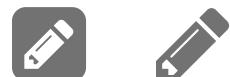


Systems Programming

Lecture 8: Dynamic Memory Management

Stuart James

stuart.a.james@durham.ac.uk



Recap

<https://PollEv.com/stuartjames>



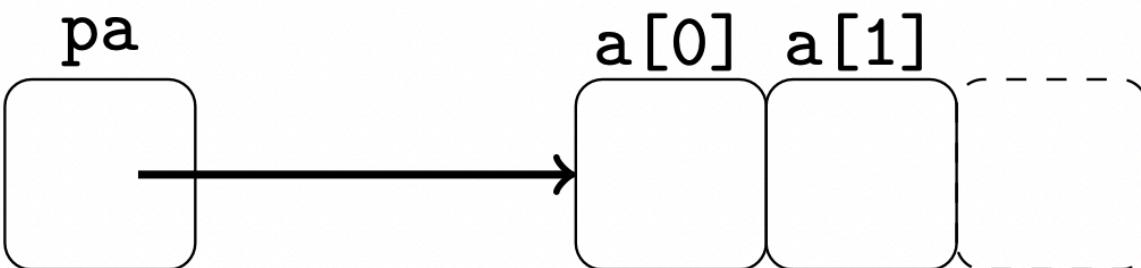
Recap

Last lecture, we learned about Pointers and Pointer Arithmetic

```
int a[10];  
int *pa;
```

These pairs of statements are equivalent using array or pointer notation:

```
pa = &a[0];  
pa = a;
```



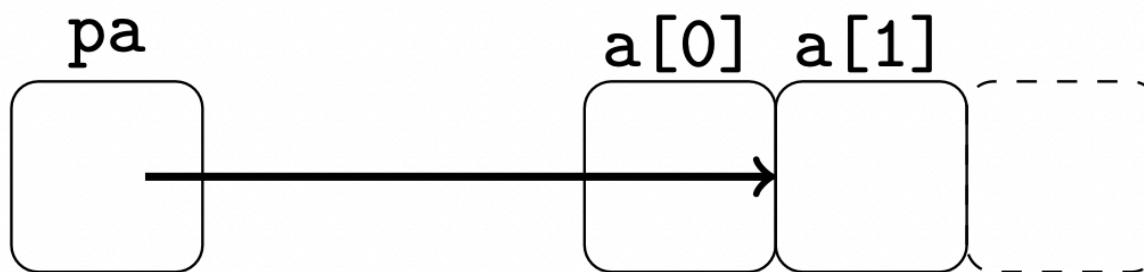
Recap

Last lecture, we learned about **Pointer Arithmetic**

```
int a[10];
int *pa;
```

These pairs of statements are equivalent (+1 translates to +4 bytes (1 int)):

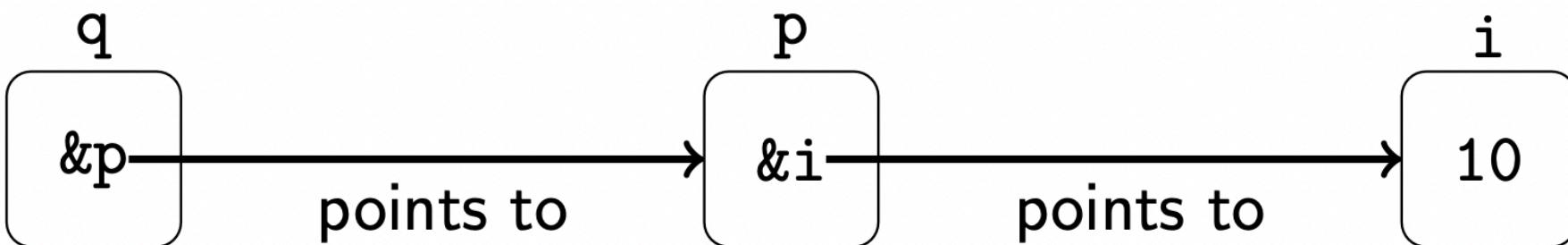
```
pa = &a[1];
pa = (a+1);
```



Recap

We also learned about Pointers to Pointers.

```
int i, *p, **q;  
p = &i;  
q = &p;
```



- Now **i**, ***p** and ****q** all have value 10.



Recap



Recap

The same but subtly different:

```
char str1[] = "Hello";    // array copy
char *str2 = "Hello";    // pointer to literal
char str1[6] =
{'H','e','l','l','o','\0'};
```

- str1 creates a new array on the stack
 - compiler copies "Hello\0" into local storage
 - contents can be modified
- str2 stores the address of a string literal
 - literal lives in read-only program memory (.rodata)
 - contents must not be changed
 - exists for the entire program



and now...

Pointer Safety

- Pointers can cause hard-to-diagnose errors in programs.
- e.g., a function returning pointers to local variables can cause errors when the memory is released back to the runtime system and reused at some time later in the program!



Pointer Safety

- Always set pointers to `NULL` when they are no longer required!
- Always use a simple guard before using pointers, e.g. from `<assert.h>`:

```
assert(ptr != NULL);
```

- There are various tools, e.g., **Valgrind** (<https://www.valgrind.org/>), that can detect many such errors at runtime (at the cost of massive slowdown and greatly increased memory-usage).



Thinking memory

Is this valid?



Thinking memory

Is this valid?

In [1]:

```
1 //args:10
2 //cflags: -std=c90 -pedantic-errors
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main(int argc, char *argv[]) {
8     int i, j;
9
10    int size = atoi(argv[1]);
11    int array[size];
12
13    for (i = 0; i < size; i++) {
14        array[i] = i;
15    }
16
17    for (j = 0; j < size; j++) {
18        printf("%d ", array[j]);
19    }
20    printf("\n");
21
22    return 0;
23}
24
```

/var/folders/4j/xg6vmdqs44z51nqdy93ncv_h0000gn/T/tmp6y6uwof6.c:1:1: error: // comments are not allowed in this language [-Werror,-Wcomment]



Thinking memory, why not Variable Length Arrays?

While variable-length arrays (VLAs) are valid in C99 and later standards, there are several reasons why you might want to avoid using them:

- Portability: Not all compilers support VLAs, especially if they are set to strict C90 or C11 modes where VLAs are optional.
- Stack Overflow Risk: VLAs are allocated on the stack, which has limited space (we will get to soon!).
- Debugging Complexity: Debugging issues related to stack overflow can be more challenging compared to alternatives.
- Standard Compliance: Might not be supported by all compilers without specific flags.

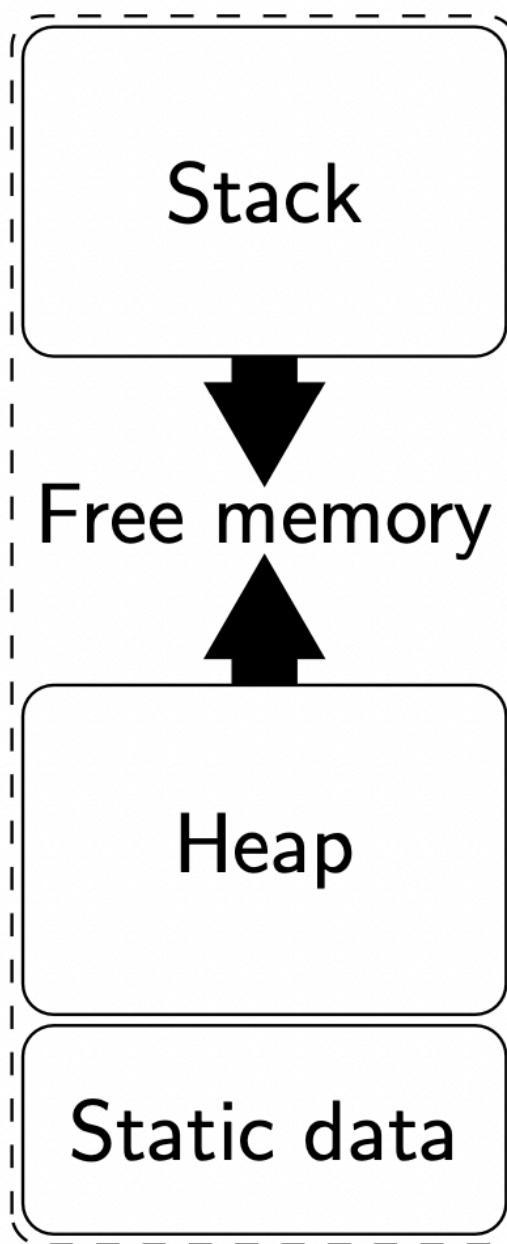


Dynamic Memory Allocation

- Variables and arrays provide fixed size allocation.
- What if the memory needed cannot be pre-determined?
- There is a need to be able to dynamically request variable sized blocks of memory from the runtime system.
- Close integration between C and Unix (and other Operating Systems).
- Requesting memory at run-time.



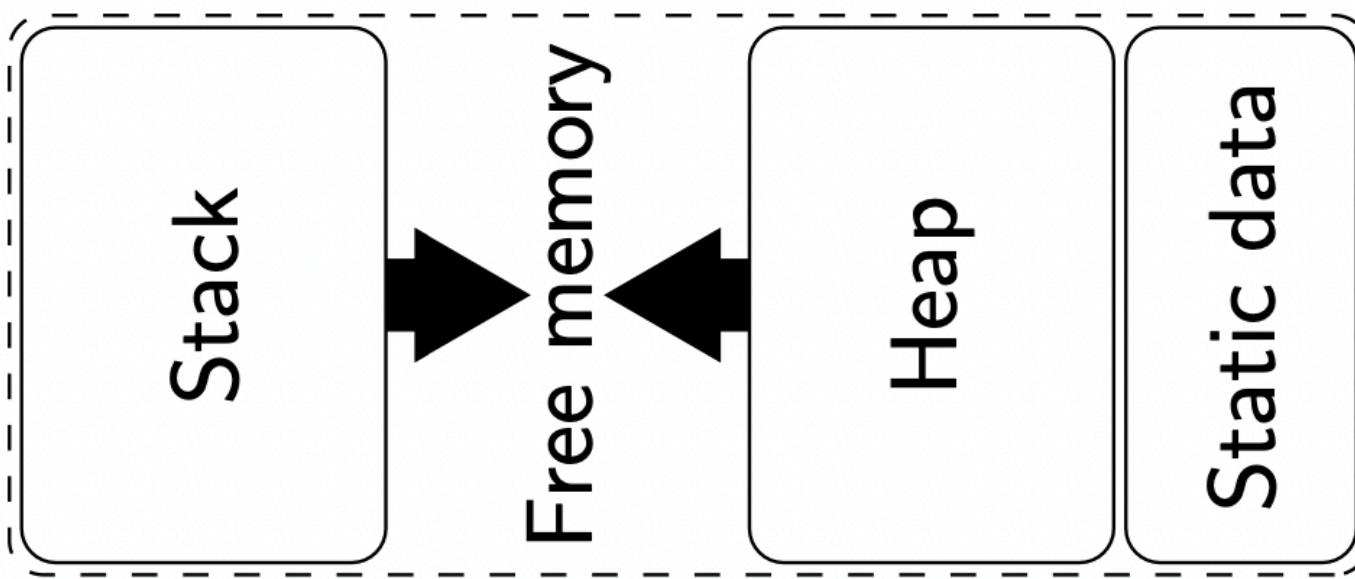
Memory Layout



Memory Layout

Stack:

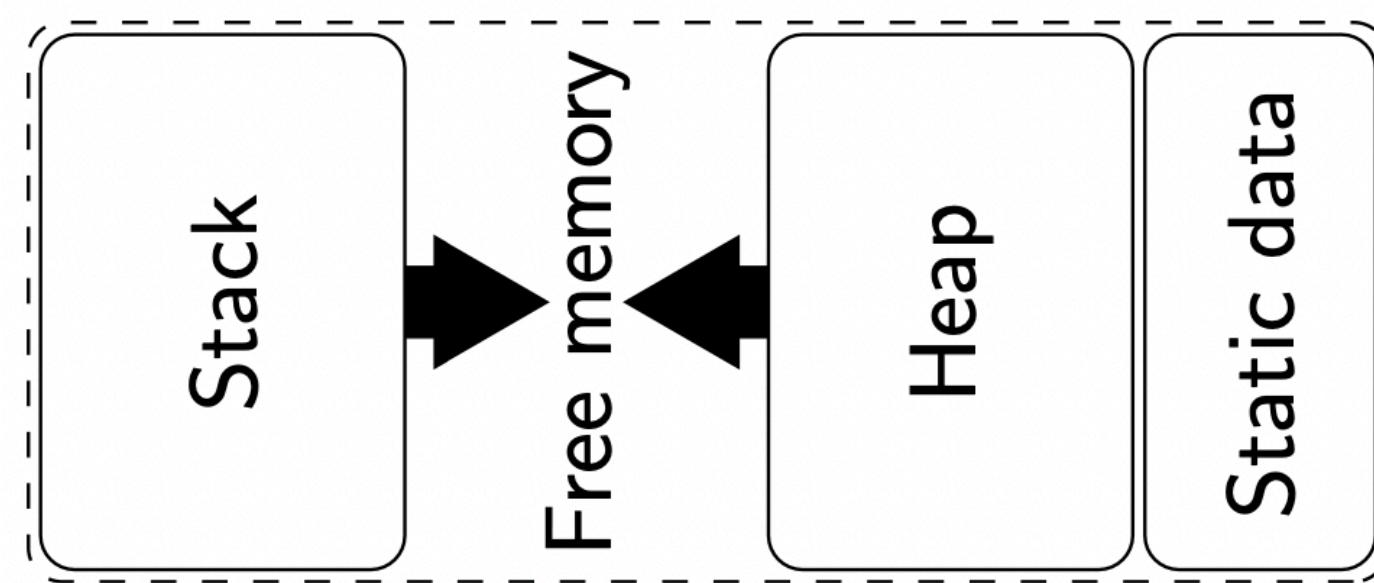
- Stores “temporary” data
- Variables in a function
- Function header
- Small data



Memory Layout

Static Data:

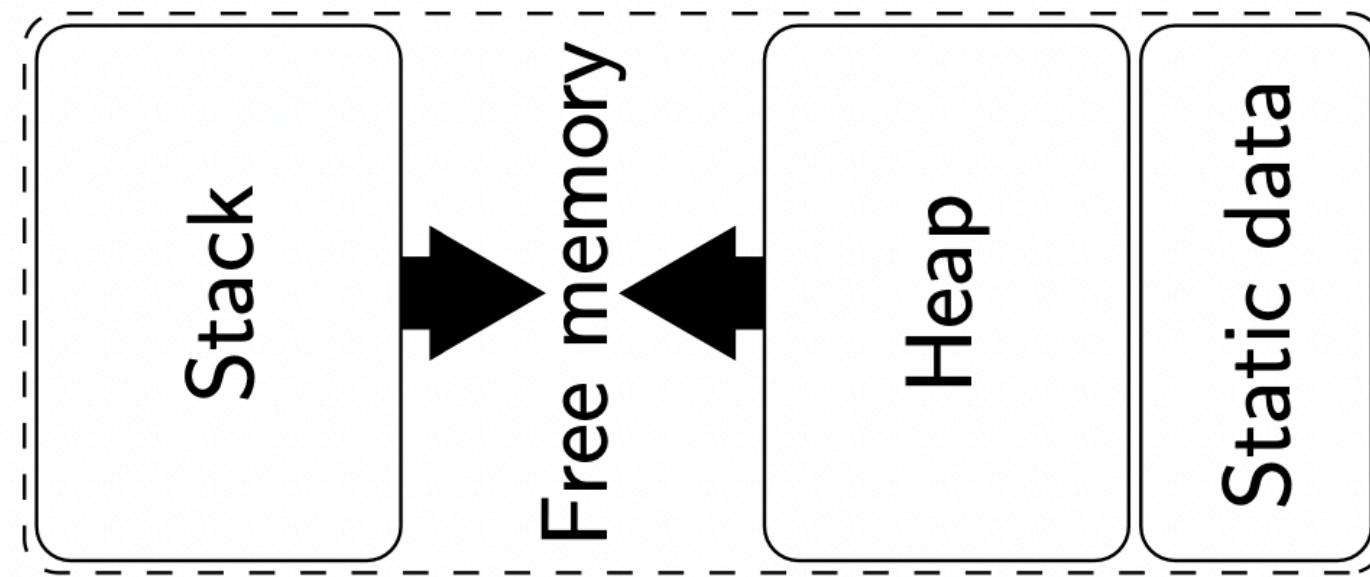
- Data that stays in memory for the duration of the program.



Memory Layout

Heap:

- Used for storing more long-term data.
- YOU, the programmer, control what is in the Heap and when it is released.
- Much more space than the Stack.



Memory Allocation: `malloc()` <stdlib.h>

Function prototype for `malloc()`:

```
void *malloc(size_t size);
```

- Allocates a contiguous block of memory `size` bytes long.
- The return type is `void*`, which is a generic pointer type that can be used with all types.
- `malloc()` returns a `NULL` pointer if it fails to allocate the requested memory.



Memory Allocation: `malloc()` <stdlib.h>

```
void *malloc(size_t size);
```

Test for `NULL` return in case `malloc()` has failed to allocate the memory!

- An Aside:
 - By default, Linux follows an optimistic memory allocation strategy.
 - Even if `malloc()` returns a non-`NULL` value, this is no guarantee that the memory is really available.
 - The operating system actually allocates the memory when you try to use it for the first time; if the system runs out of memory, a process will be killed by the Out Of Memory (OOM) killer.



Example of `malloc()`

```
#define SIZE 41 * sizeof(char)
char *line;
line = malloc(SIZE);
if ( line == NULL ) {
    printf( "Error in malloc() \n" );
    exit(1);}
```

N.B. return value of `malloc()` is automatically cast from `void*` to `char*`

- Pointers pointing to any object are automatically converted to `void*` pointers and vice versa as required.
- Conversions between other sorts of pointers with different types may cause problems due to alignment issues.



Example of `malloc()`

- Creating an array using `malloc()` and printing values.
- The values are random.
- Freeing and allocating multiple times will produce different values.



Example of malloc()

- Creating an array using `malloc()` and printing values.
- The values are random.
- Freeing and allocating multiple times will produce different values.

In [2]:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     // allocate a 20 int array
5     int *a = malloc(20 * sizeof(int));
6     if (a == NULL) exit(1);
7
8     for(int i = 0; i < 20; i++){
9         printf("%d\n",a[i]);
10    }
11    return 0;
12 }
```

```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```



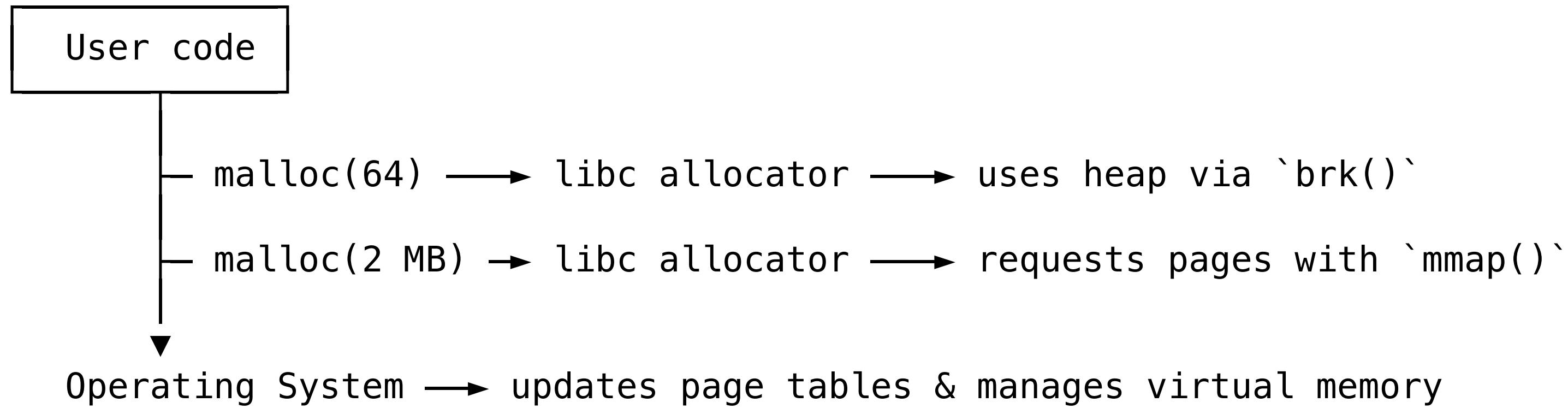
Where Does `malloc()` Get Memory?

```
void *p = malloc(1024);
```

- `malloc()` is part of the C standard library (libc) — not a system call.
- It manages a private memory pool inside your process.
- When the pool runs out of space, `malloc()` asks the kernel for more pages.
- The kernel expands your process's memory map using:
 - `brk()` / `sbrk()` → extend the heap region
 - `mmap()` → create new anonymous memory mappings



How `malloc()`, `brk()`, and `mmap()` Work Together



- Small allocations come from the **heap** (fast, reused space).
- Large allocations use **`mmap()`** (separate regions, page-aligned).
- **`free()`** returns small blocks to libc's internal pool.
- **`free()` + `munmap()`** releases large blocks back to the OS.



Memory De-allocation: `free()` <stdlib.h>

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

- Takes a generic pointer to a block of memory and returns the memory for reuse by `malloc()`.
- If you “forget” about memory you have `malloc()` ed and don’t `free()` it then you have a “memory leak”.
- “Memory leaks” can be very dangerous and difficult to trace [no garbage collection like there is in Python].
 - This can eventually use up all available memory.
 - `free()` has no return value, so even if you pass it a pointer not allocated by `malloc()`, it will process it!



Example of Memory Leak



Example of Memory Leak

In [3]:

```
1 #include<stdlib.h>
2 #include<stdio.h>
3 int main(){
4     int i, *p;
5     for(i=0; i<100000; i++){
6         p = malloc(sizeof(int));
7         if (p == NULL){
8             printf("Could not allocate more memory!");
9             exit(1);
10        }
11        *p = i;
12    }
13    return 0;
14 }
```



Think before you run...

Should you?



Think before you run...

Should you?

```
In [ ]: 1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(void)
4 {
5     char *line = NULL;
6     size_t size = 0;
7     for(;;) {
8         getline(&line, &size, stdin); // new memory implicitly allocated
9         line = NULL;
10    }
11    return 0;
12 }
```



Example of `free()`

```
// allocate some memory
char *line = malloc(SIZE);

// use line in the program

free( line ); // return memory to the O/S
line = NULL; // set pointer to NULL
```

- Errors from continuing to use a pointer after the memory has been released can be very hard to detect.
- N.B. `line` is implicitly cast to a `void*` pointer.



Dangers of **free()**



Dangers of `free()`

In [12]:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 void maybe_free(char *p, int do_free) {
6     if (do_free) free(p);    // frees conditionally
7 }
8
9 int main(int argc, char **argv) {
10     char *buf = malloc(64);
11     if (!buf) return 1;
12
13     /* use buf... */
14     snprintf(buf, 64, "hello");
15
16     maybe_free(buf, argc > 1);
17     /* ...more work... */
18     printf("Length = %zu\n", strlen(buf));
19     free(buf);
20     return 0;
21 }
```

Length = 5



Memory Allocation: `calloc()` <stdlib.h>

Function prototype for `calloc()`:

```
void *calloc( size_t n, size_t size );
```

- Allocates a contiguous block of memory of `n` elements each of `size` bytes long, initialises all bits to 0 (Security).
- Useful to ensure old data is not reused inappropriately (i.e. prevents leakage of stale data — important for secure systems programming.)
- The return type is `void*` - generic pointer type used for all types.
- `calloc()` returns a `NULL` pointer if it fails to allocate memory.
 - Always test for `NULL` return!



calloc() Example



calloc() Example

In [4]:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     //allocate an array of 20 ints
5     int *a = calloc(20, sizeof(int));
6     if (a == NULL) exit(1);
7     int i;
8     for(i = 0; i < 20; i++){
9         printf("%d\n",a[i]); // 0 because calloc zeros out allocated memory
10    }
11    return 0;
12 }
```

```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```



Memory Allocation: **realloc()** <stdlib.h>

Function prototype for **realloc()**:

```
void *realloc( void *ptr, size_t size );
```

- Allows a dynamic **change** in size of a previously allocated block of memory pointed to by **ptr**.
 - **ptr** must point to memory previously allocated by **malloc()**, **calloc()** or **realloc()**.



Memory Allocation: `realloc()` <stdlib.h>

Function prototype for `realloc()`:

```
void *realloc( void *ptr, size_t size );
```

- `realloc()` moves and copies contents if it needs to, freeing original block, which means `ptr` might change.
- `realloc()` returns a `NULL` pointer if it fails. Check for this!



realloc() Example

Simple program that takes integers typed in by the user and stores them in an array.

- Each time the array becomes full, it is dynamically increased in size to hold more numbers
- Contains a key function `getline2()`, which reads the integers from the command line



```
int getline2(char line[], int max) {
    int nch = 0;
    int c;
    max = max - 1; /* leave room for '\0' */
    while((c = getchar()) != 'q') {
        if(c == '\n')
            break;

        if(nch < max) {
            line[nch] = c;
            nch = nch + 1;
        }
    }
    if(c == 'q' && nch == 0)
        return 'q';

    line[nch] = '\0';
    return nch; }
```



realloc() Example

getline2()

- Uses `getchar()` to read in characters as they are typed.
- Runs in a loop until a 'q' or a newline is encountered.
- Reads in the characters typed by the user one by one and stores them in the array `line`.
- When the character `\n` is pressed, the function returns, via use of the `break` statement to exit a loop.
- No checking performed to see if the input is an integer.



```
ip = malloc(array_size * sizeof(int));
while( getline2(line, MAXLINE) != 'q' ) {
    if(nitems >= array_size ) /* increase allocation */
        int *newp;
        array_size += INCREASE ;
        newp = realloc(ip, array_size * sizeof(int));

        printf("<< Expanding by %d to size %d >>\n", INCREASE, array_size );
        if(newp == NULL) {
            printf("out of memory\n");
            exit(1);
        }
        ip = newp;
    }
    ip[nitems++] = atoi(line);
}
```



realloc() Example

main()

- Uses `getline2()` to read in a line of text.
- Creates an array to store current line of text, `line`.
- Creates a second array to store the integers entered: `ip`.
- As soon as `ip` is full, `realloc()` is called to resize the array



atoi() <stdlib.h>

```
int atoi(const char *s);
```

- Converts a string pointed to by `s` to an integer.
- Also see `atof()`, `atol()` and `atoll()` (since C99) equivalents
- To convert from an integer to a string use :

```
int sprintf( char *s, char *format, <value list> );
```

- Where the value list is the variables used in the format string.
- `sprintf(str, "Sum of %d and %d is %d", a, b, a+b);`



The `->` Operator

`->` gives us a shorthand accessing members of structures using a pointer.

```
struct point {  
    int x;  
    int y;  
} pt, *ptr;  
  
ptr=&pt;
```

We can now modify `pt.x` in three ways:

```
pt.x=3; // Access directly  
(*ptr).x=3; // Access by dereferencing a pointer  
ptr->x=3; // Access using the -> operator
```



Summary

- Memory Allocation
- Memory Layout
- `malloc()`
- `free()`
- `calloc()`
- `realloc()`

