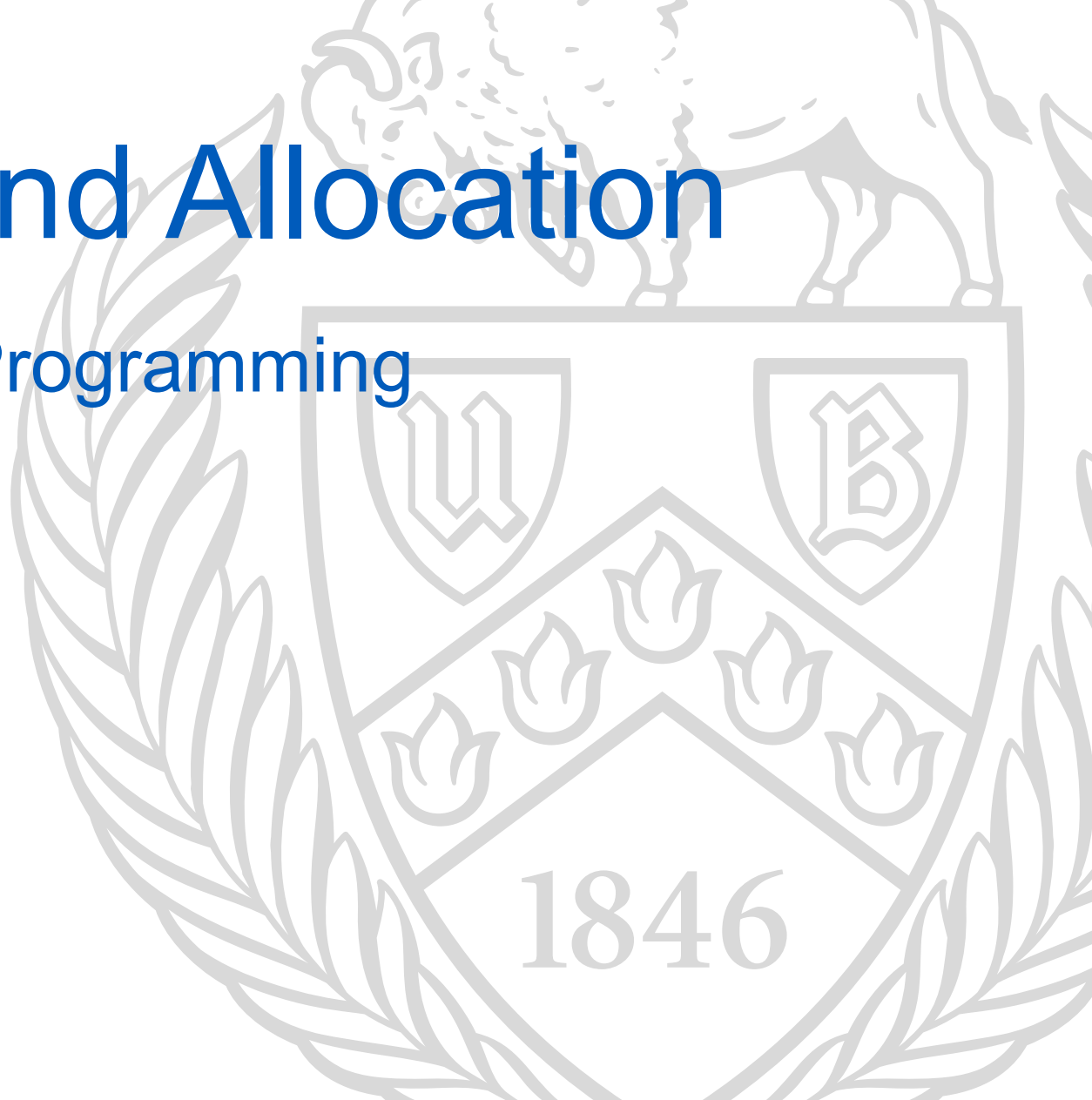


Structures and Allocation

CSE 220: Systems Programming

Ethan Blanton
Jesse Hartloff

Computer Science and Engineering
University at Buffalo



Effective Questions

For **programming questions**, ask:

- What did I **do**?
- What did I **expect** to happen?
- What **actually** happened?
- **How are they different?**

You must know what you expected to identify the problem!

When asking us questions, **tell us** what you did, what you expected, and what you got.

Building Complex Applications

Building more complex applications frequently requires:

- Data structures, including self-referential structures
- Allocation of memory at program run time

Structures are provided by the C language.

Memory allocation is provided by the C library.

Structures

A C struct aggregates multiple data items into one value.

These items are called *members*.

The aggregate is a *new (struct) type*.

The members of a structure are stored together.

Self-referential structures can form linked lists, *etc.*

Memory Allocation

The **amount of memory** used by complex data may not be known at compile time.

Solving this requires **memory allocation**.

Self-referential structures like lists are **normally allocated**.

Allocation and release of memory in C is **manual**.

This makes C **memory efficient**, but also **prone to leaks**.

The C Struct

A struct is a **compound data type** consisting of **one or more other types**.

```
struct IntList {  
    int          value;  
    struct IntList *next;  
};
```

This struct contains an **integer** and a **pointer**.

value and next are called *members* of the structure.

Any **variable of type struct IntList** contains both of these members.

Declaring and Using Structures

The syntax for structure declaration is

```
struct StructureTypeName {  
    // Members in structure  
    // Each member has a type and a name  
} varname; // semicolon required!
```

An **instance of the structure** may be created where the structure is declared, or using the type name later:

```
struct StructureTypeName varname;
```

Accessing Structure Members

The `.` [operator](#) is used to access the members of a structure.

```
struct IntList node = { 7, NULL };  
node.value = 3;
```

[Any member of a structure](#) can be accessed with `.`:

```
struct Complex {  
    double real, im;  
};  
struct ComplexList {  
    struct Complex complex;  
    struct ComplexList *next;  
} complexlist;  
complexlist.complex.real = 0.0;
```

Structure Pointers

The `.` operator is *cumbersome* for structure pointers:

```
struct IntList *list = get_list_pointer();  
(*list).next = NULL;
```

The `->` operator is *syntactic sugar* for `(*)` `.`:

```
list->next = NULL;
```

The `->` operator can be used to access any member of a structure *via a pointer to the structure type*.

Operations on Structures

A structure **value**:

- Can have its address taken with &
- Can be copied with =
- Can be used to access a member with .

A structure **pointer**:

- Can do all the things any pointer can do
- Can be used to access a member with ->

No other operations on structures are legal!

Aside: The sizeof operator

There are several [operators](#) used to help with [reflection](#) in C.

One of these is the sizeof operator.

It returns the size [in bytes](#) of its operand, which can be:

- A variable
- An expression that is “like” a variable
- A type

(Expressions “like” a variable include, *e.g.*, members of structures.)

Looking at sizeof

Examples:

```
void func(int matrix[2][3]) {  
    double dist;  
    sizeof(int);           // yields 4  
    sizeof(dist);         // yields 8  
    sizeof(matrix);       // yields ... 8?  
}
```

Note that sizeof arrays **is not reliable**.

Only arrays **declared within the current scope** will be correct. ¶

We will discuss the sizes of things in more detail, later.

The void * type

The type void * is used to indicate a **pointer of unknown type**.

You may recall that void indicates a **meaningless return value**.

void * is treated specially by the C compiler and runtime:

- A void * variable can store **any pointer type**
- Type checks are **mostly bypassed** assigning to/from void *
- Any attempt to **dereference a void * pointer** **is an error**

Interlude

Lecture question!

Pointer Assignments

Consider the following:

```
int i;  
double d;  
int *pi = &i;  
double *pd = &d;
```

Pointer Assignments

Consider the following:

```
int i;  
double d;  
int *pi = &i;  
double *pd = &d;
```

Each of these pointers is **typed**. These are errors:

```
pi = pd;  
pd = pi;
```

Pointer Assignments

Consider the following:

```
int i;  
double d;  
int *pi = &i;  
double *pd = &d;
```

This is where it gets dangerous:

```
void *p = pi;  
pd = p;
```

This is perfectly legal.

(What does it mean?)

The Standard Allocator

The C library contains a [standard allocator](#).

With this allocator you can request memory from the system.

Allocated memory is identified [by its address](#).

Requesting Memory

Memory is requested using `malloc()` or `calloc()`:

```
void *malloc(size_t size);
```

```
void *calloc(size_t nmemb, size_t size);
```

`malloc` accepts:

- A `size_t` size in **bytes**

`calloc` accepts:

- A `size_t` **number of members** of an array
- A `size_t` size in **bytes** of **each member**

Both return a `void *` **pointer** to usable memory.

`calloc()` sets **all bytes in the memory** to zero.

Allocation Sizes

It is **impossible to tell** how much memory a pointer “points to.”

The allocator returns **at least as much** as the user requested.

If you need to know how much that was, you need more information!

Typically:

- From a variable or argument (e.g., argc)
- From a member in a struct (e.g., nprios in PA2)
- Using knowledge of the data (e.g., strlen() on a string)

Allocating a Structure

As an example, let's allocate a structure:

```
struct IntList *get_list_pointer() {  
    struct IntList *head = calloc(1, sizeof(struct IntList));  
    return head;  
}
```

Note that:

- The integer in head is set to 0
- The next pointer in head is set to NULL

Allocating an Array

We can also allocate **arrays**.

```
malloc(10 * sizeof(int));    // Array of 10 ints
```

```
calloc(10, sizeof(int));    // Array of 10 ints
```

In C, an array is just multiple data items adjacent in memory!

Note the slight syntax difference!

Freeing Memory

C has no **garbage collector**.

The programmer is responsible for **freeing** memory after use.

The function `free()` does this:

```
void free(void *ptr);
```

Free accepts:

- A pointer allocated by `malloc()`, `calloc()`, or `realloc()`

Once a pointer is freed, that pointer **must not be used again**.

Failed allocations

Allocations **can fail**.

A failed allocation **will return NULL**.

On a **modern machine**, this *usually* means an unreasonable allocation.

E.g., you accidentally allocated 2 GB instead of 2 KB.

On **smaller systems**, failed allocations are **normal**.

Often you can't **do much** about a failed allocation, of course.

Use-after-free

A common class of error is **use-after-free**.

This is when a **freed pointer** is used.

This is **particularly dangerous**, because C may **reuse that pointer**.

Therefore, it is:

- Pointing to **usable memory**
- Not valid
- **Likely to corrupt data!**

Setting free'd pointer variables to NULL can help prevent this.

Out-of-bounds access

Because heap allocations **have no obvious size**, out-of-bounds access is easy.

```
int *array = malloc(2 * sizeof(int)); /* int[2] */

for (int i = 0; i <= 2; i++) {           /* 0, 1, 2! */
    array[i] = 0;                        /* Illegal */
}
```

The compiler **will not catch this**.

Interlude

Lecture question!

Summary

- Structs are a collection of values.
- Structs can be self-referential.
- The C standard library contains a flexible allocator.
- Standard allocator allocations are **sized by the programmer**.
- C **does not provide a way** to query the size of an allocation.

Bibliography

Required Readings

- [1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 2: 2.7; Chapter 6: Intro, 6.1–6.7. Prentice Hall, 1988.
- [2] Linux project. man 3 malloc. no date.

Optional Readings