Continuing with C: Part 2

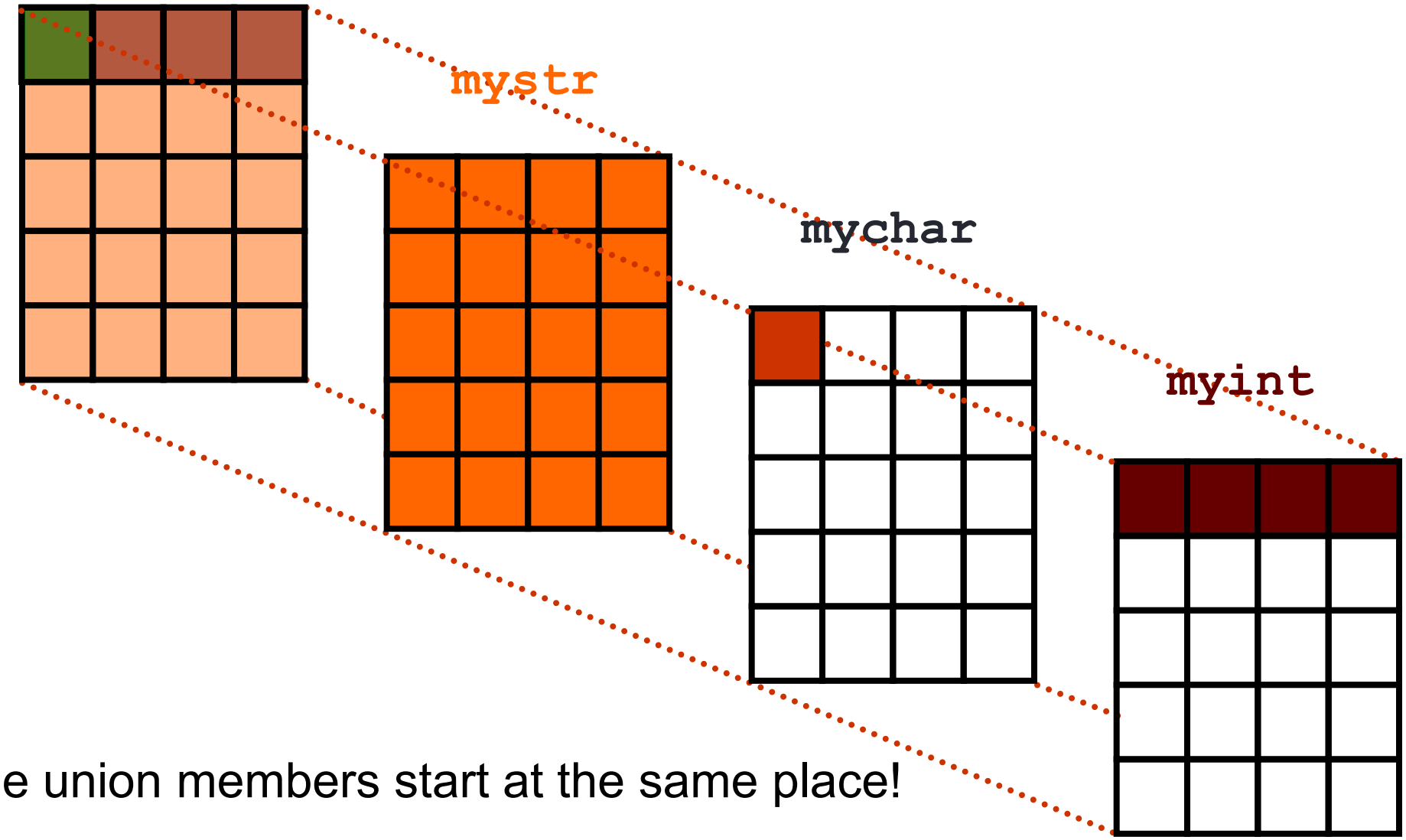
- Unions
- Function Pointers

Unions

```
union {
    int myint;
    char mychar;
    char mystr[20];
} myun;
```

- Looks like a struct
- and the access is the same as a struct
- So what's the difference?
- All of the members have an offset of zero – that’s it!

Unions



All of the union members start at the same place!

Unions

```
union {
   int myint;
   char mychar;
   char mystr[20];
} myun;
```

- &myun.myint == &myun.mychar == &myun.mystr[0]
- And sizeof(myun) is the size of the largest member

- Effectively all items in a union "start" at the same place
- But why?

Unions and base+offset

- Compiler keeps track of offsets into structure of each member in "some sort" of symbol table

member	offset
myint	0
mychar	0
mystr	0

Question: Assume "mystruct" is located at location 1000
What will be address of myint, mychar and mystr?

1000, 1000, 1000

Unions: raison d'être

- Suppose we want to store information about athletes
- For all we want
  - Name, JerseyNum, Team, Sport
- For football players we want
  - Attempts, yards, TDs, Interceptions, etc.
- For baseball players we want
  - Wins, Losses, Innings, ERA, Strikeouts, etc.
- For basketball players we want
  - Shots, Assists, Rebounds, Points, etc.

We code:

```
struct player {
   char name[20];
   char jerseynum[4];
   char team[20];
   int player_type;
   union sport {
      struct football {...} footbstats;
      struct baseball {...} basebstats;
      struct basketball {...} baskbstats;
   } thesport;
} theplayer;

theplayer.thesport.footbstats.tds = 3;
```

Unions

- Often used in implementing the polymorphism found in object-oriented languages

Unions may

Unions may

- be copied or assigned
- have their address taken with &
- have their members accessed
- be passed as arguments to functions
- be returned from functions
- be initialized (but only the first member)

Unions may not

- be compared

Function Pointers

If we want a function pointer

- To store the address of a variable we use a pointer (e.g. `int *ip;`)
- Same is true for holding the address of a function:

```
int fi(void);          /* Function that returns an int  */
int *fpi(void);        /* Function that returns a pointer
                        * to an int                      */
int (*pfi)(void);      /* pfi is a pointer to a function
                        * returning int!  */
```

Recall

Using it...

```
int fi(void);       /* Function that returns an int   */
int *fpi(void);     /* Function that returns a        */
                    /* pointer to an int              */
int (*pfi)(void);   /* Declaring pfi to be a          */
                     * pointer to a function!         */
pfi = fi;           /* Legal assignment               */
pfi = fi();         /* NO NO NO NO NO NO NO NO NO NO  */
```

- Notice similarity to

```
int ia[10];
int *ip;
ip = ia;
```

But what good is a function pointer?

- Say you are writing a general purpose sorting function.
- You want it to be able to sort anything
  - Numbers
  - Strings
  - Structs, Unions, and other stuff
- Obviously comparing numbers, strings, and other stuff calls for at least two different techniques
- What if we write functions that do the comparison we need
  - A function to compare numbers
  - A function to compare strings
  - A function to compare other stuff

But what good is a function pointer?

- Now when we call the function to do the sorting we pass in a pointer to the appropriate function for the type of data we have!

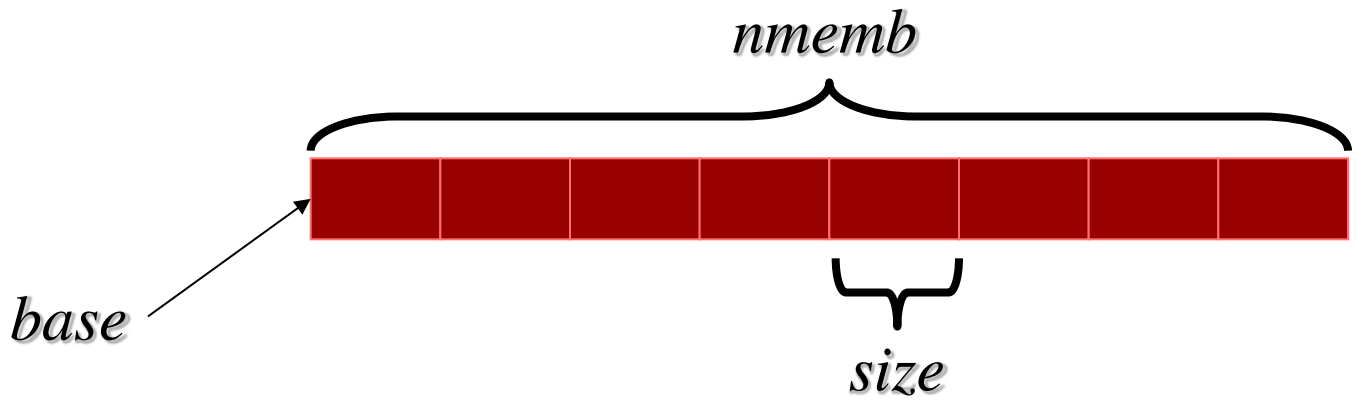
- "But wait," I hear you say, "It would be easier to write my own sorting function!"

Qsort(

```
qsort(3)                                                    qsort(3)

NAME
       qsort - sort an array

ANSI_SYNOPSIS
       #include <stdlib.h>
       void qsort(void * base, size_t nmemb, size_t size,
           int (* compar)(const void *, const void *) );
```



DESCRIPTION

qsort sorts an array (beginning at base) of nmemb objects. size describes the size of each element of the array.

You must supply a pointer to a comparison function, using the argument shown as compar. (This permits sorting objects of unknown properties.) Define the comparison function to accept two arguments, each a pointer to an element of the array starting at base. The result of (*<[compar)>> must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where ``less than'' and ``greater than'' refer to whatever arbitrary ordering is appropriate).

The array is sorted in place; that is, when qsort returns, the array elements beginning at base have been reordered.

```
qsort(3)                                                    qsort(3)

RETURNS
       qsort does not return a result.

PORTABILITY
       qsort  is required by ANSI (without specifying the sorting
       algorithm).

SOURCE
       src/newlib/libc/stdlib/qsort.c
```

QSort Demo

```
#include <stdlib.h>

void qsort (

        void * base,

        size_t nmemb,

        size_t size,

        int (* compar)(const void *, const void *)

);
```

The result of "compar" must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where "less than" and "greater than" refer to whatever arbitrary ordering is appropriate).

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int compar_ints(const void *pa, const void *pb) {
   return *((int *)pa) - *((int *)pb);
}

int compar_strings(const void *ppa, const void *ppb) {
   return strcmp( *((char **)ppa) , *((char **)ppb) );
}
```

```
# define MAX 100
int main(int argc, char **argv) {
    char *strings[] = {"dec", "sun", "ibm", "apple", "hp",
    "ti", "univac"};
    int i, s;
    int a[MAX];
    if(argc == 2 && *(argv[1]) == 'a') {
        s = sizeof(strings)/sizeof(strings[0]);
        qsort(strings, s, sizeof(strings[0]),
              compar_strings);
        for(i = 0; i < s; i++) {
              printf(" %s", strings[i]);
        }
        printf("\n");
    }
    else...
```

```
    else {
        for(i = 0; i < MAX; i++) {
                a[i] = rand() % 100;
                printf(" %d", a[i]);
        }
        printf("\n\n");
        qsort(a, MAX, sizeof(int),
                                compar_ints);
        for(i = 0; i < MAX; i++)            {
                printf(" %d", a[i]);
        }
    }
    return 0;
}
```

7th Ed Unix Device Driver Table

Major, minor device numbers

```
$ ls -l /dev
total 0
crw-------  1 dan   staff      0,   0 Mar 13 05:15 console
crw-rw-rw-  1 root  wheel      3,   2 Mar 13 13:59 null
crw-------  1 root  wheel     11,   0 Feb  9 15:00 pf
crw-rw-rw-  1 root  tty       15,  16 Mar 13 13:59 ptmx
crw-rw-rw-  1 root  wheel      5,   0 Feb  9 15:00 ptyp0
crw-rw-rw-  1 root  wheel      5,   1 Feb  9 15:00 ptyp1
crw-rw-rw-  1 root  wheel     14,   0 Feb  9 15:02 random
crw-r-----  1 root  operator   1,   0 Feb  9 15:00 rdisk0
crw-r-----  1 root  operator   1,   4 Feb  9 15:00 rdisk1
crw-r-----  1 root  operator   1,   7 Feb  9 15:00 rdisk2
crw-rw-rw-  1 root  wheel      2,   0 Feb 23 10:35 tty
crw-rw-rw-  1 root  wheel      4,   0 Feb  9 15:00 ttyp0
crw-rw-rw-  1 root  wheel      4,   1 Feb  9 15:00 ttyp1
crw-rw-rw-  1 root  wheel      4,   2 Feb  9 15:00 ttyp2
crw-rw-rw-  1 root  wheel      3,   3 Feb  9 15:00 zero
```

Defining the Table Struct

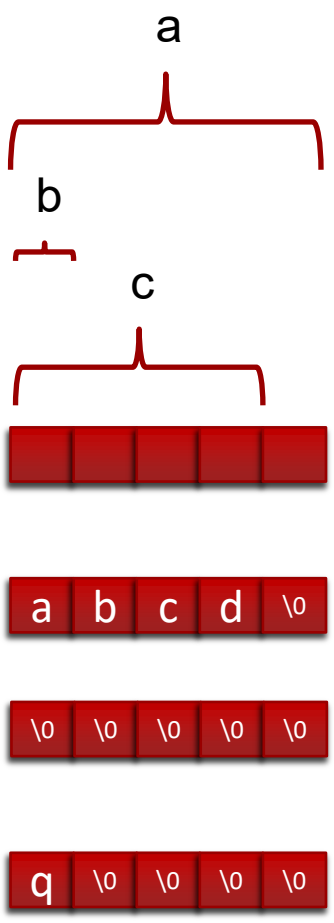
```
extern struct cdevsw
{
    int (*d_open)();
    int (*d_close)();
    int (*d_read)();
    int (*d_write)();
    int (*d_ioctl)();
    int (*d_stop)();
    struct tty *d_ttys;
} cdevsw[];
```

Major device number is the index to this table
Minor device number is passed on to the function

```
struct  cdevsw  cdevsw[] =
{
    klopen, klclose, klread, klwrite, klioctl, nulldev, 0,/* console = 0 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* pc = 1 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* lp = 2 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dc = 3 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dh = 4 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dp = 5 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dj = 6 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dn = 7 */
    nulldev, nulldev, mmread, mmwrite, nodev, nulldev, 0,/* mem = 8 */
    nulldev, nulldev, rkread, rkwrite, nodev, nulldev, 0,/* rk = 9 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rf = 10 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rp = 11 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* tm = 12 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* hs = 13 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* hp = 14 */
    htopen, htclose, htread, htwrite, nodev, nulldev, 0,  /* ht = 15 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* du = 16 */
    syopen, nulldev, syread, sywrite, sysioctl, nulldev, 0,/* tty = 17 */
    nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rl = 18 */
    0
};
```

Questions?

Question



Given execution of the following code, what is the value of m.a?

```
union m {
    char a[5];
    char b;
    int c;
} m;
strcpy(m.a, “abcd”);

m.c = 0;

m.b = ‘q’;
```

\0

A. m.a contains “q”
B. m.a contains “abcd”
C. m.a contains an empty string
D. m.a contents are unknown

Question

What kind of value will you typically find in a pointer to a function?

A. The address of the function arguments on the stack
B. The address of the first word of the function code
C. An integer describing the success or failure of the function
D. A string containing the function's name