# Introduction

Let’s look at the memory allocation interfaces found in UNIX systems. This section is short, sweet, and should help us answer the following questions?

- **What types of interfaces are often used to allocate and manage memory?**
- **What mistakes should be avoided?**

An empty `.c` program and terminal will be provided for you throughout this section to practice as you learn.

# Types of Memory

A C program allocates two types of memory, **stack memory** and **heap memory**.

## Stack Memory

**Stack memory**, and allocations and deallocations of it are handled implicitly by the compiler for you, the programmer.

In C, declaring memory on the stack is simple. Let's say you need some space in a function `func()` for an integer named `x`. Your code might look something like this:

```
void func() {
  int x; // declares an integer on the stack
  ...
}
```

The compiler handles the rest, ensuring that `func()` has enough stack space when you call into it. The compiler deallocates the memory for you when you return from the function, so don't leave data on the stack if you want it to survive the call.

This need for long-lived memory brings us to the second type of memory, **heap memory**.

## Heap Memory

In **heap memory**, all allocations and deallocations are handled explicitly by you, the programmer. Here's how to allocate an integer on the **heap**:

```
void func() {
  int *x = (int *)malloc(sizeof(int));
  ...
}
```

This snippet allocates both stack and heap:

1. When the compiler sees your declaration of an integer pointer (`int *x`), it knows to make place for it.

2. A call to `malloc()` seeks space for an integer on the heap. This address is returned (on success or `NULL` on failure) and saved on the stack to be used by the program.

Heap memory is more challenging to both users and systems because of its explicit nature and it’s more varied usage. Most of our focus moving forward will be on heap memory.

## 1. Fill in the blanks to complete the statements below.

## 2. Rearrange the blocks below to create a function that allocates memory on the stack for a variable `pinky`

## 3. Rearrange the blocks below to create a function that allocates memory on the heap for a variable `brain`

## malloc()

**The `malloc()` function takes a size parameter in bytes qand returns:**

- A reference to the freshly allocated space, or
- NULL if it fails.

Type `man malloc` at the command line to see how to use `malloc()`. You should see something similar to the following:

```
#include <stdlib.h>
 ...
 void *malloc(size_t size);
```

To use `malloc()`, you only need to include the `<stdlib.h>` header file. The C library has the code for`malloc()`. Adding the header allows the compiler to check whether you are calling `malloc()` correctly.

To use `malloc()`, pass it a single parameter of type `size_t`.

Most programmers do not directly type a number (like 10) here because it's considered bad form.

Instead, macros and procedures are used. You can do something like this for a double-precision floating point value:

```
double *d = (double *) malloc(sizeof(double));
```

This example uses the `sizeof()` operator to request the right amount of space.

## Passing Variables to malloc()

You can also use `sizeof()` with a variable's name (not just a type), but this may not always work.

Consider the following code snippet:

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

We've specified space for an array with 10 integers in the first line. However, `sizeof()` produces a small value, like 4 (on 32-bit platforms) or 8 (on 64-bit machines).

This is because `sizeof()` thinks we're just asking for an integer pointer, not how much memory we've dynamically allocated.

Sometimes, `sizeof()` does work exactly as expected:

```
int x[10];
printf("%d\n", sizeof(x));
```

In this instance, there's enough information to tell the compiler that 40 bytes have been allocated.

### `malloc()` tips

- Using `sizeof()` instead of `malloc()` may lead to trouble when trying to declare space for a character in a **string**.
  - Use `malloc(strlen(s) + 1)`, which takes the string's length and adds 1 to it to accommodate the end-of-string character.
- `malloc()` returns a void pointer. This is C's way of returning an address to the programmer.
- The programmer can also help by using a **cast**. In our `double` example, the programmer casts `malloc()`'s return type to a reference to a double for additional reassurance.

## free()

Allocating memory is the easy part. Deciding when, how, and even if to free memory is the difficult part.

**`free()` is used by programmers to free heap memory that is no longer in use:**

```
 int *x = malloc(10 * sizeof(int));
...
free(x);
```

The routine takes one input, a `malloc()` pointer. Because of this, the user can’t specify the size of the allocated area, and the memory-allocation library has to track this itself.

## Common Errors: Intro

**We'll go over some common `malloc()` and `free()` errors.**

While compiling a C program is necessary to build a correct C program, it is not enough. As a result, many modern languages support **automatic memory management**.

Instead of having to execute something like `free()` to free memory, a **garbage collector** runs and figures out what memory you no longer have references to and frees it for you.

## Forgetting to Allocate Memory

**Many routines need memory allocation before calling them.**

For example, the routine `strcpy(dst, src)` copies a string from one pointer to another. If you're not careful, you might do something like:

```
char *src = "hello";
char *dst;        // oops! unallocated
strcpy(dst, src); // segfault and die
```

When you run this code, you will almost certainly get a **segmentation fault**. The correct code might look like this:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

You may also use `strdup()` to make your life even easier. For further details, see the `strdup` man page.

## Not Allocating Enough Memory

A **buffer overflow** occurs when not enough memory is allocated.

A common error is to make *almost* enough room for the destination buffer.

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

Depending on how `malloc()` is implemented and a variety of other factors, this program can look like it works perfectly.

When the string copy is executed, it may write one byte too much over the end of the allotted space in some cases. In other cases, this is harmless, maybe overwriting a variable that is no longer used.

In other cases, the `malloc` library allocates a bit of additional space anyway, so your application doesn't scribble on the value of another variable and runs perfectly fine.

The  program may also just fail and crash.

## Forgetting to Initialize Allocated Memory

With this mistake, you correctly run `malloc()` but fail to fill your freshly allocated data type.

If you forget, your program will encounter an **uninitialized read**, where it reads unknown data from the heap.  If you're lucky, the program still works (e.g., zero). If you're unlucky, something harmful could happen.

# Forgetting to Free Memory

## Forgetting to Free Memory

A **memory leak** is another typical issue that **occurs when you forget to free memory**.

This is a big problem with long-running applications or systems (like the operating system itself), as steadily leaking memory eventually leads to a running out of memory. When this happens, a restart is required.

So, when you are through with a piece of memory, **make sure you release it**.

Adopting a garbage-collected language doesn't help in this case: if you still have a reference to some region of memory, no garbage collector will ever free it, so memory leaks persist even in more recent languages.

**In some instances, it may appear that not using `free()` is a reasonable choice.**

For example, if your software is short-lived and will exit soon, when the process is finished, the OS will clear up all of its allocated pages, and no memory leak will occur.

While this *"works"*, in the long run, one of your goals as a programmer should be to build healthy habits. One of these is understanding how you manage memory and (in languages like C) freeing the blocks you have allocated.

Even if you can get away with it, it's a good idea to get into the habit of **freeing every byte you explicitly allocate**.

# Other Mistakes Using free()

## Freeing Memory Before You’re Done With It

A **dangling pointer** happens when a program releases memory before it has finished using it. For example, calling `free()`, then calling `malloc()` to allocate something else, which recycles the incorrectly-freed memory. This can cause the program to crash or overwrite good memory.

## Freeing Memory Repeatedly

A **double free** is when a program frees memory twice. The memory-allocation library may become confused and crash.

## Calling free() Incorrectly

The incorrect use of `free()` is our last issue. `free()` expects you only to pass to it one of the pointers you received from `malloc()` earlier. Using a different value can (and does) cause issues.

# Underlying OS Support: brk/sbrk

`malloc()` and `free()` are library calls, not system calls. The `malloc` library handles virtual address space, but it is **built on top of system calls** that ask the OS for more memory or release it.

## `brk` / `sbrk`

`brk` and `sbrk` are two basic memory management system calls in Unix and Unix-like operating systems. These system calls are used to control how much memory is allocated to the process's data segment. Generally, these functions are called from a higher-level memory management library function like malloc.

- `brk` is used to change the program's **break**: the heap's end.
    - It takes one argument (the new break's address) and either increases or decreases the heap's size dependent on the new break's size.
- `sbrk` takes an increment but essentially does the same thing as `brk`.

Let's take a look at an example in the code to the left, `sbrk.c`.

- The brk() system call sets the program break to the location specified by end_data_segment. Virtual memory is allocated in units of pages, so end_data_segment is effectively rounded up to the next page boundary.

Copy and paste the following code to compile this example.

```
gcc sbrk.c -o sbrk
```

And run it.

```
./sbrk
```

Your output should look similar to the output below.

```
My page size: 4096
program break address: 0x565446a6a000
sizeof char: 1
c1: 0x565446a6a001
program break address: 0x565446a6a001
```

warning

## Warning

You should never explicitly call `brk` or `sbrk`. They are used by the memory-allocation library and attempting to use them will probably result in disaster. Instead use `malloc()` and `free()`.

## Underlying OS Support: mmap

The `mmap()` method can also get memory from the operating system.

When successful, `mmap()` returns a pointer to the mapped area.

Let’s look at a quick example in the code to the left, `mmap.c`.

Copy and paste the following code to compile this example.

```
gcc mmap.c -o mmap
```

And run it.

```
./mmap
```

Here we allocate memory to an array using `mmap` and we want to print it.

- PROT_READ | PROT_WRITE gives us protection for reading and writing to the mapped region.
- MAP_PRIVATE is used because the mapping region is not shared with other processes.
- MAP_ANONYMOUS is used because here, we have not mapped any file.

In virtual memory, `mmap()` can establish an anonymous memory region within your program by giving in the appropriate inputs. This memory can then be controlled as a heap.

# calloc()

A few other calls are also supported by the memory-allocation library.

## `Calloc()`

`calloc()` functions are passed a number of elements and a single element size. `calloc` returns a void pointer to allocated memory, for example, `ptr = (int*) calloc (i,sizeof(int))` where `i` is the number of int type elements.

- `calloc()` allocates and zeros memory before returning.
    - This eliminates issues such as assuming memory is zeroed and failing to populate it.

Take the code to the left, `calloc.c` as an example.

Copy and paste the following code to compile this example.

```
gcc calloc.c -o calloc
```

And run it.

```
./calloc
```

Your output should look like the output below.

```
By defaul, all values are: 0 0 0 0 0
After entered values:0 1 2 3 4
```

## realloc()

`realloc()` can be useful when you've allocated space for something (for example, an array) and then need to add to it. It creates a larger memory region, copies the old region into it, and returns the pointer to the new region.

Re-allocates memory allocated by the `malloc()` and `calloc()` methods that has not been freed with a new size.

The `realloc()` function copies the content of the old memory pointed by ptr to the new memory and then deallocates the old memory internally.

Let's look at the code to the left, `realloc.c`, as an example.

```
gcc realloc.c -o realloc
```

And run it.

```
./realloc
```

Your output should look like the following.

```
5
1
2
3
4
```