Interlude: Memory API

malloc, free.

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Types of Memory

- **There are two types of memory that are allocated**
  - 1) Stack and 2) Heap

- **Stack memory is allocated and deallocated implicitly by compiler**
  - It is sometimes called automatic memory
  - If you declare **`int x;`** compiler does the rest, making sure to make space on the stack, deallocating memory, etc

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

- **Heap memory is allocated and deallocated explicitly by user**
  - It is a long-lived memory
  - A heavy responsibility, no doubt
  - Certainly the causes of many bugs
  - Compiler allocates stack memory when see **`int *x`**
  - **`malloc()`** returns the address of heap space (or **`NULL`** if can't be allocated)
  - The address is stored at **`x`** located at the stack

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

The `malloc()` Call

- **The `malloc()` call is quite simple**
  - You pass it a size asking for some room on the heap
  - If succeeded, a pointer to the newly-allocated space is given, otherwise **`NULL`**
  - The single parameter **`malloc()`** takes is of type **`size_t`**, which describes how many bytes you need; but usually **`sizeof()`** operator is used as follow:

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

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

  - **`sizeof()`** is a compile-time operator (not function, which is called at run-time), meaning that the actual size is known at compile time

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

⇒ 4

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

⇒ 40

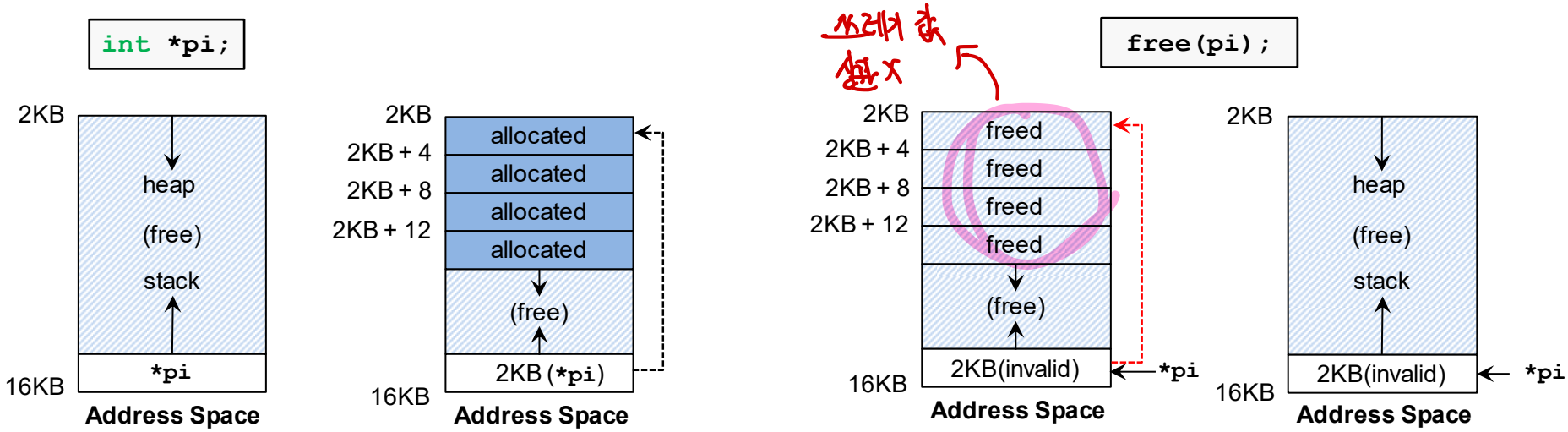
  - The **`malloc()`** call returns a pointer to type **`void`** and let the programmer decide what to do with it; by using cast

The `free()` Call

- **To free heap that is no long in use, you simply call `free()`**
  - One argument, a pointer returned by `malloc()`
  - The size of the allocated region must be tracked by the memory-allocation library itself

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

- **Visualizations of memory allocation and deallocation**




```
pi = (int *)malloc(sizeof(int)* 4);
```

Courtesy of Prof. Youjib Won @ KAIST

Common Errors: Forgetting To Allocate Memory

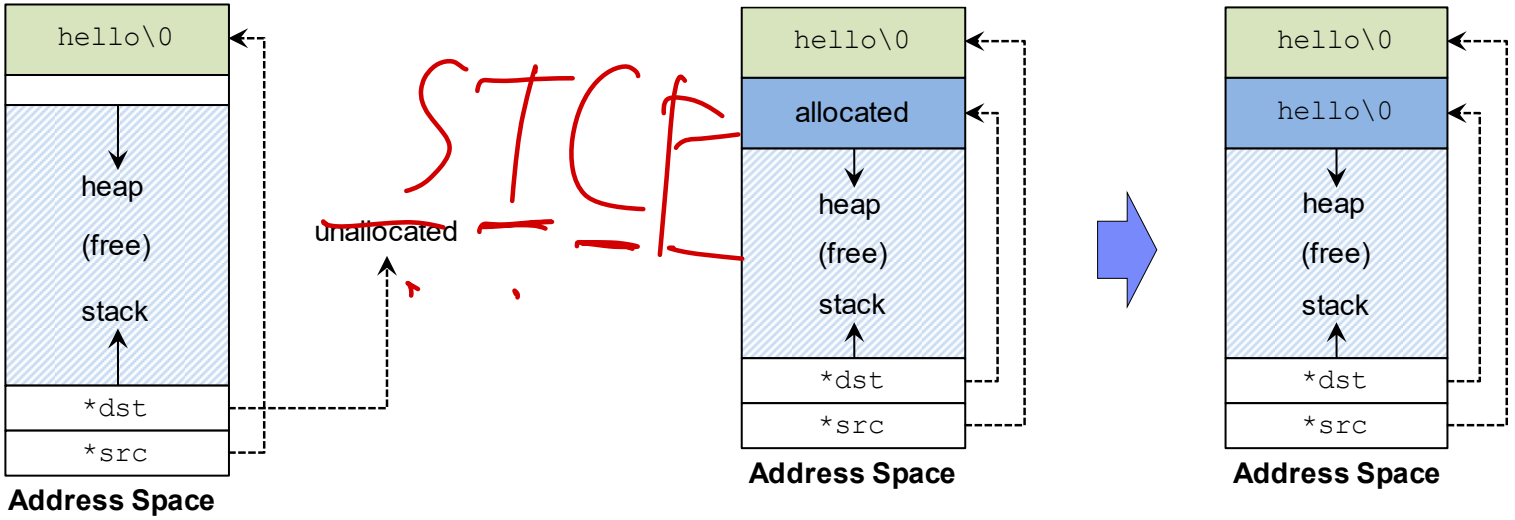
- **There are a number of common errors that arise in the use of `malloc()` and `free()`**
  - Many languages have support for automatic memory management; memory allocation and garbage collection

- **Segmentation Fault**

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

- **Correct Code**

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

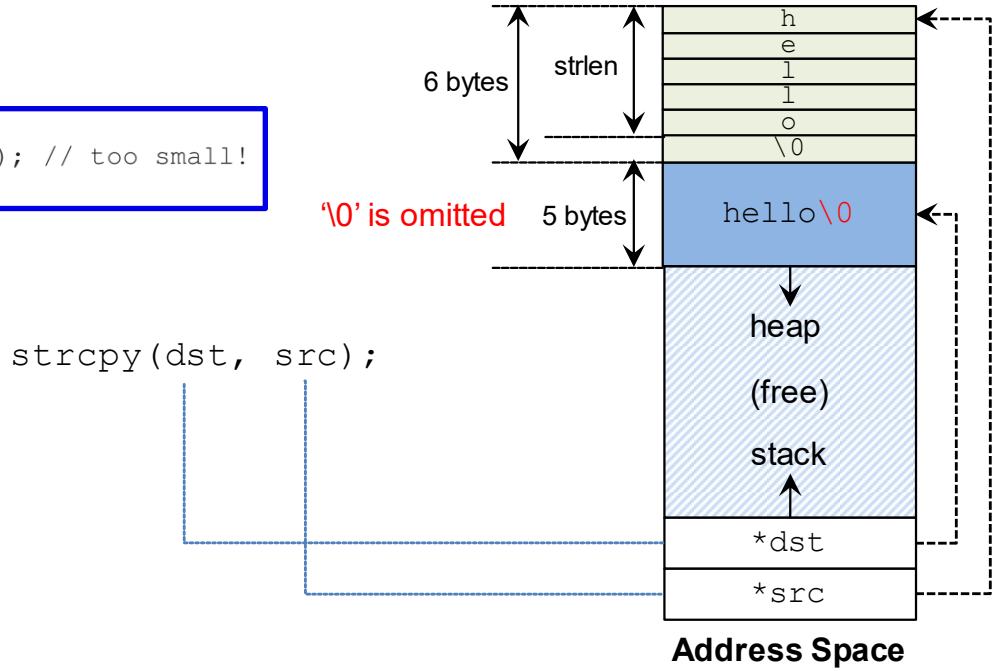


Courtesy of Prof. Youjib Won @ KAIST

Common Errors: Not Allocating Enough Memory

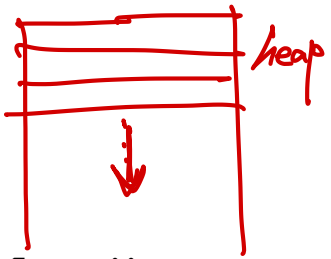
- **A related error is not allocating enough memory, sometimes called a buffer overflow**
  - The buffer overflows can be incredibly harmful, and in fact are the source of many security vulnerabilities in systems

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

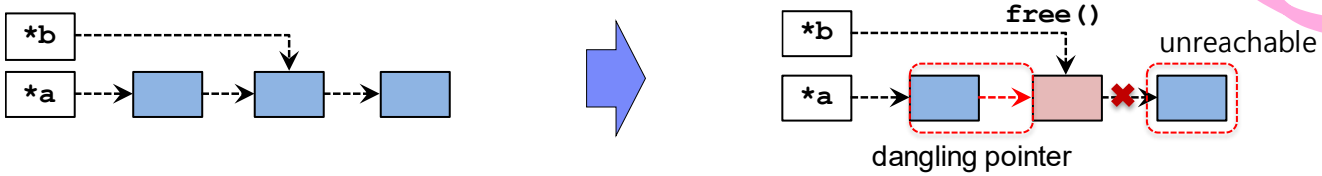


Courtesy of Prof. Youjib Won @ KAIST

Common Errors: Others



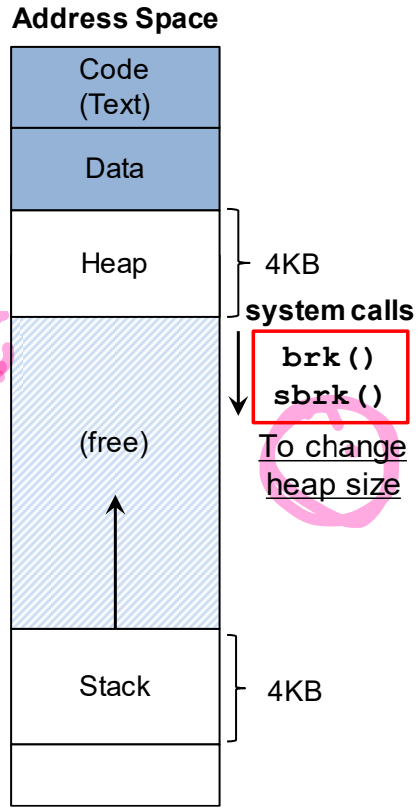
- **Forgetting to Initialize Allocated Memory**
  - If you forget to initialize the memory allocated by `malloc()`, your program will eventually encounter an uninitialized read
- **Forgetting to Free Memory**
  - Another common error is known as memory leak, and it occurs when you forget to free memory; A program runs out of memory and eventually is killed by OS
  - `malloc()`? Then, don't forget `free()`!
- **Freeing Memory before You are Done with It**
  - A program will free memory before it is finished using it; called dangling pointer



- **Freeing Memory Repeatedly**
  - The results of doing double free is undefined
- **Calling `free()` Incorrectly**
  - `free()` expects to have one of pointers you received from `malloc()` earlier

Underlying OS Support

- **`malloc()` & `free()` are not system calls, but rather library calls**
  - The malloc library manages space within the virtual address space, but itself is built on top of some system calls
- **One such system call is called `brk()`,**
  - used to change the location of the program's break: the location of the end of the heap
  - `sbrk()` is a similar call
  - Programmers should never directly call either `brk()` & `sbrk()`, stick to `malloc()` & `free()` instead
- **You can also obtain memory by `mmap()`**
  - `mmap()` can create an anonymous memory region within your program (swap space)
- **Other Calls**
  - `calloc()` allocates memory with zero initialization
  - `realloc()` resizes the allocated region



Courtesy of Prof. Youjib Won @ KAIST