

# Alignment, Padding, and Packing

CSE 220: Systems Programming

Ethan Blanton & Carl Alphonse

Department of Computer Science and Engineering  
University at Buffalo



# Effective Questions

Asking follow-up questions **is also a skill**.

When you get an answer, **set a timer**.

(Maybe 5 or 10 minutes, this time!)

**Think about the answer** during that time.

When the timer goes off:

- Can you make progress now?
- If not, **why not?**
- Do you need to ask a clarifying question?

# Interlude

# Lecture question!

# Simple vs. Compound

C has two basic kinds of types: **simple** and **compound**.

A **simple type** is a type that contains a **single value**.

In C, the simple types are:

- **arithmetic types** (integers and floating point numbers)
- **pointers** (which we have learned are special integers)

**Compound types** are **collections of other values**.

In C, the compound types are:

- **arrays** of simple or compound values of the same type
- **structs** containing values of the same or different types

# Memory Layout

Many data types must be located in memory according to certain rules.

In most cases, this is not obvious to the programmer.  
(In many languages, it is impossible to tell!)

Compound types, and pointers to compound types, expose this.

We will explore alignment and stride.

# More on `void` Pointers

Void pointers are powerful for `raw memory manipulation`.

You can use them to put `arbitrary values` into memory.

You will use this in PA3 and PA4!

We will look at using `void *` to:

- Pass a pointer of an arbitrary type
- Read and write arbitrary types in memory
- Manipulate memory without respect to alignment and stride

# Alignment

We have previously discussed **words**.

Recall that:

- The **memory bus** has a certain width
- Memory transfers data in **words**

Most systems can only access **words in memory** on **addresses divisible by the word size**.

Often the **address** of a value must be **evenly divisible** by the **size of its type**.

Thus, if an **int** is 32 bits, its address is divisible by 4.  
(32 bits / 8 bits per byte = 4 bytes, addressed in bytes)

# Simple Type Layout

Simple types must typically be **aligned to their size**.

Alignment rules **vary between architectures**.

Some platforms can **still access** unaligned simple types.

Some platforms **will raise a hardware error** for unaligned access.

**Most platforms suffer a performance penalty** for unaligned access.



# Array Layout

The first element of an array of simple types is typically aligned to the size of an array element.

This automatically aligns all items in the array.

For other types of arrays, things can get more complicated.

To understand alignment of compound types, we must understand structure layout.

# Structure Layout

The members of a structure are **adjacent** in memory.

This is similar to simple types in an array.

However, there are **additional considerations** regarding layout.

The **alignment of array members** must be preserved!

**Padding** is inserted between values to bring them into alignment.

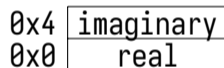
Padding is **unused memory** and you **cannot assume its value**.

# Uncomplicated Layout

In the least complicated case,  
members are adjacent.

Every member is laid out in  
order.

```
struct ComplexFloat {  
    float real;  
    float imaginary;  
};
```

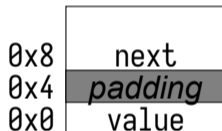


# Struct Padding

In a structure, padding is applied **between values**.

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

This struct is **16 bytes** and contains **4 bytes of padding**.



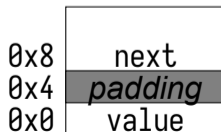
# Struct Alignment

For **padding in structures** to work, **the struct must be aligned**.

Consider the previous example:

- If the address of the struct is divisible by 4, value is aligned, but next **might not be**
- If the address of the struct is divisible by 8, **both are aligned**

The **struct itself** is ordinarily aligned to the largest requirement of **any member**.



# Alignment and Allocation

Recall that **the standard allocator doesn't know what you're allocating**.

For this reason, `malloc()` et al. normally align **to the largest requirement on the system**.

This ensures that **any properly aligned structure** will be aligned.

This leads to **overhead** which can cause **significant waste**.

We'll see much more about this later.

# Stride

Stride is closely related to [alignment](#), yet different.

Stride is the [difference between two pointers](#) to [adjacent values](#) of a particular type.

For simple types, [stride is the same as size](#).

For example:

- `int` is 32 b, `sizeof(int)` is 4, stride of `int *` is 4.
- `double` 64 b, `sizeof(double)` is 8, stride of `double *` is 8.

For [compound types](#), this can get more complicated.

`void *` is a [special case](#), and its stride is 1.

# Interlude

# Lecture question!

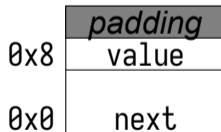


# Stride in Compound Types

Consider this struct:

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

It lays out in memory like this:



Padding here is to **adjust stride** to **preserve alignment**.

# Pointer Arithmetic

Pointers are **integer types**, and **can be computed**.

**Pointer arithmetic** operates in **stride-sized** chunks.  
(This is why pointers can dereference like arrays!)

```
double *dptr = &somedouble;
```

If the value of `dptr` were  $\theta$ , `dptr + 1` would be **eight**, not one!  
This is because a double is **8 bytes wide**.

# Pointer Arithmetic — Compound Types

Stride for **compound types** can be quite large.

Consider:

```
struct Big {  
    char array[256];  
};  
struct Big *b = NULL;
```

In this case,  $b + 1$  is **the address 256!**

# Dumping Memory

```
#include <stdio.h>

void dump_mem(const void *mem, size_t len) {
    const char *buffer = mem;    // Cast to char *
    size_t i;

    for (i = 0; i < len; i++) {
        if (i > 0 && i % 8 == 0) { putchar('\n'); }

        printf("%02x ", buffer[i] & 0xff);
    }
    putchar('\n');
}
```

# dump\_mem Details

What is this for?

```
const char *buffer = mem;
```

# dump\_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use mem as an array of bytes”.

# dump\_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use mem as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ putchar('\n'); }
```

# dump\_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use mem as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ putchar('\n'); }
```

It prints a newline after every 8th byte excepting the first.



# dump\_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use `mem` as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ putchar('\n'); }
```

It prints a newline after every 8th byte excepting the first.

Finally:

```
buffer[i] & 0xff
```

# dump\_mem Details

What is this for?

```
const char *buffer = mem;
```

It tells the compiler “we’re going to use mem as an array of bytes”.

What about this:

```
if (i > 0 && i % 8 == 0){ putchar('\n'); }
```

It prints a newline after every 8th byte excepting the first.

Finally:

```
buffer[i] & 0xff
```

This is necessary to avoid [sign extension](#).

# Inconvenient Representation

Pointers to `void *` can be used to store and interpret representations that are inconveniently represented in C.

Consider the following structure:

```
struct Inconvenient {  
    int fourbytes;  
    long eightbytes;  
} inconvenient;
```

This structure contains 12 bytes of data, but occupies 16 bytes.  
(Because of padding...)

To communicate this structure we wish to send only 12 bytes.

# Serialization

Communicating such data is often done via **serialization**.

**Serialization** is the storage of data into a **byte sequence**.

In C, we do this with **pointers**, and often **void pointers**.

Consider:

```
void *p = malloc(12);
*(int *)p = inconvenient.fourbytes;
*(long *)(p + sizeof(int)) = inconvenient.eightbytes;
```

This builds a 12-byte structure **without padding**.

(In the process, it **violates alignment restrictions**.)

# Flexible Sizes

Another use for `void` pointer representation is **flexible sizes**.

Consider a structure (**not legal C**):

```
struct Variable {
    size_t nentries;
    int entries[nentries];
    char name[]; /* name is NUL-terminated */
} variable;
```

This structure **does not have a well-defined size**.

Its size depends on `nentries` and the length of `name`!

# Packing the Data

We can **serialize** this data as follows:

```
size_t nentries = 3;
int entries[] = { 42, 31337, 0x1701D };
const char *name = "Imogen Temult";

void *buf = malloc(sizeof(size_t)
                  + nentries * sizeof(int)
                  + strlen(name) + 1);
void *cur = buf;
```

# Packing the Data

We can **serialize** this data as follows:

```
*(size_t *)cur = nentries;
cur += sizeof(size_t);
for (int i = 0; i < nentries; i++) {
    *(int *)cur = entries[i];
    cur += sizeof(int);
}

for (int i = 0; i <= strlen(name); i++) {
    *(char *)cur++ = name[i];
}
```

# Packing the Data

We can **serialize** this data as follows:

```
size_t nentries = 3;
int entries[] = { 42, 31337, 0x1701D };
const char *name = "Imogen Temult";
```

```
03 00 00 00 00 00 00 00
2a 00 00 00 69 7a 00 00
1d 70 01 00 49 6d 6f 67
65 6e 20 54 65 6d 75 6c
74 00
```



# Interlude

# Lecture question!

# Summary

- Integers, pointers, and floating point numbers are **simple types**.
- Arrays and structures are **compound types**.
- Structures can contain members of **mixed type**.
- Simple types must be **aligned**.
- Compound types must **align for simple types**.
- Allocation normally aligns to the **largest requirement**.
- Pointer arithmetic **uses stride** in computations.
- `void *` has a **stride of 1**.
- The `void *` type can be used for **raw memory manipulation**
- **Casting `void *`** to another type is convenient
- Math on `void *` is **by byte**

# References I

## Required Readings

- [2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 5: Intro, 5.1-5.7; Chapter 6: Intro, 6.1-6.7. Prentice Hall, 1988.

## Optional Readings

- [1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 3: 3.8.1–3.8.3, 3.9.1, 3.9.3. Pearson, 2016.

# License

Copyright 2019–2025 Ethan Blanton, All Rights Reserved.

Copyright 2024 Eric Mikida, All Rights Reserved.

Copyright 2022–2025 Carl Alphonse, All Rights Reserved.

Copyright 2019 Karthik Dantu, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.