Memory Management

CISC 3350 — CUNY Brooklyn College
Lecture Notes

Memory Management: Motivation

Memory is among the most basic yet essential resources of processes. **Memory management** includes responsibilities such as allocation, manipulation, and release of memory slots.

The **address space** of a process is all the memory slots that the process has allocated (dynamically or statically,) and the memory space that the OS gave to that process to store its variables in. The process, therefore, has legal access to these memory locations.

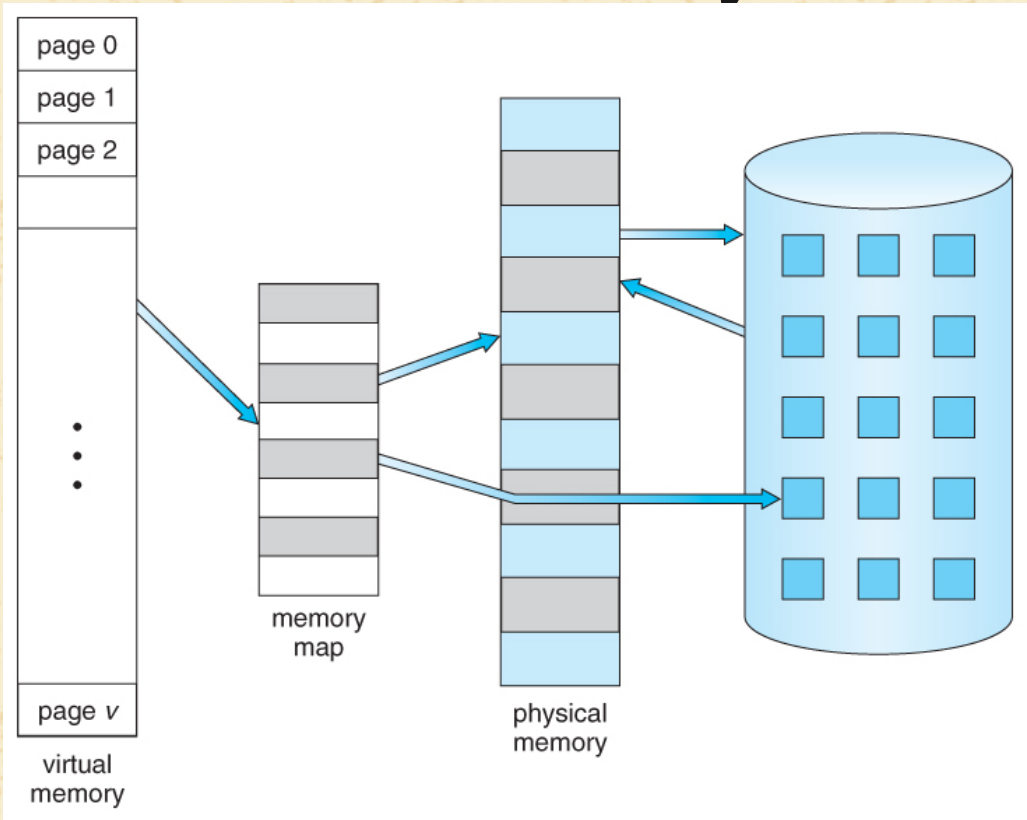
Processes don't directly access **physical addresses**, which are the addresses of memory slots on the RAM chip: each process sees its own address space as the 'entire universe', without even being aware of the existence of other memory slots, and the addresses it uses (called **logical addresses** or **virtual addresses**) are different from physical memory addresses, so the **memory management unit (MMU)**, a software part in the OS, needs to covert each logical address to a physical address. Both physical and virtual address spaces are linear and flat, just like a simple array is.

Memory sizes, from smallest to largest, are: bits → bytes → words → pages. Every byte in RAM has its own address.

Pages and Paging

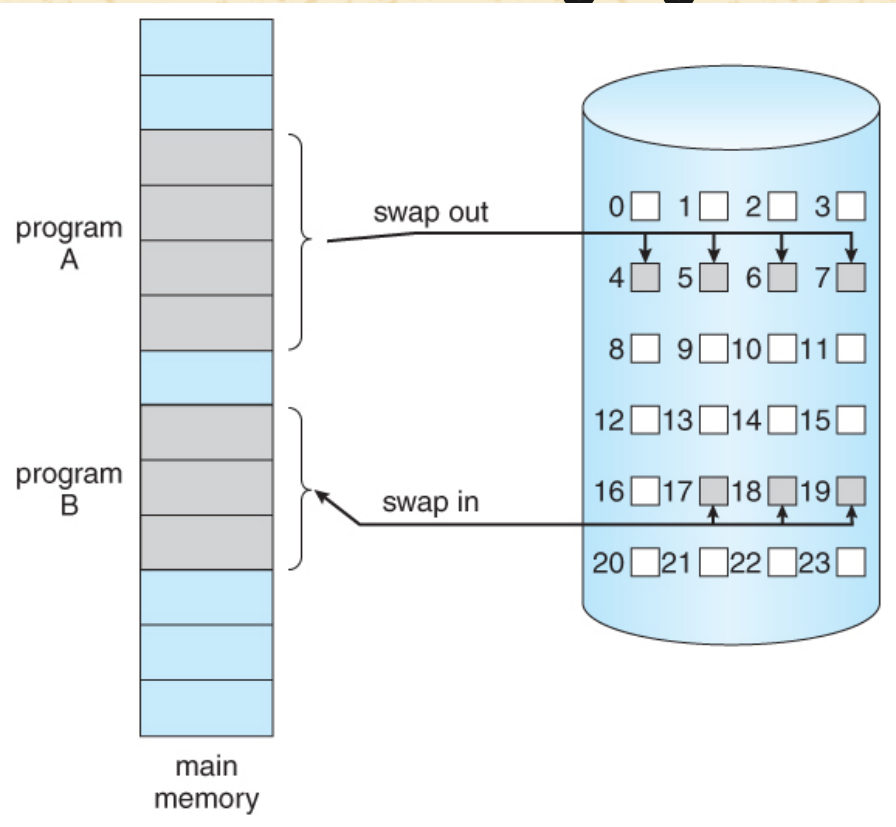
- The virtual address space of a process is divided into **pages**, which are contiguous groups of bytes.
- Machine architecture determines the page size of that computer. Typical sizes include 4KB on 32-bit systems and 8KB on 64-bit systems. (32-bit and 64-bit tell the length of a memory address: 32 vs. 64 bits long.)
- A page is called **valid** if the data (values of variables and objects, the program's instructions, etc.) of this page is now located somewhere in physical RAM.
- A page is called **invalid** if its data is located on disk instead of in main memory. ***How could this happen?*** Sometimes, when there's not enough space in RAM to store data, the OS keeps some of the process's data on disk. This memory management technique is called **virtual memory**.
- A **page fault** happens when the process needs to use data that is located on an invalid page. That is, when the data is located on disk and not in main memory. The kernel will **page-in** (= copy) the data from the disk into RAM. The location in RAM that stores a page is called a **frame**.
- **Paging-out** is the process of moving data from RAM to secondary storage (to make room for new pages). Usually, such data are unlikely to be used soon, which is why we decide to move them to disk instead of moving other, more crucial data.

Virtual Memory



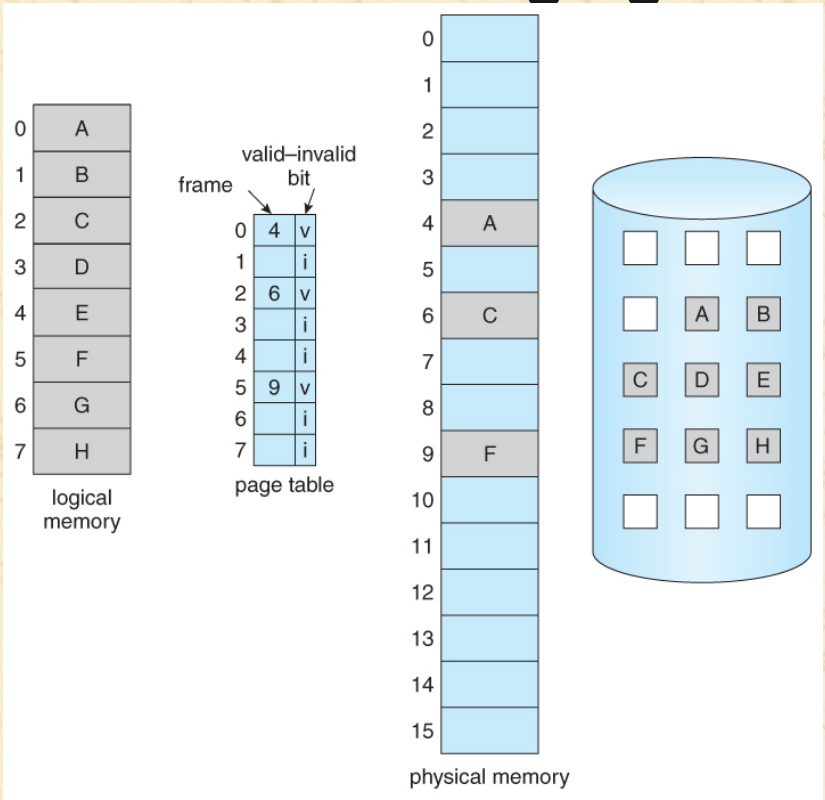
General layout of virtual memory. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Demand Paging



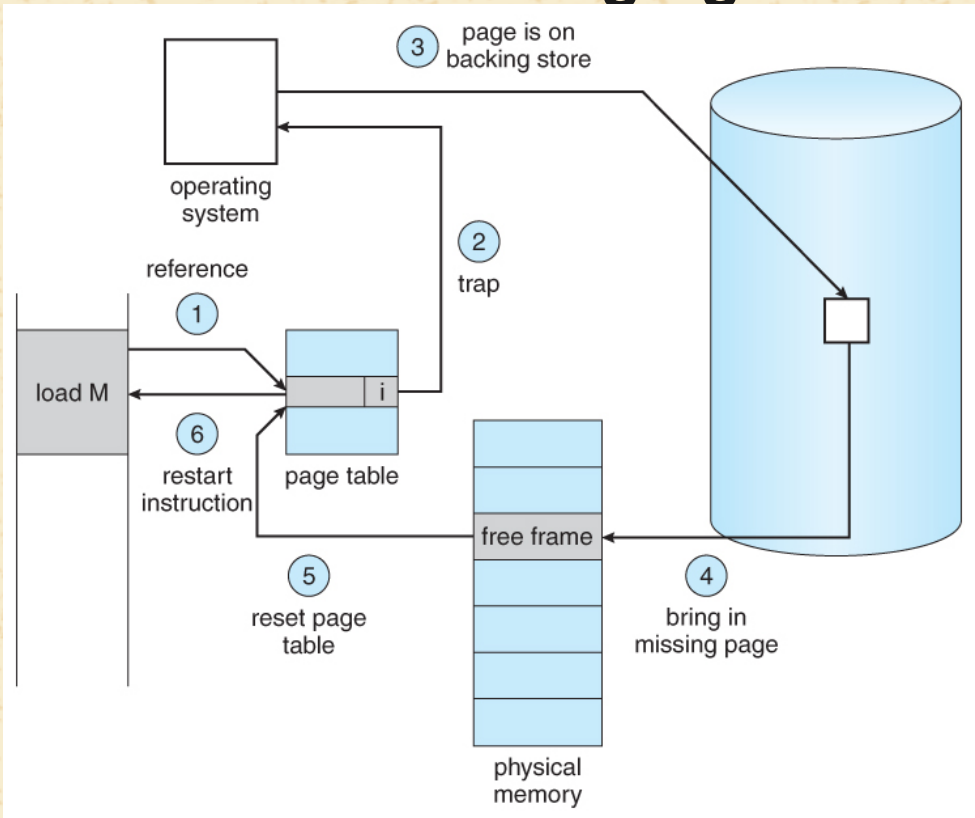
Swapping pages in and out. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Demand Paging



Some pages are out of memory. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Demand Paging



Steps in handling a page fault. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Page Replacement

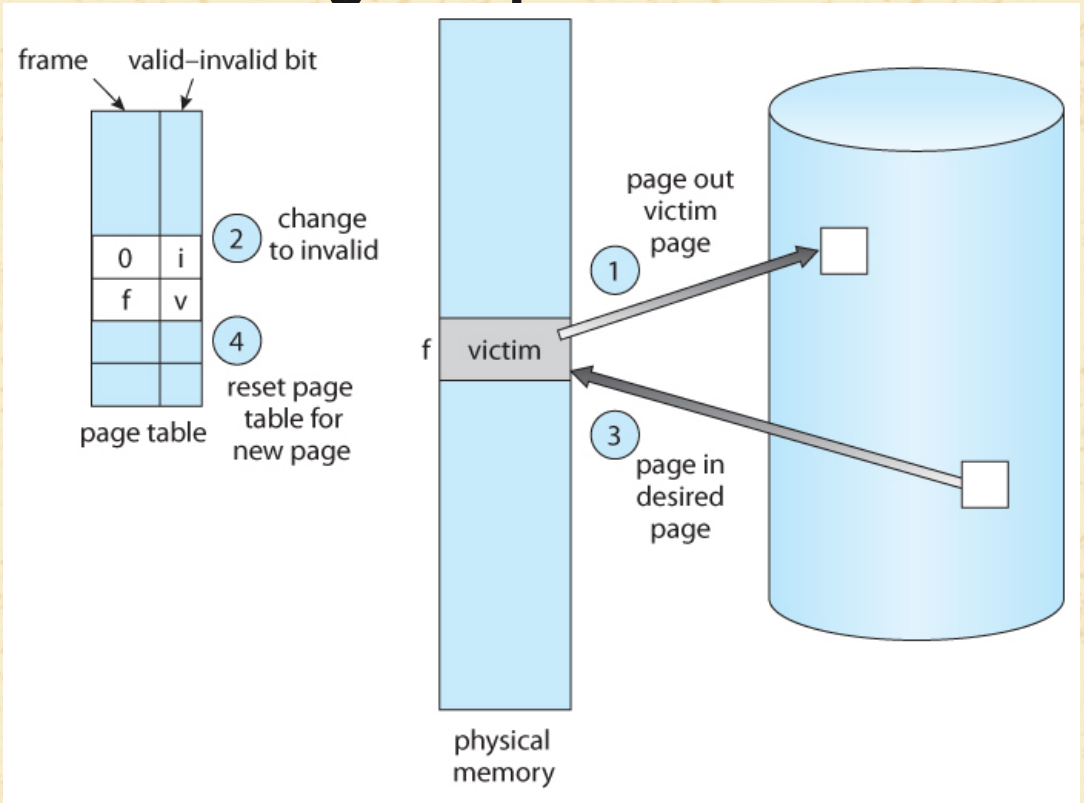


Illustration of page replacement. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

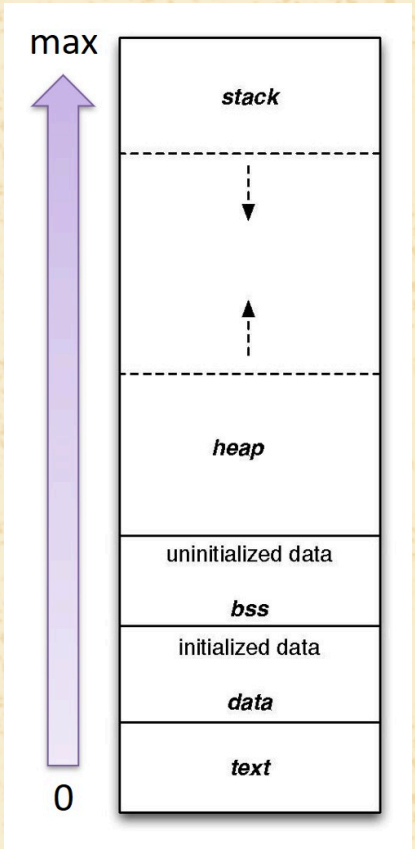
Any questions so far?

Memory Regions of a Process

Each process's memory address space consists of the following sections:

- The **text** section, in which the binary code of the program (the process's instructions) and read-only data (constants) are stored. This section is fixed in size (depending on how long the binary code is.)
- The **data** section, in which global and static variables that were initialized are stored. It has a fixed size, too. (***How do we know that it's fixed in size?***)
- The **bss** (= **b**lock **s**tarted by **s**ymbol) section, in which global and static variables that *weren't yet* initialized are stored. It has a fixed size, too.
- The **stack** section, in which local variables, such as those defined inside functions, are stored, and
- The **heap** section, in which variables that were dynamically allocated are stored. Example: when you use the `new` operator in Java or the `malloc()` function in C, your variable (or object) is stored on the heap.

Memory Regions of a Process



Memory sections of a process. Based on Dougct, CC BY-SA 3.0, via Wikipedia.

Memory Regions of a Process

Any questions so far?

Allocating Dynamic Memory

<table>
<tr><td>Name: <code>malloc()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: allocates memory of <code>size</code> bytes on the heap and returns a pointer to the start of that allocated memory.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <stdlib.h> // Defines malloc().</code></td></tr>
<tr><td colspan="3">Syntax: <code>void * malloc (size_t size);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>size</code>: the size of memory you want to allocate (in bytes)</td></tr>
<tr><td rowspan="3">What type it returns:
<code>void*</code></td><td colspan="2">On success: A non-<code>NULL</code> pointer to the start of the newly-allocated memory. Note that the allocated memory's content is junk values (not zeros.)</td></tr>
<tr><td colspan="2">On failure: <code>NULL</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:

```
struct Cat *fifi = malloc (sizeof (struct Cat)); // Note: in C, you don't need to
// typecast the returned pointer from void* to the correct type (it's automatic.)
// In C++, however, you must cast it explicitly.
if (fifi == NULL)
    perror ("malloc");
```

</td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

Allocating and Zero-Initializing Arrays

Name: `calloc()`	**System call or function:** Function	**Links:** Online Manual | Course Packet

What it does: allocates memory into `nr` elements of `size` bytes each. It is allocated on the heap, and the function returns a pointer to the start of that allocated memory.

What libraries you must include:
```
#include <stdlib.h> // Defines calloc().
```

Syntax:
```
void * calloc (size_t nr, size_t size); // The 'c' in 'calloc' stands for 'contiguous'.
```

Description of arguments: `nr`: the number of elements memory for which you want to allocate
`size`: the size of each such element you want to allocate (in bytes)

What type it returns: `void*`	**On success:** A non-`NULL` pointer to the start of the newly-allocated memory. Unlike `malloc()`, `calloc()` sets all the bytes to zero (`0`s for numbers, and `NULL`s for chars and pointer types.)
	On failure: `NULL`
	If failure, does it set `errno`? Yes

Quick example:
```
double *temperatures = calloc (31, sizeof (double)); // Temperatures over a month.
if (temperatures == NULL)
    perror ("calloc");
```

Other variations: N/A

Resizing Allocated Memory

<table>
<tr><td colspan="2">Name: <code>realloc()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: resizes the memory allocation starting at <code>ptr</code> (which was previously allocated with either <code>malloc()</code> or <code>calloc()</code>) to the new, smaller or larger, size <code>size</code> (in bytes).</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdlib.h> // Defines realloc().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void * realloc (void * ptr, size_t size);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>ptr</code>: the pointer to an already-allocated memory space. Passing <code>NULL</code> will make <code>realloc()</code> behave just like <code>malloc()</code>: allocate new memory.
<code>size</code>: the desired new size in bytes of the allocated memory.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>void*</code></td><td colspan="2">On success: A non-<code>NULL</code> pointer to the start of the resized memory. It may or may not be at the same physical place in main memory as the previous allocation (depending on available space.)</td></tr>
<tr><td colspan="2">On failure: <code>NULL</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<pre>if ((temperatures = realloc (temperatures, 365 * sizeof (double))) == NULL)
    perror ("realloc");</pre></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

An Example of Expanding Allocated Memory


```c
// Below, space for 5 'Cat' elements is allocated:

struct Cat *cats;
cats = calloc (5, sizeof (struct Cat));

if (cat == NULL) {
    perror ("calloc");
    exit (EXIT_FAILURE);
}

// After the above code runs, we can now
// proceed to use cats[0], ..., cats[4].
```

```c
// We want to add 5 more 'Cat's:

cats = realloc (cats, 10 * sizeof (struct Cat));

if (cats == NULL)
{
    perror ("realloc");
    exit (EXIT_FAILURE);
}

// Proceed to use the cats array as planned.

free (cats); // Free the cats at the end.
```

Freeing Dynamically Allocated Memory

<table>
<tr><td colspan="2">Name: <code>free()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: frees (releases) dynamically allocated memory starting at the pointer <code>ptr</code>, which was returned by <code>malloc()</code>, <code>calloc()</code>, or <code>realloc()</code>. Not calling <code>free()</code> on allocated memory will lead to a memory leak (= tons of allocated but unused space.) On the other hand, a pointer whose memory was freed becomes a dangling pointer and should not be accessed until new memory is allocated. Calling <code>free()</code> on an already-freed memory will result in undefined behavior.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdlib.h> // Defines free().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void free (void * ptr);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>ptr</code>: a pointer to the beginning of allocated memory: it mustn't be a pointer to some midway spot in that memory. If you pass <code>NULL</code>, <code>free()</code> will return right away, without doing anything.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>void</code></td><td colspan="2">On success: – This function doesn't return a value –</td></tr>
<tr><td colspan="2">On failure: – This function doesn't return a value –</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>free (temperatures);</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Freeing Dynamically Allocated Memory

Any questions so far?

Manipulating Already-Allocated Memory

<table>
<tr><td colspan="2">Name: <code>memset()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: sets each of <code>n</code> bytes starting at memory location <code>s</code> (so, from <code>s</code> to <code>s + n - 1</code>) to the character <code>c</code> (e.g.: for <code>c = 'A'</code>, the memory starting at <code>s</code> will include <code>"AAAA..."</code>).</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <string.h> // Defines memset().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void * memset (void *s, int c, size_t n);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>s</code>: a pointer to some allocated memory (doesn't need to be the start.)
<code>c</code>: the character you want to set as the content of each of the bytes from from <code>s</code> to <code>s + n - 1</code>.
<code>n</code>: the number of characters you want to set to <code>c</code>.</td></tr>
<tr><td rowspan="3">What type it returns:
<code>void*</code></td><td colspan="3">On success: It returns its 1st argument: the pointer <code>s</code>.</td></tr>
<tr><td colspan="3">On failure: If <code>s</code> is an invalid pointer, or if <code>s + n - 1</code> is beyond the end of the allocated memory, the behavior of <code>memset()</code> is undefined.</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>memset (arr, '\0', 10); // Setting 10 chars starting at 'arr' to the null character.</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Manipulating Already-Allocated Memory

<table>
<tr><td>Name:
<code>memcmp()</code></td><td colspan="2">System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: compares the first <code>n</code> bytes of <code>s1</code> to the first <code>n</code> bytes of <code>s2</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <string.h> // Defines memcmp().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int memcmp (const void *s1, const void *s2, size_t n);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>s1</code>: a pointer to some allocated memory (doesn't need to be the start.)
<code>s2</code>: another pointer to some (other) allocated memory (doesn't need to be the start.)
<code>n</code>: the number of characters from <code>s1</code> and <code>s2</code> that you want to compare.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: If the <code>n</code>-character-long substring of <code>s1</code> alphabetically preceeds this of <code>s2</code>, it returns a negative integer. If the substring of <code>s2</code> preceeds this of <code>s1</code>, a positive integer is returned. Otherwise, if both substrings are equal, <code>0</code> is returned.</td></tr>
<tr><td colspan="3">On failure: If the comparison goes beyond an allocation, the program might crash.</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example:

```
if (memcmp (arr_1, arr_2, 10) == 0)
    printf ("The first 10 bytes of both memory locations are equal!\n");
```

</td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Manipulating Already-Allocated Memory

<table>
<tr><td colspan="2">Name: <code>memmove()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: copies the first <code>n</code> bytes of <code>src</code> to <code>dst</code>. No problem occurs if some of the bytes of <code>src</code> and <code>dst</code> overlap.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <string.h> // Defines memmove().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void * memmove (void *dst, const void *src, size_t n);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>dst</code>: the memory location to which we want to copy <code>src</code>'s first <code>n</code> bytes.
<code>src</code>: the source memory location from which we want to copy bytes.
<code>n</code>: the number of byes from <code>src</code> that we want to copy to <code>dst</code>.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>void*</code></td><td colspan="2">On success: It returns the pointer <code>dst</code>.</td></tr>
<tr><td colspan="2">On failure: If the copying action exceeds the memory allocation, the program might crash.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>memmove (destination, source, 10);</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Manipulating Already-Allocated Memory

<table>
<tr><td colspan="2">Name: <code>memcpy()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: copies the first <code>n</code> bytes of <code>src</code> to <code>dst</code>. It's faster than <code>memmove</code>, but <code>src</code> and <code>dst</code> mustn't overlap; otherwise, the app might crash.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <string.h> // Defines memcpy().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void * memcpy (void *dst, const void *src, size_t n);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>dst</code>: the memory location to which we want to copy <code>src</code>'s first <code>n</code> bytes.
<code>src</code>: the source memory location from which we want to copy bytes.
<code>n</code>: the number of byes from <code>src</code> that we want to copy to <code>dst</code>.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns:
<code>void*</code></td><td colspan="2">On success: It returns the pointer <code>dst</code>.</td></tr>
<tr><td colspan="2">On failure: If the copying action exceeds the memory allocation, or if the source and destination locations overlap, the program might crash.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>memcpy (destination, source, 10);</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Manipulating Already-Allocated Memory

<table>
<tr><td colspan="2">Name: <code>memchr()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: searches the first <code>n</code> bytes of <code>s</code> for the character <code>c</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <string.h> // Defines memchr().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void * memchr (const void *s, int c, size_t n);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>s</code>: a pointer to the memory in which we want to search for <code>c</code>.
<code>c</code>: the character we want to find.
<code>n</code>: the number of byes from <code>s</code> that we search.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>void*</code></td><td colspan="2">On success: It returns the pointer to the first instance of the character <code>c</code> in <code>s</code>.</td></tr>
<tr><td colspan="2">On failure: <code>NULL</code> (if <code>c</code> isn't found.)</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>char * str = memchr (source, 'a', 20);</code>
<code>// If 'source' is "I ate an apple\0", then 'str' will point to: "ate an apple\0".</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

memory_functions.c: Full C Program


```c
/* A program demonstrating uses of some C memory
 *    functions: malloc(), calloc(), realloc(),
 *    free(), memset(), memcpy(), and memcmp().
 *
 *    Miriam Briskman, 5/14/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

// For perror() and printf():
#include <stdio.h>
// For malloc(), calloc(), realloc(), and free():
#include <stdlib.h>
// For memset(), memcpy(), memcmp(), strcmp(),
```

memory_functions.c: Full C Program

Any questions so far?

Memory Management: Sources

Topic 11: Memory Management. Briskman, Miriam. *CISC 3350 OER Course Packet.* 2024, CC BY-NC 4.0.
URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/CISC_3350_OER_Course_Packet.pdf#topic+11:+memory+management.

Partly based on the teaching of Professor **Yongqing Xiang** of Brooklyn College.