More Detailed C Topics

More on Structures

Structures

- Recall
  - Collection of items which may be of different types
- Analogous to
  - Records in Pascal
  - Classes (with just variables) in Java
- In C, structs
  - Use named, not structural, type equivalence
  - Untagged structs are each a different type

Struct Declaration/Definition

```
struct [ <optional tag> ] [ {
   <type declaration>;
   <type declaration>;
   ...
} ] [ <optional variable list> ];
```

- Struct declarations that have a **member list** in curly braces define a **new type**, specifically a struct type.
- If the optional tag is omitted it creates an unnamed struct type that’s different from every other struct type.

Struct Name Scopes

```
struct [ <optional tag> ] [ {

   <type declaration>;

   <type declaration>;

   ...

} ] [ <optional variable list> ];
```

- Filling in <optional variable list> often makes the declaration also a definition; the variables defined appear in the function’s name scope just like any other variable.
- Struct tags are in a **separately-scoped name space** from variables, i.e. a struct variable can have the same name as a struct tag without causing confusion
- Struct member names are in yet another name space local to the structure type, e.g. every structure type could have a member named “next”

Initialization

```
struct mystruct_tag {
     int myint;
     char mychar;
     char mystr[20];
   };

struct mystruct_tag ms = {42, 'f', "goofy"};
```

Question

```
struct mystruct_tag {
        int myint;
        char mychar;
        char mystr[20];
} ms;
ms.mystr = "foo";
```

A. The assignment is legal

B. The assignment is illegal because “foo” is a different size than ms.mystr

C. The assignment is illegal because “foo” is stored in the constant data section of memory

D. The assignment is illegal because it is trying to assign an array to an array

Today's Number:
46,186

Question

```
struct mystruct_tag {
           int myint;
           char mychar;
           char mystr[20];
      };

struct mystruct_tag ms = {42,'b',"Boo!"};
```

A. The initializer is legal because a character array can be initialized to a string as a special case ←

B. The initializer is illegal because “Boo!” is a different size than ms.mystr

C. The initializer is illegal because “Boo!” is stored in the constant data section of memory

D. The initializer is illegal because it is trying to assign an array to an array

Copying Structs

```
struct s {
   int i;
   char c;
} s1, s2;

s1.i = 42;
s1.c = 'a';
s2 = s1;
s1.c = ‘b’;
s2.i contains ?
42
s2.c contains ?
a
```

Note that assigning the structure just copied the bytes from one memory block to another. There is no connection between them.

Copying Structs

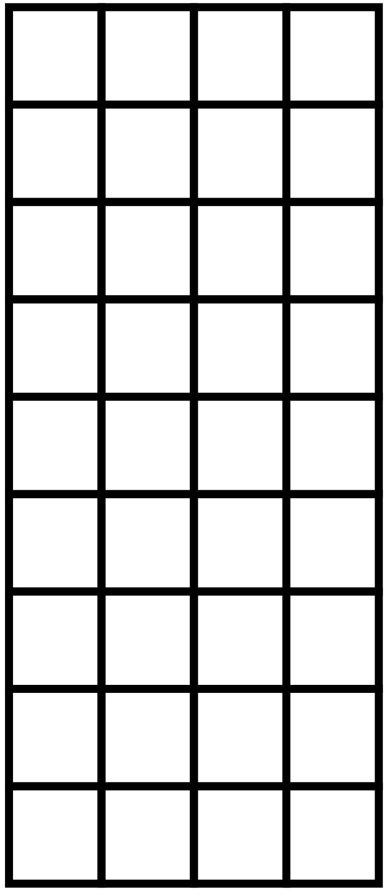
```
struct s {
   int i;
   char c[8];
} s1, s2;

s1.i = 42;
strcpy(s1.c, "foobar");
s2 = s1;
s2.i contains 42
s2.c contains:
'f','o','o','b','a','r','\0',??
```

Since we assigned s1 to s2, s2.c is going to contain exactly the same characters as are in s1.c, even including the unspecified character at s1.c[7]!

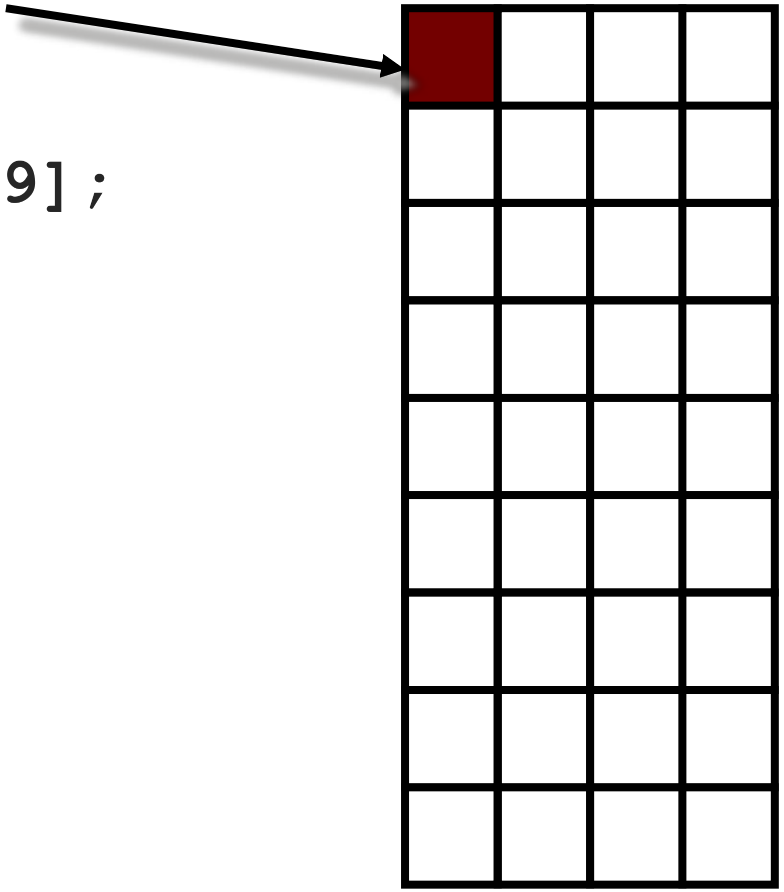
Where Do Members Get Stored in Memory?

```
struct {
   char mychar;
   int myint;
   char mystr[19];
} mystruct;
```



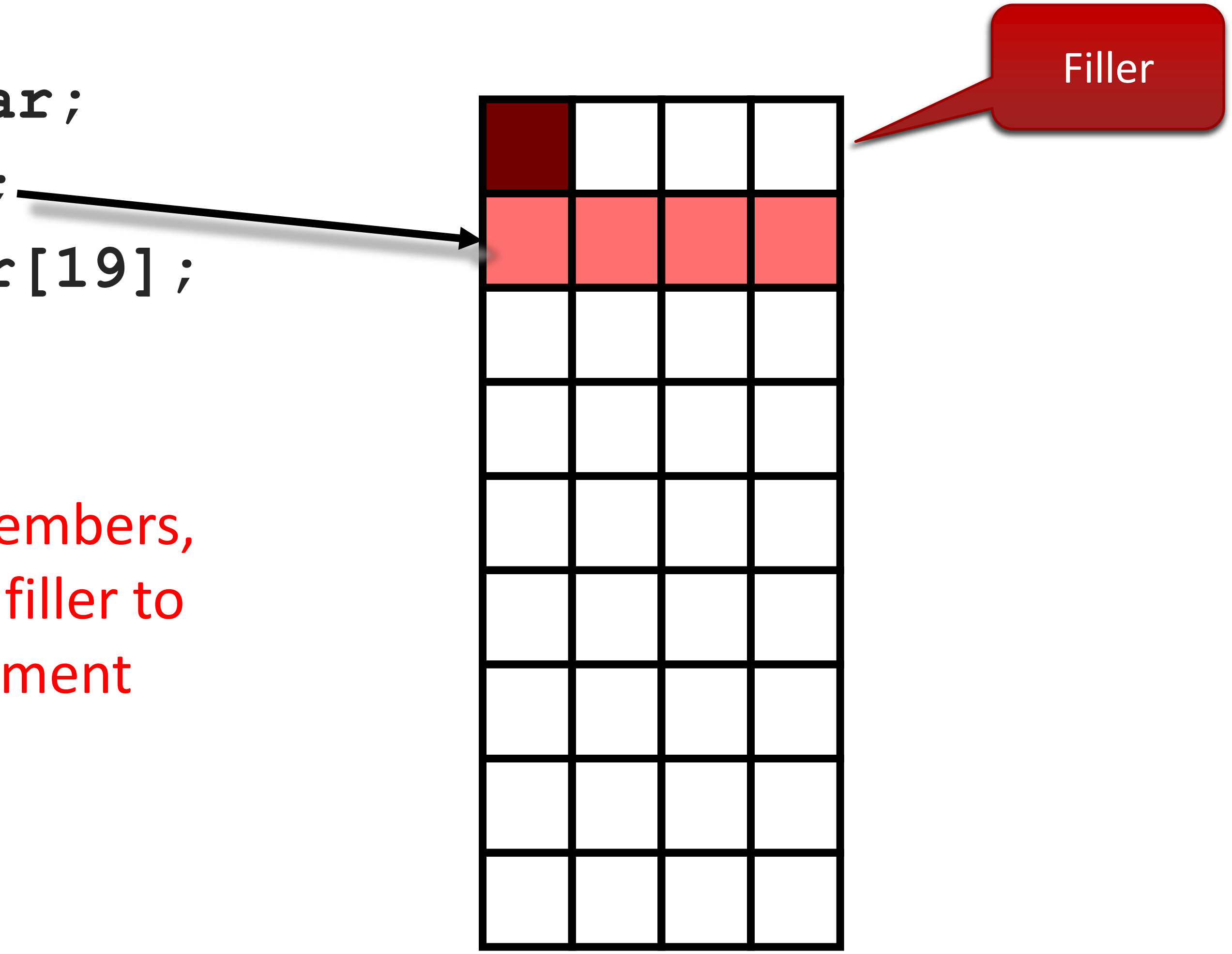
Where Do Members Get Stored in Memory?

```
struct {
   char mychar;
   int myint;
   char mystr[19];
} mystruct;
```



Where Do Members Get Stored in Memory?

```
struct {
   char mychar;
   int myint;
   char mystr[19];
} mystruct;
```

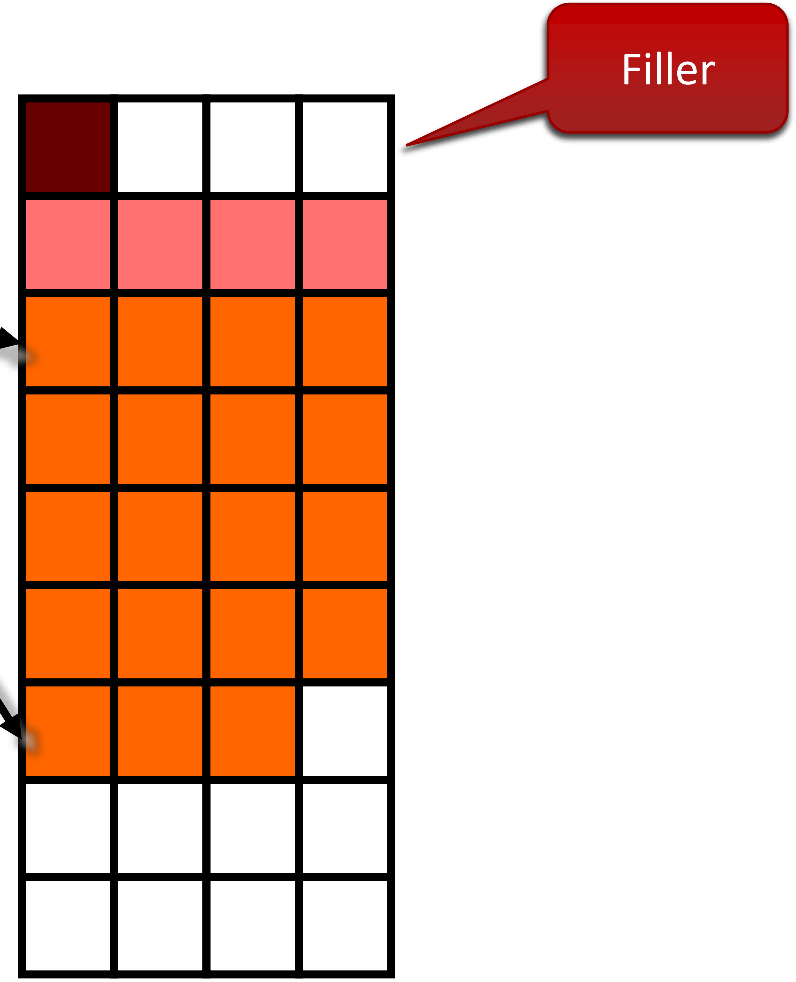


Filler

C can’t reorder members,
but it can add filler to
preserve alignment

Alignment Rules are Respected

```
struct {
   char mychar;
   int myint;
   char mystr[19];
} mystruct;
```

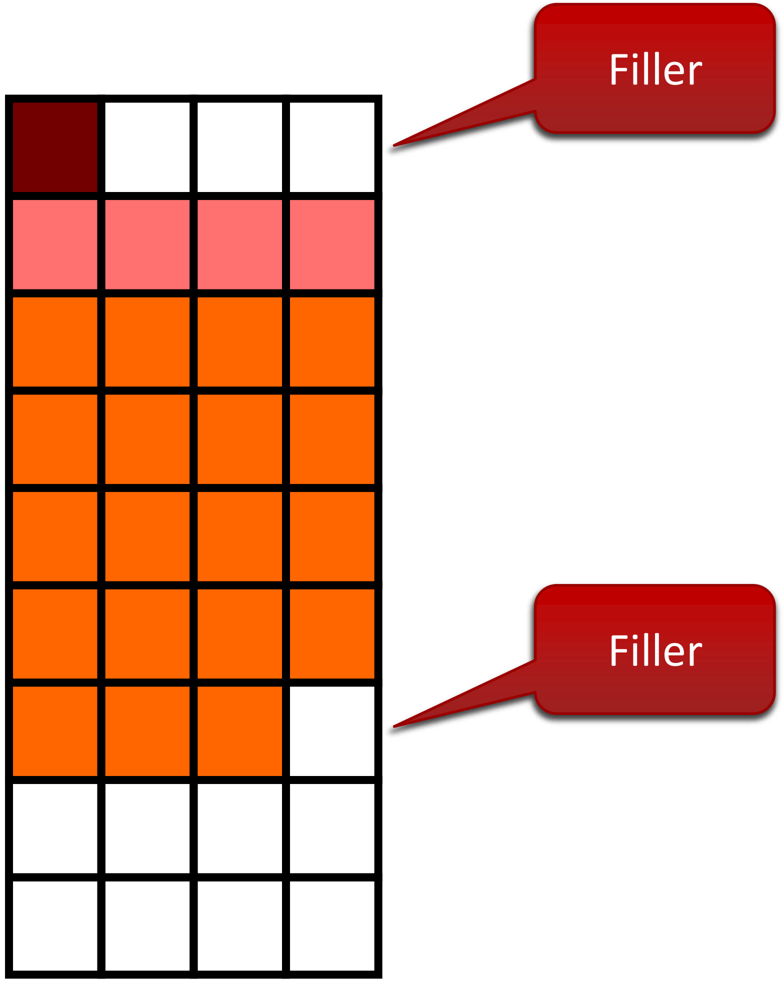


What is sizeof(mystruct)?

```
struct {
   char mychar;
   int myint;
   char mystr[19];
} mystruct;
```

Structs are usually filled out to meet the most stringent alignment of its members

sizeof(mystruct) = 28



We Still Use base+offset !

- The compiler keeps track of the offsets of each member in a table for each struct

member	offset
mychar	0
myint	4
mystr	8 (Sum of sizes of all previous elements incl filler)

Question: Assume "mystruct" is located at location 1000

What will be address of mychar, myint and mystr?

1000, 1004, 1008

Don’t Believe Me? Ask the Compiler!

```
#include <stdio.h>

struct {
       char mychar;
       int myint;
       char mystr[19];
} mystruct;

int main() {
       printf("Address of mystruct = %p\n", (void *)&mystruct);
       printf("Offset of mychar = %ld\n",
              (void *)&mystruct.mychar - (void *)&mystruct);
       printf("Offset of myint = %ld\n",
              (void *)&mystruct.myint - (void *)&mystruct);
       printf("Offset of mystr = %ld\n",
              (void *)mystruct.mystr - (void *)&mystruct);
       printf("Size of mystruct = %ld\n", sizeof(mystruct));
}
```

Don’t Believe Me? Ask the Compiler!

```
$ gcc sizes.c
$ ./a.out
Address of mystruct = 0x10ae74018
Offset of mychar = 0
Offset of myint = 4
Offset of mystr = 8
Size of mystruct = 28
```

Question?

- Can we say:

```
struct {
   char mychar;
   int myint;
   char mystr[19];
} mystruct;
mystruct.mystr[4] = ‘x’;
```

- Yes we can!

Base+offset again!

```
mystruct.mystr[4] = 'x';
```

- How do we get to the right character?
  - First find the address of the struct member
    - &mystruct + offset to mystr → &mystruct + 8
  - Then find the location within the struct member (if needed)
  - Using the type of the member, find the offset to the desired element
    - Offset of char mystr[4] → 4 * sizeof(char)
  - Add them: &mystruct + 8 + 4

- Example
  - mystruct is located at 2000
  - Address of mystr is 2000 + 8
  - So element 4 of mystr is offset by 4 * sizeof(char)
  - The address of mystruct.mystr[4] is...
  - ...2000 + 8+ 4 =

1. 2005
2. 2010
[✓ 3. 2012]
4. 2014

The Calculation in Detail

Base address of mystruct	&mystruct
Offset of mystr within structure	8
Offset of element 4 within mystruct.mystr	4
	&mystruct + 12

```
mystruct.mystr[4] = 'x';
```

- If mystruct is at address 2000, then mystruct.mystr[4] will be at address 2012.

Arrays of structs?

```
struct m astruct[25];

astruct[6].mystr[3] = ’y';
```

- How do we get to the right character?
  - First find the address of the struct member
    - &astruct + offset to astruct[6] + offset to mystr ➔ &astruct + 6*sizeof(struct m) + 8
  - Then find the location within the struct member (if needed)
  - Using the type of the member, find the offset to the desired element
    - Offset of char mystr[3] is 3 * sizeof(char)
  - Add them: &astruct + 6*28 + 8 + 3
- Example
  - astruct is located at 2000
  - Address of mystr is 2000 + 6*28 + 8
  - So element 3 of mystr is offset by 3 * sizeof(char)
  - The address of astruct.mystr[3] is...
  - ...2000 + 6*28 + 8+ 3 =

1. 2176
2. 2177
[✓ 3. 2179]
4. 2181

Detailed Calculation

Base address of mystruct	mystruct
Offset of element 6 of mystruct	6 * 28
Offset of mystr within structure	8
Offset of element 3 within mystruct.mystr	3
	&mystruct + 179

```
struct foo mystruct[25];

mystruct[6].mystr[3] = ’y';
```

- If mystruct is at memory location 2000, then mystruct[6].mystr[3] is at location 2179

Question

```
static struct {
        int n;
        char m[3];
        double p;
} s[12];
```

If *s* is stored at memory address 0x0e3c, where is *s[5].m[2]* stored?

A. 0xe91

[✓ B. 0xe92]

C. 0xe96

D. 0xea0

- Member offsets
  n: 0, m: 4, p: 8
- Struct size
  16
- Offset from s to &s[5]
  5*sizeof(s[0]) = 80 = 0x50
- Offset from &s[5] to s[5].m
  4 = 0x4
- Offset to from s[5].m to &s[5]m[2]
  2*sizeof(char) = 2 = 0x2
- Address
  0x0e3c+0x50+0x4+0x2 = 0xe92

Structures may

- Structures may
  - be copied or assigned
  - have their address taken with &
  - have their members accessed
  - be passed as arguments to functions
  - be returned from functions
- Structures may not
  - be compared