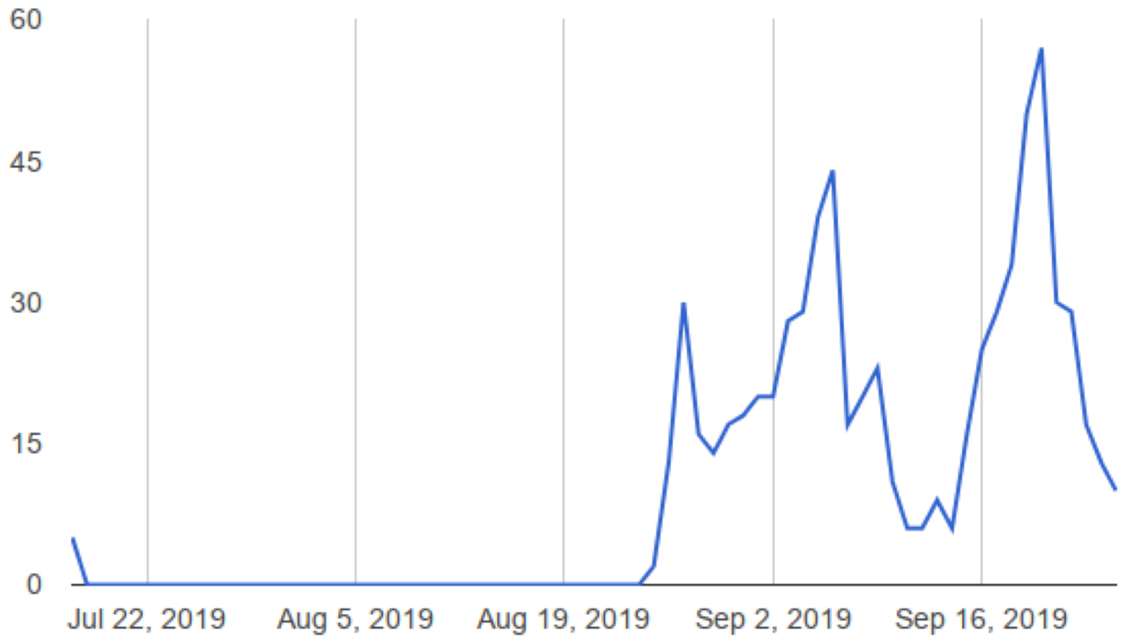
Alignment, Padding and Packing

Karthik Dantu
Ethan Blanton
Computer Science and Engineering
University at Buffalo
kdantu@buffalo.edu

Administrivia

- Midterm – postponed to Oct 11 (Friday) in class
- Lab exam 2 next week
  - No books, no notes
  - No Internet including Google Translate
  - Please hydrate and visit restroom before
- PA2 – please get started



Scalars vs Aggregates

- C has two types of data types: scalars and aggregates
- A scalar is a data type that contains a single value
- In C, scalar types are:
  - Arithmetic types (Integers, Floats, char)
  - Pointers (special integers)
- Aggregates contain collections of scalar values
- In C, aggregate data types are
  - Arrays – collections of scalars of the same data type
  - Structs – collections of scalars of different data types

Memory Layout

- Many data types must be located in memory according to certain rules
- In most cases, this is not obvious
- Aggregate types, and pointers to aggregate types demonstrate this
- We will explore this through alignment and stride

Aside: `void` Pointers

- Void pointers are useful for raw memory manipulation
- You can use it to put arbitrary values to individual bytes in memory
- You will need this in PA3 and PA4
- We will use `void *` to
  - Pass a pointer of an arbitrary type
  - Read and write arbitrary types of memory
  - Manipulate memory without respecting alignment and stride

Alignment

- Recall that
  - Memory bus has a certain width
  - Memory transfers data in words
- Most systems can only access words in memory on addresses divisible by word size
- Typically, the address of a value must be evenly divisible by the size of its data type
- E.g., if `int` is 32 bits, the address must be divisible by 4

Scalar Layout

- Scalars must typically be aligned to their size
- Alignment rules vary with architecture
- Some platforms can still access unaligned scalars
- Some platforms will raise a hardware error for unaligned access
- Most platforms will suffer a performance penalty

Array Layout

- The first element of an array of scalars is typically aligned to the size of the array element
- This aligns all items in the array
- For other types of arrays, things can get more complicated
- To understand the alignment of aggregate types, we must understand structure layout

Structure Layout

- The members of a structure are adjacent in memory
- This is similar to scalars 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

Simple Layout

- Members are adjacent
- Every member is laid out in order
- Lets assume `float` is 32-bit

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

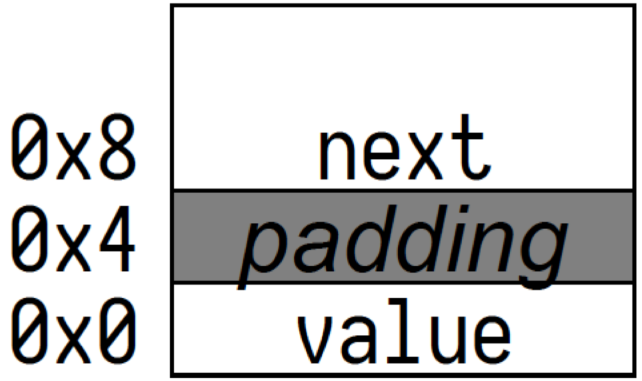
0x4	imaginary
0x0	real

Struct Padding

- In a struct, padding may be applied between values
- Lets assume pointers are 8 bytes long

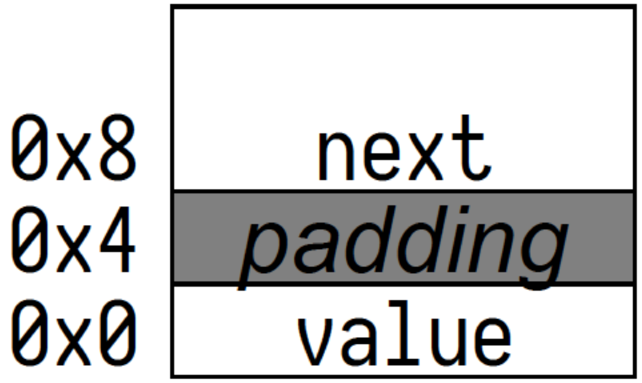
```
struct IntList {
  int value;
  struct IntList *next;
}
```

- This struct is 16 bytes with 4 bytes of padding



Struct Padding

- 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, then both are aligned
- The struct itself is aligned to the requirements of its largest 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 system requirement
- 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:

  If int is 32 bits, `sizeof(int)` is 4 and the stride of `int *` is 4

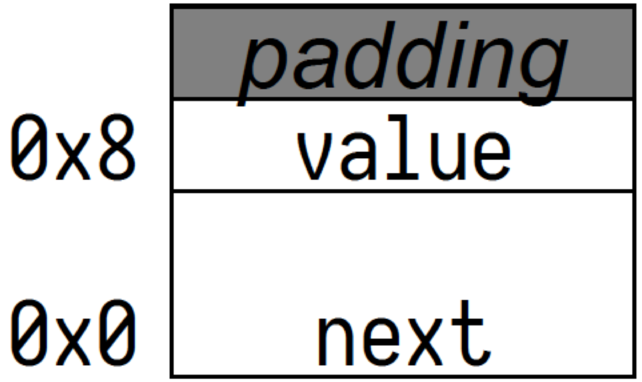
  If `double` is 64 bits, `sizeof(double)` is 8 and the stride of `double *` is 8
- For aggregate types, this can get more complicated
- `void *` is a special case, and its stride is 1

Stride in Aggregate Types

- Consider this struct

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

- Its memory layout is as follows



- 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 `0`, `dptr + 1` would be eight, not one
- This is because a `double` is 8 bytes wide.

Pointer Arithmetic – Aggregate Types

- Strides for aggregate data types can be large
- Consider

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

- In this case, `b + 1` is the address 256

Dumping Memory – dump_mem

```
#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) { printf("\n"); }
    printf("%02x ", buffer[i] & 0xff);
  }
  if (i > 1 && i % 8 != 1) { puts(""); }
}
```

dump_mem Details

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

- What is this for?
- "We are going to interpret `mem` as an array of bytes"

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

- "Print a newline fter everyeigth byte except the first"

```
printf("%02x ", buffer[i] & 0xff);
```

- Necessary to avoid sign extension

Inconvenient Representation

- Pointers to void * can be used to store and representations that are inconveniently represented in C
- Consider the following structure

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

- Structure contains 12 bytes of data but occupies 16 bytes
- To communicate this structure, we wish to send only12 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 of void pointer representation is flexible sizes
- Consider a structure (not legal C)

```
struct Variable {
  size_t nentries;
  int entries[nentries];
  char name[];
} variable;
```

- This structure does not have a well-defined size
- Its size depends on `nentries` and the size of `name`

Packing the Data

```
size_t nentries = 3;
int entries [] = { 42, 31337 , 0x1701D };
const char *name = "Caleb Widowgast";
void *buf = malloc(sizeof(size_t)
         + nentries * sizeof(int) +
           strlen(name) + 1);
void *cur = buf;
```

Packing the Data (2)

```
*(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 (3)

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

```
dump_mem()
    03 00 00 00 00 00 00 00
    2a 00 00 00 69 7a 00 00
    1d 70 01 00 43 61 6c 65
    62 20 57 69 64 6f 77 67
    61 73 74 00
```

Summary

- Integers, pointers, and floating point numbers are scalar types
- Arrays and structures are aggregate types
- Structures can contain members of mixed type
- Scalar types must be aligned
- Aggregate types must align for scalars
- Allocation normally aligns to the largest type
- Pointer arithmetic uses stride in computations
- `void *` has a stride of 1
- The void * type can be used for raw memory manipulation