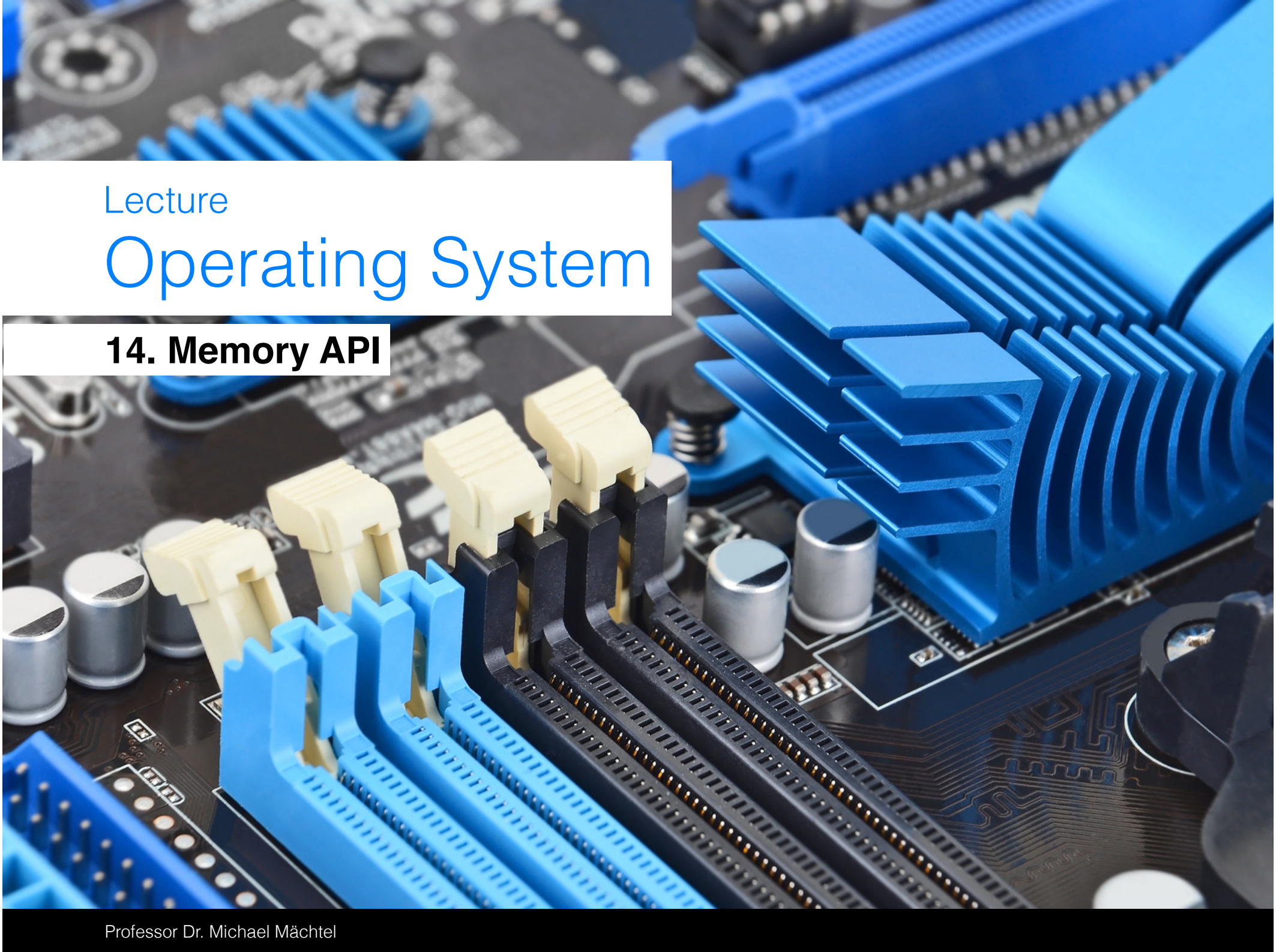


Lecture

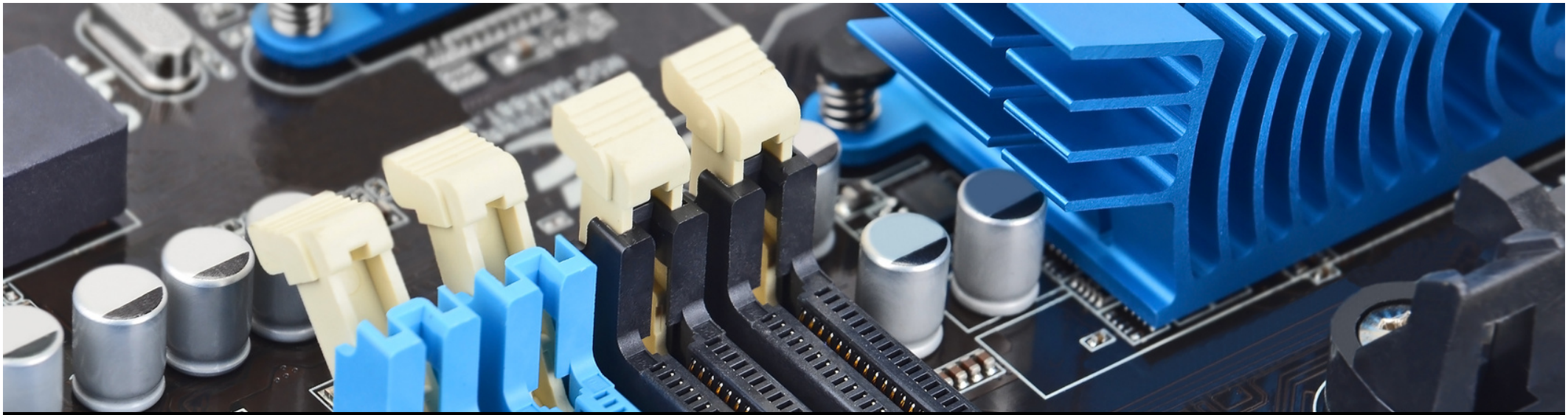
Operating System

14. Memory API

Professor Dr. Michael Mächtel

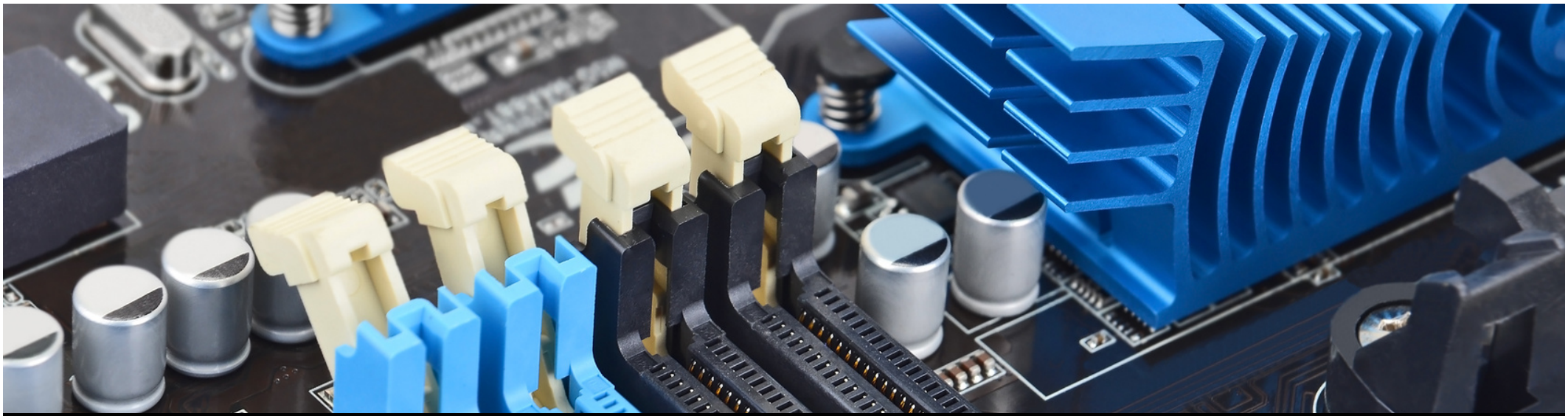
14. Memory API

1. `malloc`, `sizeof` and `free`
2. Common Errors
3. Underlying OS Support
4. Garbage Collector



14. Memory API

1. **`malloc`, `sizeof` and `free`**
2. Common Errors
3. Underlying OS Support
4. Garbage Collector



Memory Region on the stack

- Declaring memory on the stack in C is easy.
- Integer in function
  - Declare int
  - The compiler does the rest
    - After return from the function, the compiler deallocates the memory

```
void func(int argc, char *argv[]){

    int x = 3; // declares an integer on the stack

    …
}
```

Memory Region on the heap

- `malloc()`: Allocate a memory region on the heap.
- Argument
  - `size_t size` : size of the memory block(in bytes)
  - `size_t` is an unsigned integer type.
- Return
  - Success : a void type pointer to the memory block allocated by `malloc`
  - Fail : a `null` pointer

```
#include <stdlib.h>

void free(void* ptr)
```

Size of Memory on the heap

- Routines and macros are utilized for `size` in `malloc` instead typing in a number directly.
- Two types of results of `sizeof` with variables
  - The actual size of `x` is known at run-time.
  - The actual size of `x` is known at compile-time.

heap1.c

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

```
prompt> ./heap1
4
```

heap2.c

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

```
prompt> ./heap2
40
```

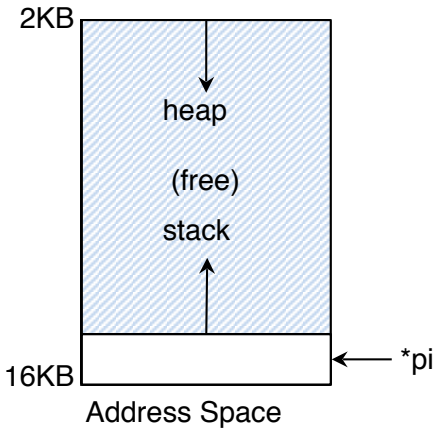
Free memory region on heap

- `free()`: Free a memory region allocated by a call to `malloc`.
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

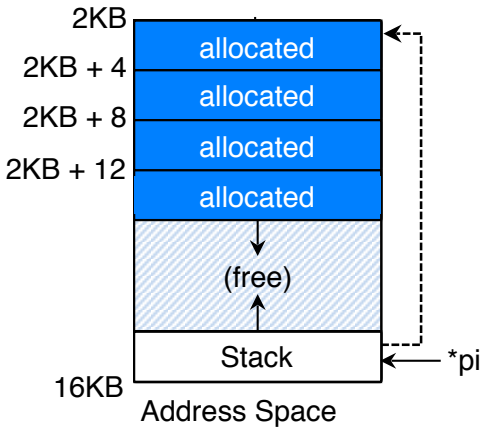
```
#include <stdlib.h>

void free(void* ptr)
```

Memory Allocation

```
int *x; // local variable
```

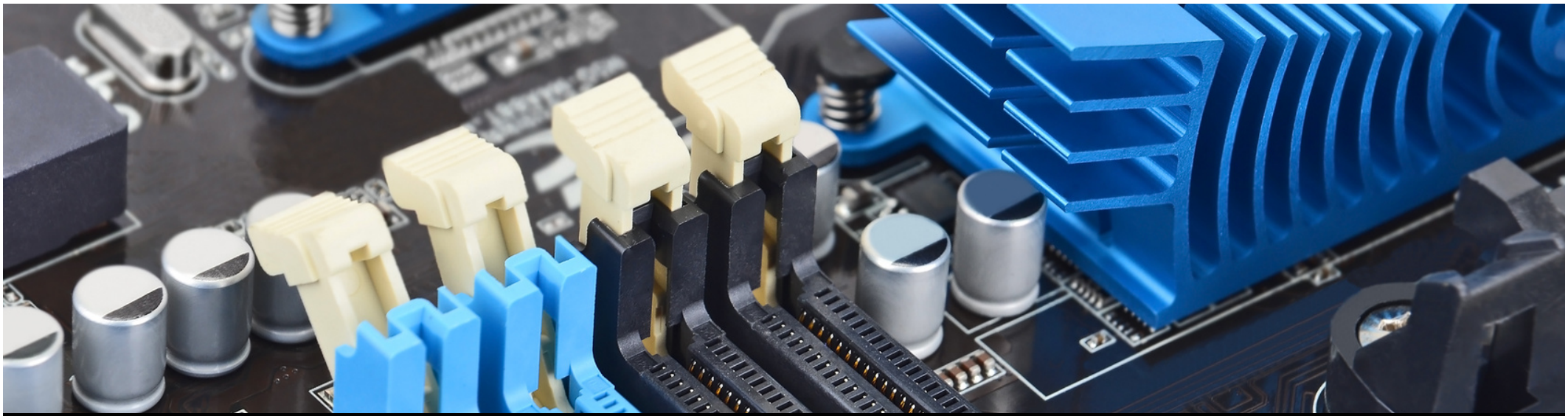



```
pi = (int *x) malloc(sizeof(int)*4);
```



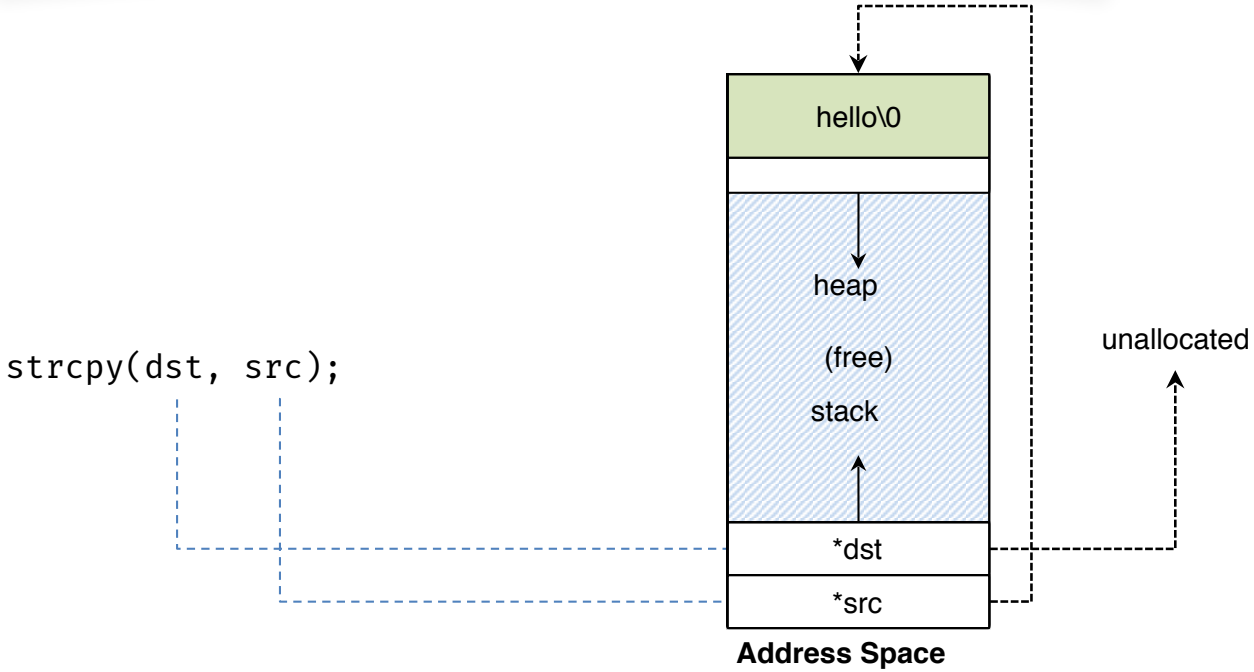
14. Memory API

1. `malloc`, `sizeof` and `free`
2. **Common Errors**
3. Underlying OS Support
4. Garbage Collector



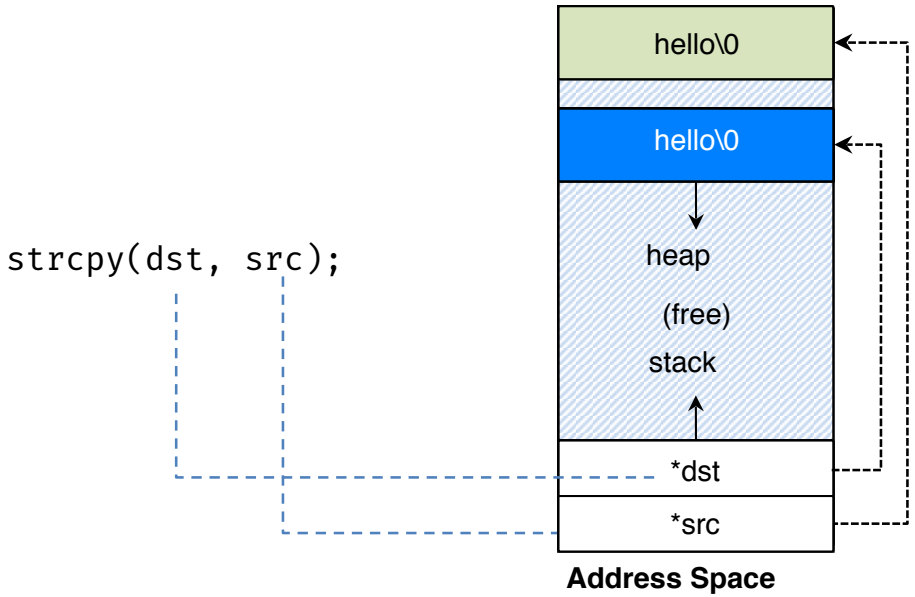
Forgetting To Allocate Memory


```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



NOT Forgetting To Allocate Memory

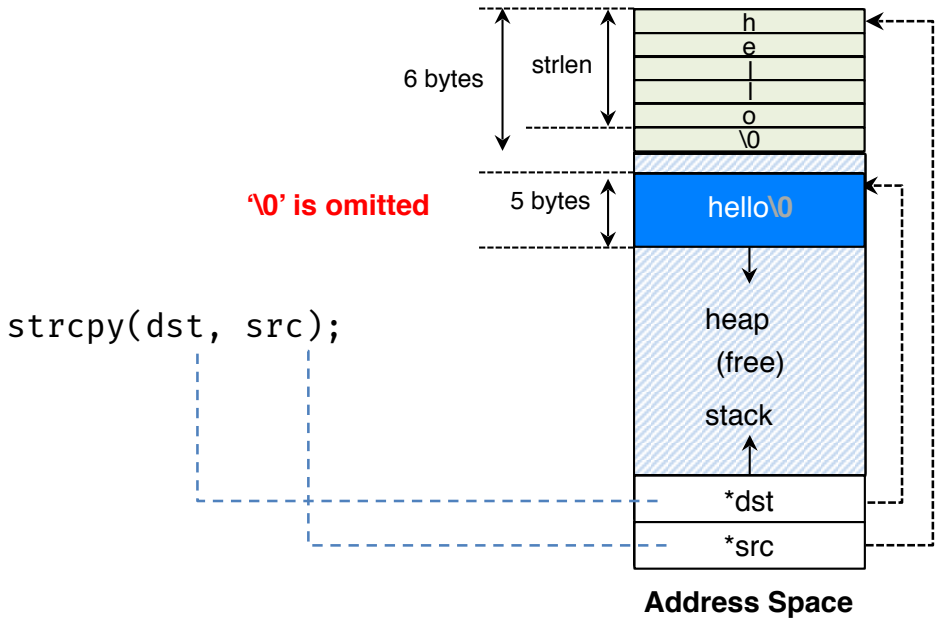
```
char *src = "hello"; //character string constant
char *dst = (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);    //work properly
```



Not Allocating Enough Memory

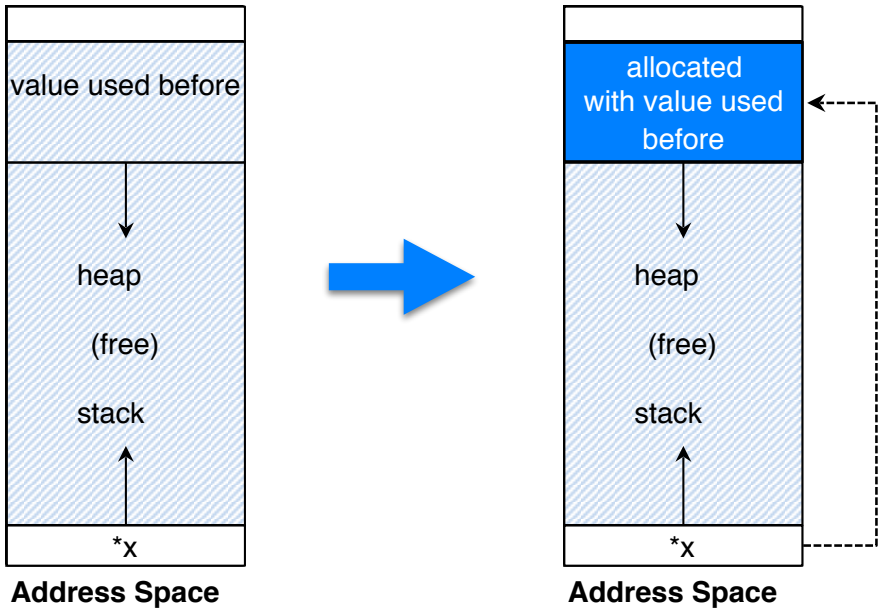
- Incorrect code, but `strcpy` works properly

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



Forgetting To Initialize

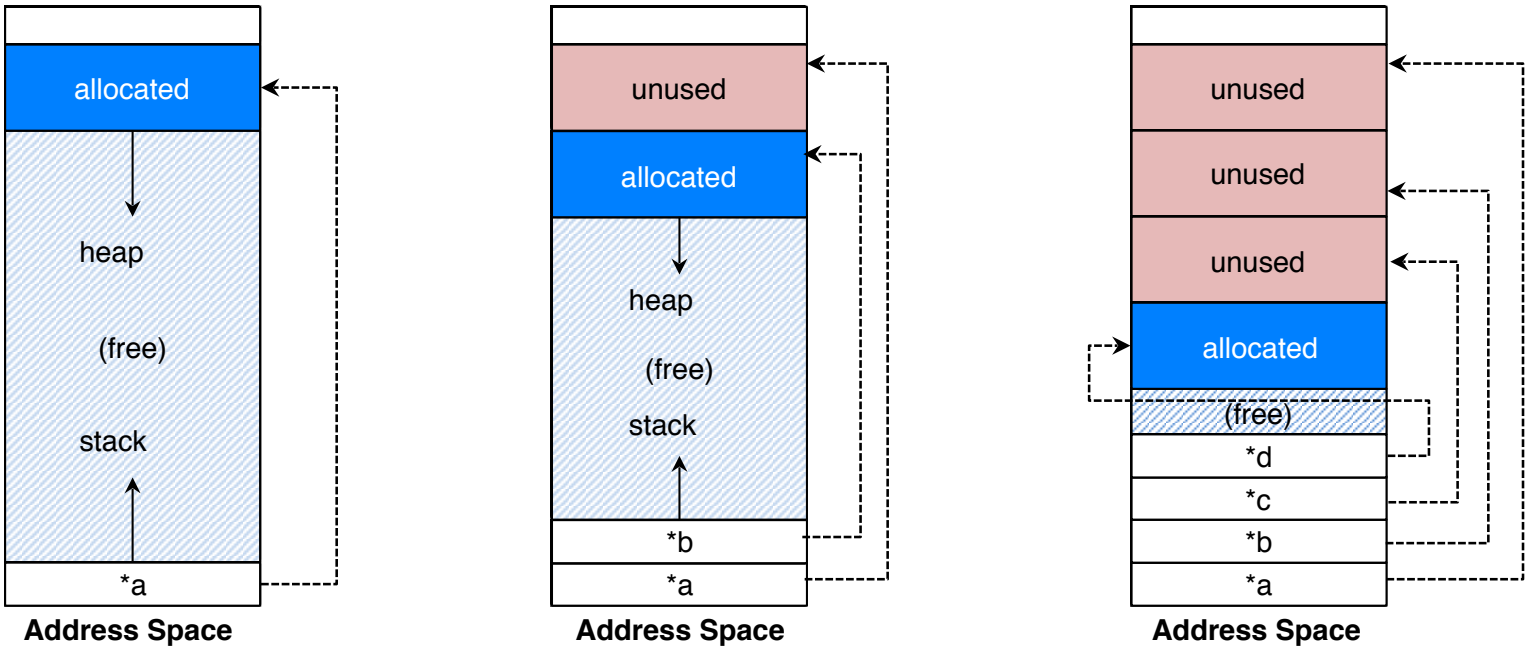

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```



Memory Leak

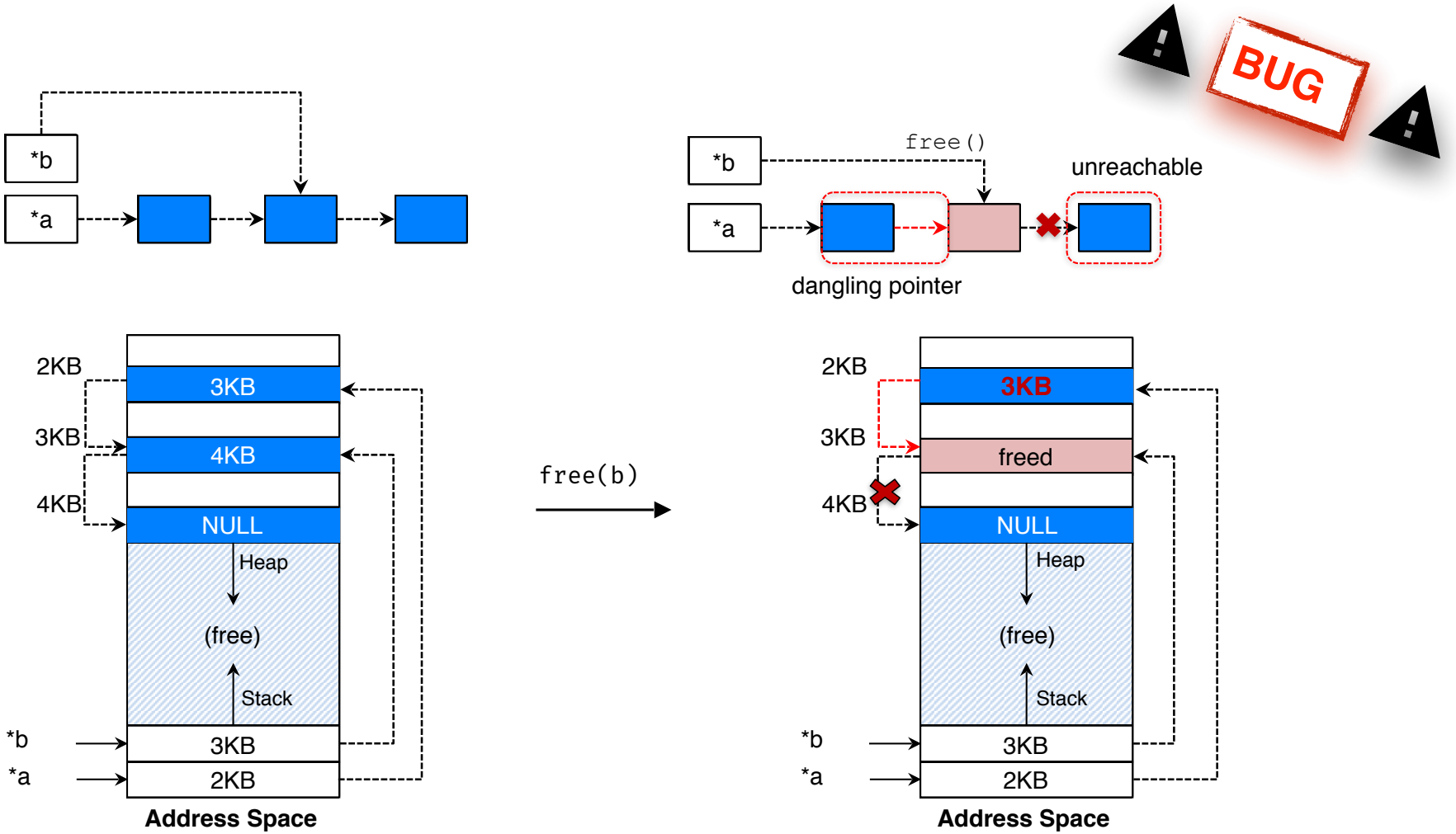
A program runs out of memory and eventually dies.

unused : unused, but not freed



Dangling Pointer

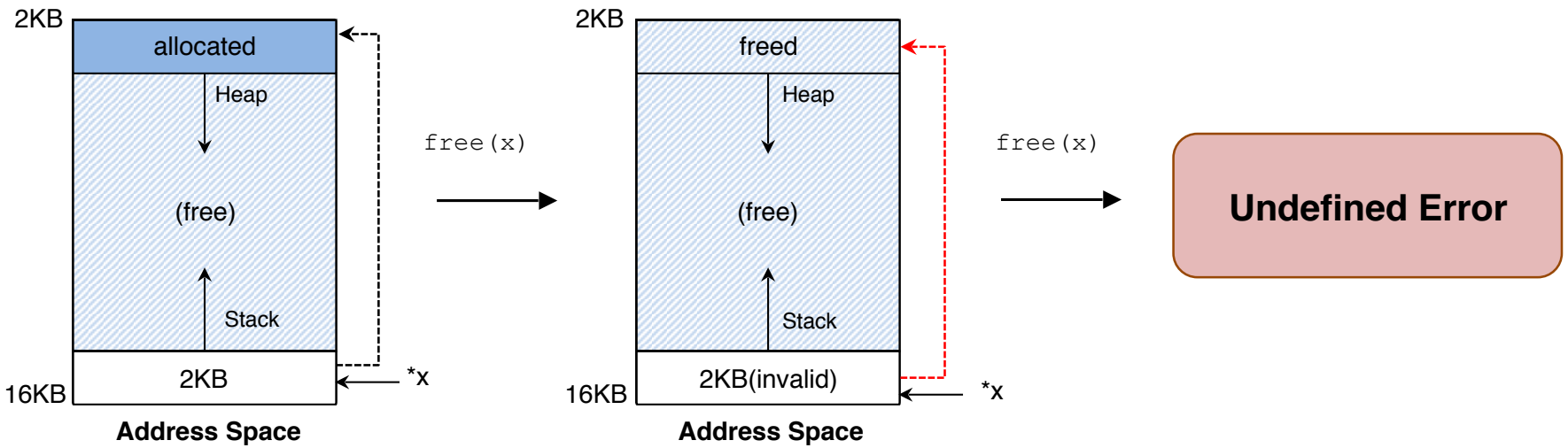
A program accesses to memory with an invalid pointer



Double Free

Free memory that was freed already.


```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```

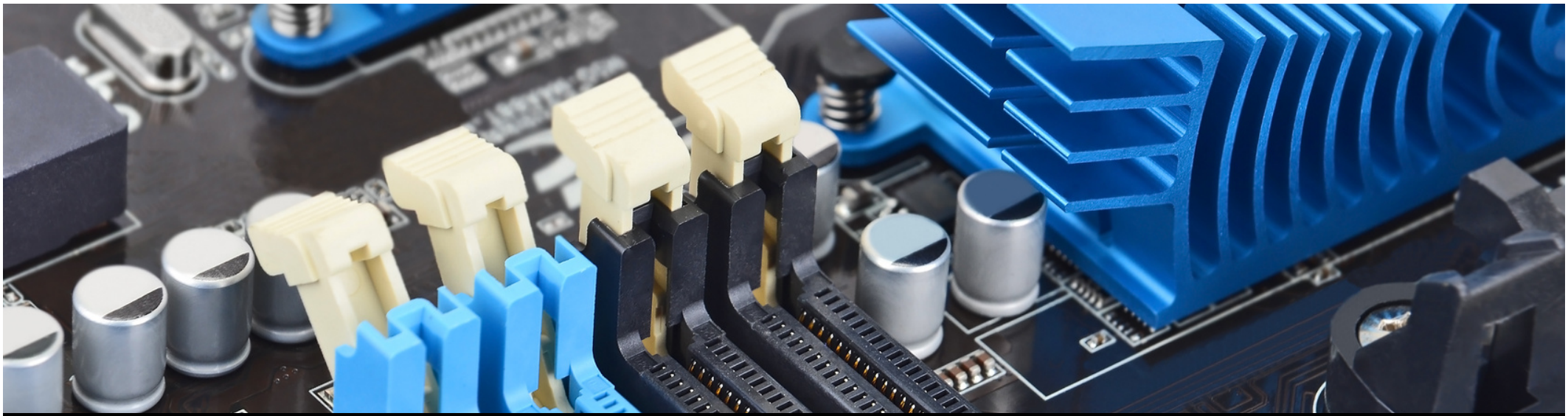


Other Memory APIs: realloc()

- Change the size of memory block.
  - A pointer returned by `realloc` may be either the same as `ptr` or a new.
  - Argument
    - `void *ptr`: Pointer to memory block allocated with `malloc`, `calloc` or `realloc`
      - `size_t size`: New size for the memory block(in bytes)
- Return
  - Success: Void type pointer to the memory block
  - Fail : Null pointer

14. Memory API

1. `malloc`, `sizeof` and `free`
2. Common Errors
3. **Underlying OS Support**
4. Garbage Collector



Underlying OS Support

- `malloc()` and `free()`
  - are library calls, **not system calls**
- `malloc` library call use `brk` system call.
  - `brk` is called to expand the program's *break*.
    - *break*: The location of **the end of the heap** in address space
  - `sbrk` is an additional call similar with brk.
- Programmers **should never directly call** either `brk` or `sbrk`.

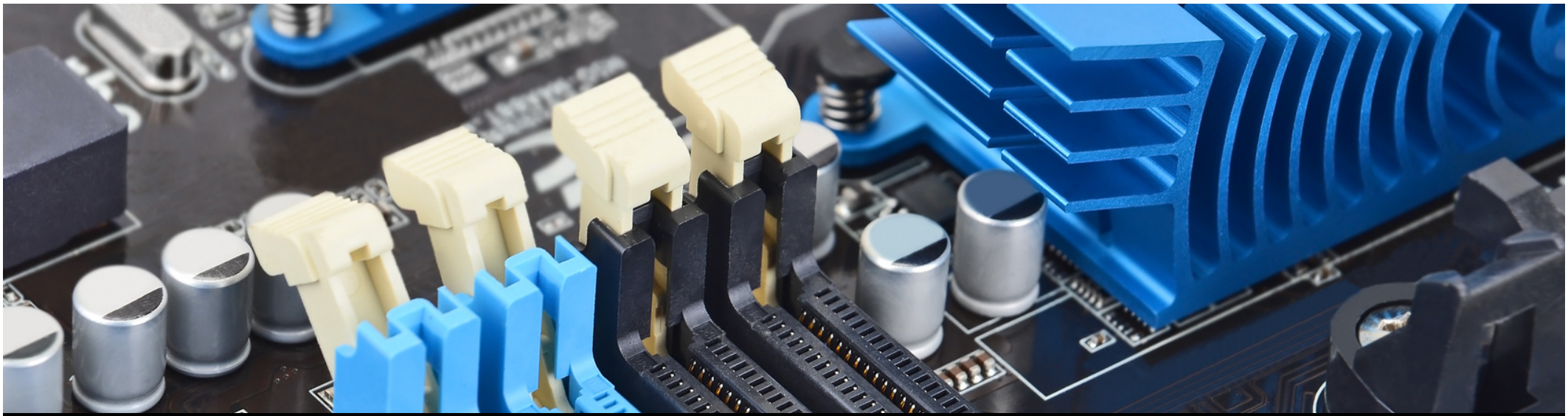
mmap()

- Finally, you can also obtain memory from the operating system via the `mmap()` call
  - `mmap` system call can create **an anonymous** memory region.
    - a region which is not associated with any particular file but rather with swap space.
    - This memory can then also be treated like a heap and managed as such.
    - is used to create ‘shared’ Memory

```
#include <sys/mman.h>

void *mmap(void *ptr, size_t length, int port, int flags, int fd, off_t offset)
```

14. Memory API

1. `malloc`, `sizeof` and `free`
2. Common Errors
3. Underlying OS Support
4. **Garbage Collector**



Garbage Collector?

Reference: „Why I hate garbage collectors“

Spot the Leak: C/C++

old C++

```
class Enemy {
private:
    EventFilter *filter;

public:
    Ememy(World *w, EnemyType t) {
        // we only want to receive specific events
        this→filter = new EventFilter(t);
        w.register_event_handler(filter, this);
    }
};
```

"I own this"

who own's this?

where's the delete?

shouldn't delete

Spot the Leak: Java

Java

```
class Enemy {

    private EventFilter filter;


    public Ememy(World w, EnemyType t) {
        // we only want to receive specific events
        this.filter = new EventFilter(t);
        w.register_event_handler(filter, this);
    }
};
```

GC deletes `filter`

`this` still in world

```
List<Enemy> enemies;
World w;

if (random())
    enemies.add(new Enemy(weak));

// remove all dead enemies
enemies.removeIf(e → e.isDead());
```

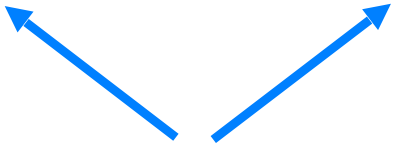
Rules!

- Important Rules:
  - For every `malloc` there must be a `free`
  - For every `new` there must be a `delete`
  - Don't you dare to cross those and don't `free` twice
  - Be clear about ownership
  - C++: **RAII** (we'll come to that later)
- Rules?

~~GC to the rescue~~

Memory Leak	Lingering Object	Resource Leak
forgot to **`delete`**/ **`free`**	referenced but unused	forgot to close

Garbage Collector



well …. something ….

What are resources?

- examples:
  - file
  - socket
  - database handle
  - …
- **All with „open()/close() semantics“ is a resource!**

Resource Acquisition Is Initialization

- The constructor acquires resources
- The destructor releases resources
- The destructor is guaranteed to be called
- Stack mechanics to the rescue!
- RAII in detail:
  - RAII relies on an automatic call to a common function at a defined time.
  - C++ uses a function that is already known to everyone = the destructor.
  - Destructor (in Rust drop()) is garanteed to be called
- **All three types of leaks can be avoided!**

GC and RAII

- Idea: GC for memory leaks, RAII for everything else
- Java: finalize()?
  - not reliable!
  - is only called when GC is in the mood …
- Fix the leak: Java

```
class Foo implements Closable {
  private EventFilter f;

  public Foo(int a) {
    f = new EventFilter(a);
    GlobalEvents.Get().Register(f, this);
  }

  public void Close() {
    GlobalEvents.Get().Unregister(this);
  }
};
```

```
void UseFoo() {
  Foo f;
  try {
    f = new Foo(10);
  } finally {
    f.Close();
  }
```

```
void UseFoo() {
  try (Foo f = new Foo(10)) {
    // code goes here  }
  }
```

GC with RAII

Memory Leak	Lingering Object	Resource Leak
forgot to `delete`/ `free`	referenced but unused	forgot to close

- ~~Finalizer?~~ unreliable!
- Defer (do, D)?
- Disposable (C#) / Closable (Java)?

Manual resource mangement

- Traditional Languages
  - Static code analysis to the rescue
- Some things can only be seen at runtime ...
  - ... unless you extend the language
  - ... and let the compiler detect things.
- Hello: **`Rust`**
  - If you are not used to think about ownership you will forget things
  - If you have no enforced rules you will miss things

With GC

- Developers get a false sense of safety
- Things tend to get even more complicated
- The problem is delayed, not solved
- Technical implications of GCs are often very bad for realtime or embedded applications