 bytes (if each float is 4 bytes)

and coerced to float *.

So, *fp is the float at address 800, *(fp+1) is the next float (at address 804) and so on…

**REMARK:** There is no other name of the allocated memory space, we access only via the pointer!

# void * calloc(size_t nobj, size_t size);

- Allocates storage for an array of nobj objects, each of size size
- and, the allocated storage is initialized to zeros!

```
float *fp;
fp = (float *) calloc(10, sizeof(float));
```

# Remark!

- A good practice is checking whether we get the memory or not after calling malloc or calloc!

```
int *ip;
if  ( (ip = (int *) malloc(5 * sizeof(int))) == NULL )
    printf("Error, not enough memory \n");
else /* you allocated, do whatever you want… */
```

# void * realloc(void *p, size_t size);

- Changes the size of the object **pointed to by p** to *size*.
- If succesfull,

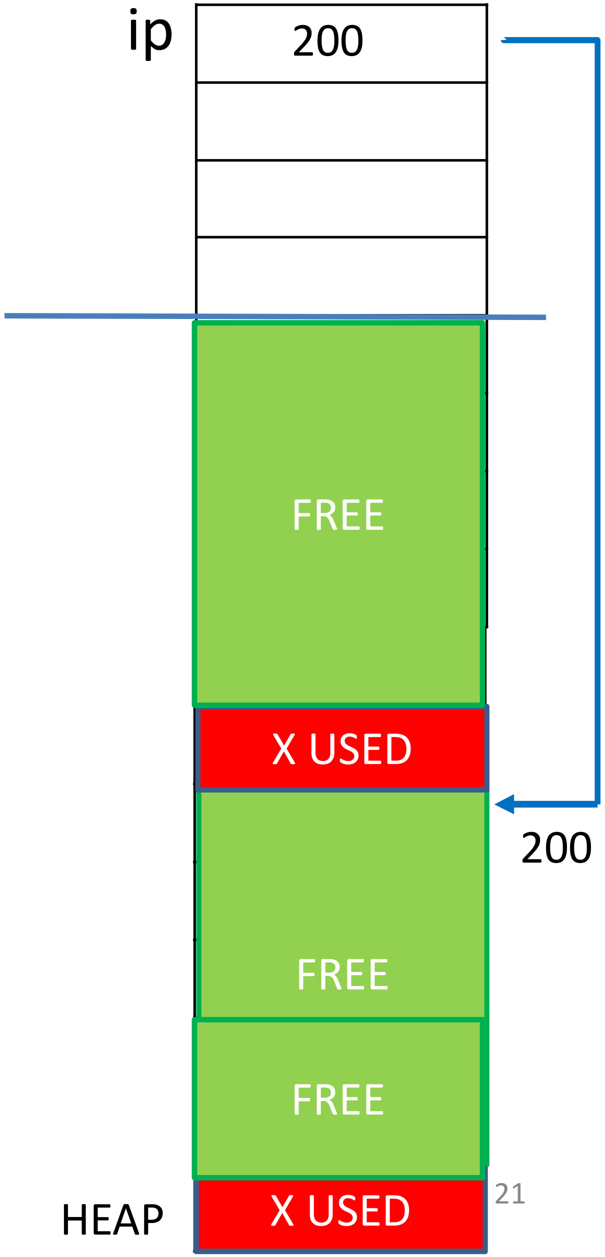
  returns a pointer to the new object,

  else

  returns NULL, and *p remains unchanged

# realloc

```
int *ip;
ip =  (int *) malloc (sizeof(int) * 5);
ip[0]=10; ip[1]=25; ip[2]=31;
ip[3]=24; ip[4]=2;
```



```
ip =  (int *) realloc (ip, sizeof(int) * 3);
```

- If the new size is smaller, original contents up to new size are preserved.

# realloc

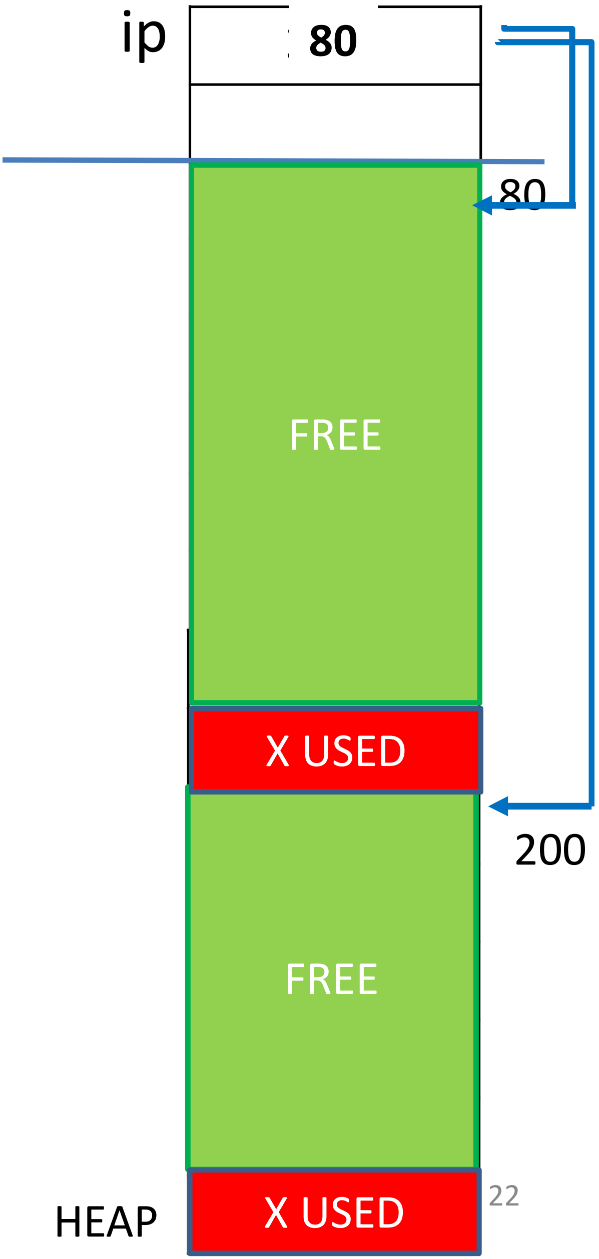
int *ip;

ip = (int *) malloc (sizeof(int) * 5);

ip[0]=10; ip[1]=25; ip[2]=31;

- ip = (int *) realloc (ip, sizeof(int) * **7**);
  - If the new size is **larger**, the original contents are preserved and the remaining space is uninitialized.
  - In particular, if there is additional space after the initially allocated area, allocate it. Otherwise, allocate a new block (of the new size), copy data there, release old block, return the pointer.

**80**



# Pitfalls!

```
int *ip;
ip =  (int *) malloc (sizeof(int) * 5);
ip =  (int *) realloc (ip, sizeof(int) * 10);
```

- if allocated, fine; but what if realloc fails to allocate the required memory?
  - returns NULL, ip = NULL, and the pointer to access the original area (of 5 ints) is lost!!

```
int *tmp;
tmp = (int *) realloc (ip, sizeof(int) * 10);
if (tmp) ip =tmp;
```

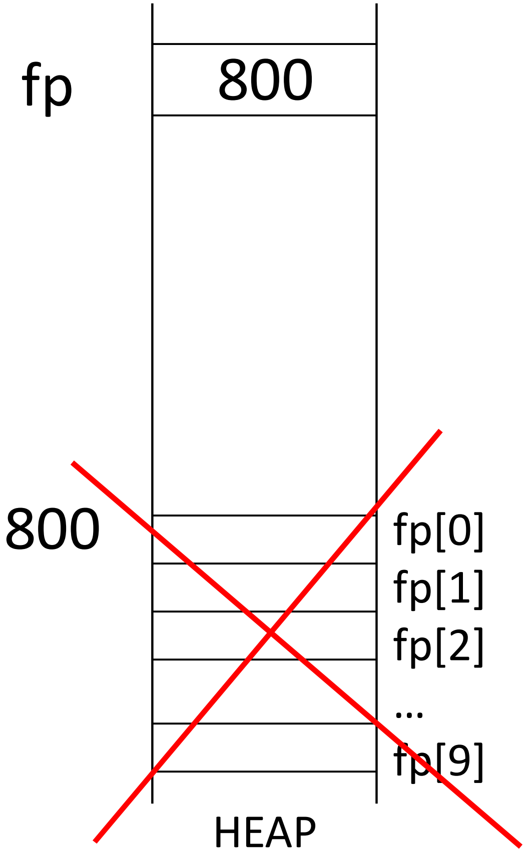
# Pitfalls!

- Similarly, if more than one pointer points to the original place, after realloc, some of them may point to a wrong place!
  - guarantee that all point to the newly allocated area!
- Another common mistake:
- ip = (int *) realloc (ip, sizeof(ip) + sizeof(int) * **5**);
- sizeof(ip) → sizeof(int *)!

# void free(void *p);

- Deallocates the storage pointed by p, where p is previously allocated by malloc/calloc/realloc
  - if p is NULL, does nothing
- Behaviour of free (and realloc) are **undefined**:
  - if p does not match to a pointer returned by a call to malloc/calloc/realloc
  - if the storage has already been deallocated by a call to realloc or free!

```
int main()
{
float *fp;
fp = (float *) malloc (sizeof(float)*10);
/* do whatever you want
    with fp */
free(fp); /* frees the storage for 10
                  floats */
/* do not access *fp anymore */
}
```



# Pitfall

- Memory leak:

```
int *p;
for (i = 0; i< 1000; i++)
      p = (int *) malloc (sizeof(int) * 10000);
```

# Warning