Dynamic Memory Management

Lesson 1.2

Learning Objectives

LO 1.2.1 **Utilize** void* in memory referencing

LO 1.2.2 **Use** the sizeof operator to acquire the memory size allocated by a specific reference

LO 1.2.3 **Avoid** memory leaks and dangling pointers when managing dynamic memory

LO 1.2.4 **Assert** valid use of free function in deallocating dynamic memory

Revisiting scanf()

- Prototype:

```
   int scanf(char* str, void*, void*, …);
```

- What is **void***?
  - void* is similar to Object in Java
  - it can point at any address
- Since the data types being passed into scanf can be anything, we need to use void* pointers
- If you want to scan a value into a local variable, you need to pass the address of that variable
  - this is the reason for the ampersand (&) in front of the variable

Dereferencing

- Pointers work because they deal with **addresses** – not value
  - an operator performs an action at the value indicated by the pointer
  - the value in the pointer is an address

- We can find the value of any variable by dereferencing it
  - simply put an **ampersand (&)** in front of the variable and you now have the address of the variable

Function **`sizeof()`**

- The **`sizeof()`** function is used to determine the size of any data type
  - — prototype: **`size_t sizeof(data_type);`**
  - — returns how many bytes the data type needs
    - for example:

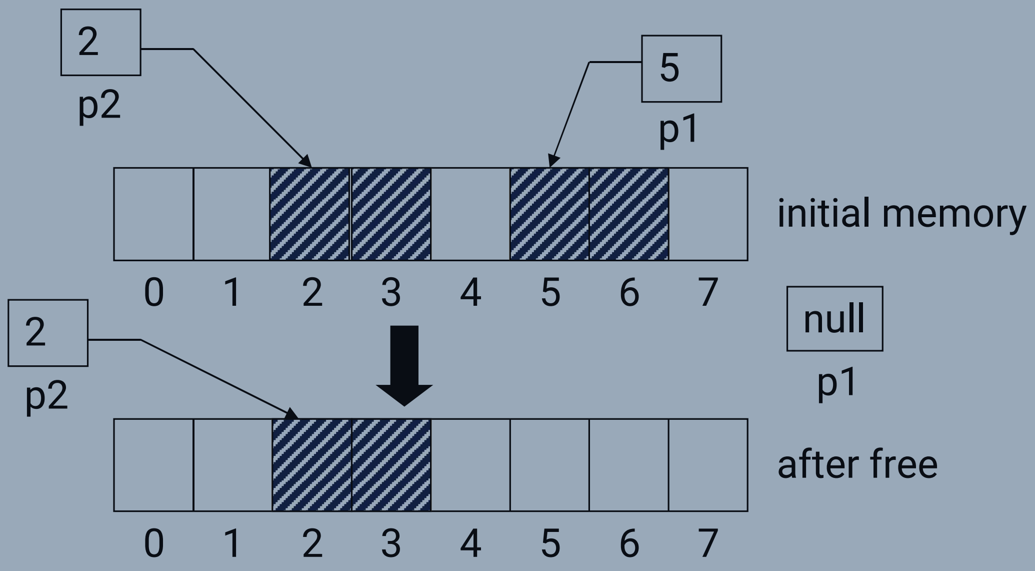
```
sizeof(int) = 4  //this will depend on the
sizeof(char) = 1 // computer you are using
```

  - — works for standard data types and user defined data types (**`struct`**s)

Freeing Memory

- Prototype: **`void free(void* ptr);`**
  - — releases the area pointed to by ptr
  - — ptr **must not be null** (trying to free the same area twice will generate an error)
- Example:

```
free(p1);
p1 = null;
```



Key

▨ allocated memory

□ free memory

Common Mistakes in Using Pointers

- Forgetting to free space on the heap (memory leak)

```
int *p1 = malloc(sizeof(int));
int *p2 = malloc(sizeof(int));
p1 = p2;  // making p1 point to p2 is fine,
          // but now you can’t free the space
          // originally allocated to p1
```

Strengthening the Learning Objectives

LO 1.2.1 Utilize void* in memory referencing

Example:

```
#include <stdio.h>

int main() {
   char data1 = ‘X’;
   int data2 = 100;
   float data3 = 3.14f;
}
```

Print the values stored in `data1`, `data2`, and `data3` using only 1 variable reference for all print statements.

LO 1.2.2 Use the sizeof operator to acquire the memory size allocated by a specific reference

Example:

Allocate memory for an integer, a character, and a double value referenced to a, b, and c, respectively.

Display the size of the memory referenced by `a`, `b`, and `c`.

LO 1.2.3 Avoid memory leaks and dangling pointers when managing dynamic memory

Example:

```
#include <stdio.h>
int main() {
   int* data1 = (int*) malloc(sizeof(int));
   int* data2 = (int*) malloc(sizeof(int));
   int* data3 = (int*) malloc(sizeof(int));
   *data1 = 3; *data2 = 5; *data3 = 10;
}
```

Create a variable, `ave`, that references a floating-point value. Compute the average of `data1`, `data2`, and `data3` and place the value where `ave` is referencing. Deallocate the memory referenced by `data1`, `data2`, and `data3` so they cannot be used thereafter.

LO 1.2.4 Assert valid use of free function in deallocating dynamic memory

Example:

```
#include <stdio.h>
int main() {
   int* data1 = (int*) malloc(sizeof(int));
   int* data2 = (int*) malloc(sizeof(int));
   //TODO: Average and merge data1 and data2 values
}
```

Compute the average of the values pointed by `data1` and `data2` (*write the code on the `//TODO` comment*) but before the main program ends, only 1 integer memory allocation should remain, referenced by `data1` (t*he average of the two values*); no more, no less.