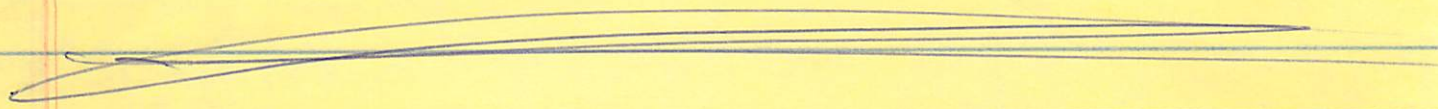
ECE198KL

L3CoPA
15 April 2013

generalizing sort with function pointers
hierarchies of ~~class~~ structures
    linked list example (smaller sentinel)
"library" example
functions on "base" types
classes

— see ~shared/io example
(update to get the new version)

read "Week 14" notes (p. 57 –) in 198 lab manual



delay PA4? → Tues

combine 12&13? NO

didn't grade exam yet--

```
/*
 * sort_lines -- performs an insertion sort on an array of strings (char*'s)
 *    inputs: lines -- an array of strings
 *            n_lines -- the number of lines in the array
 *    outputs: lines -- returned in sorted order
 *    returns: nothing, but changes array in place
 */
static void
sort_lines (char* lines[], int n_lines)
{
    int sorted;         /* outer loop index; number of lines sorted       */
    char* current;      /* current line being placed into sorted subarray */
    int index;          /* inner loop index for placing current line      */

    /* We start with a subarray of length 1 already sorted, so
       we need iterations to sort each larger subarray from length 2
       up to the full length of the array. */
    for (sorted = 2; n_lines >= sorted; sorted++) {

        /* Keep track of the line being moved into position. */
        current = lines[sorted - 1];

        /* Move other array entries aside to make room for "current." */
        for (index = sorted - 1; 0 < index; index--) {

            /* Check the order of "current" against the line before
               that at index.  If it's still smaller, move the line
               and continue the loop.  Otherwise, we've found the place
               to which we must move "current." */
            if (strcmp (current, lines[index - 1]) < 0) {
                lines[index] = lines[index - 1];
            } else {
                break;
            }
        }

        /* Store current in the right place. */
        lines[index] = current;
    }

    /* No return value. */
}
```

```
/*
 * string_less_than -- is string one "less than" string two (ASCII order)
 *    inputs: one -- pointer to first string (that is, to the char*)
 *            two -- pointer to second string (that is, to the char*)
 *    outputs: nothing
 *    returns: 1 if the first string comes before the second in ASCII order,
 *             or 0 otherwise
 */
static int
string_less_than (void* one, void* two)
{
    return (strcmp (*(char**)one, *(char**)two) < 0);
}
```

```
/*
 * generic_sort_lines -- performs an insertion sort on an array
 *    inputs: base -- start of array
 *            n_elts -- the number of elements in the array
 *            size -- array element size in bytes
 *            is_smaller -- function to call to compare two elements;
 *                          must return non-zero if first element is smaller
 *                          than second element, or zero otherwise
 *    outputs: base -- array returned in sorted order
 *    returns: 1 on success, 0 on failure
 */
static int
generic_sort_lines (void* base, int n_elts, int size,
                    int (*is_smaller) (void* one, void* two))
{
    char* array = base; /* array pointer (used for pointer arithmetic)       */
    void* current;      /* current element being placed into sorted subarray */
    int sorted;         /* outer loop index; number of lines sorted          */
    int index;          /* inner loop index for placing current line         */

    /* We first allocate a space to hold one element. */
    current = malloc (size);
    if (NULL == current) {
        return 0;
    }

    /* We start with a subarray of length 1 already sorted, so
       we need iterations to sort each larger subarray from length 2
       up to the full length of the array. */
    for (sorted = 2; n_elts >= sorted; sorted++) {

        /* Keep track of the line being moved into position. */
        memcpy (current, array + (sorted - 1) * size, size);

        /* Move other array entries aside to make room for "current." */
        for (index = sorted - 1; 0 < index; index--) {

            /* Check the order of "current" against the line before
               that at index.  If it's still smaller, move the line
               and continue the loop.  Otherwise, we've found the place
               to which we must move "current." */
            if (is_smaller (current, array + (index - 1) * size)) {
                memcpy (array + index * size,
                        array + (index - 1) * size, size);
            } else {
                break;
            }
        }

        /* Store current in the right place. */
        memcpy (array + index * size, current, size);
    }

    /* Get rid of the temporary space. */
    free (current);

    /* Return success! */
    return 1;
}
```

L 36 P1

L36P2

1 on success, 0 on failure

caller can pass any pointer

pointers to elements

can't use void* with +/-

```
static int
generic_sort_lines (void* base, int n_elts, int size,
                    int (*is_smaller) (void* one, void* two))
{
    char* array = base; /* array pointer (used for pointer arithmetic)        */
    void* current;      /* current element being placed into sorted subarray */
    int sorted;         /* outer loop index; number of lines sorted          */
    int index;          /* inner loop index for placing current line         */

    /* We first allocate a space to hold one element. */
    current = malloc (size);
    if (NULL == current) {
        return 0;
    }

    /* We start with a subarray of length 1 already sorted, so
       we need iterations to sort each larger subarray from length 2
       up to the full length of the array. */
    for (sorted = 2; n_elts >= sorted; sorted++) {

        /* Keep track of the line being moved into position. */
        memcpy (current, array + (sorted - 1) * size, size);

        /* Move other array entries aside to make room for "current." */
        for (index = sorted - 1; 0 < index; index--) {

            /* Check the order of "current" against the line before
               that at index.  If it's still smaller, move the line
               and continue the loop.  Otherwise, we've found the place
               to which we must move "current." */
            if (is_smaller (current, array + (index - 1) * size)) {
                memcpy (array + index * size,
                        array + (index - 1) * size, size);
            } else {
                break;
            }
        }

        /* Store current in the right place. */
        memcpy (array + index * size, current, size);
    }

    /* Get rid of the temporary space. */
    free (current);

    /* Return success! */
    return 1;
}


static int
string_less_than (void* one, void* two)
{
    return (strcmp (*(char**)one, *(char**)two) < 0);
}
```

avoid limiting element size

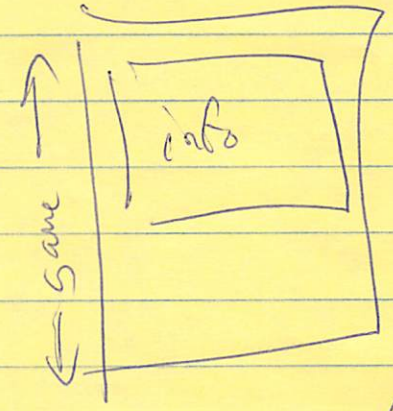
"(*is_smaller)" also works

element is char*, so pointer to element is char**

Hierarchies of Structures

In Prog 11. we used one structure as a field of another

```
struct save_game_t {
    save_info_t info;
    :
};
```



The "info" just happened to be at the start of the "game" structure.

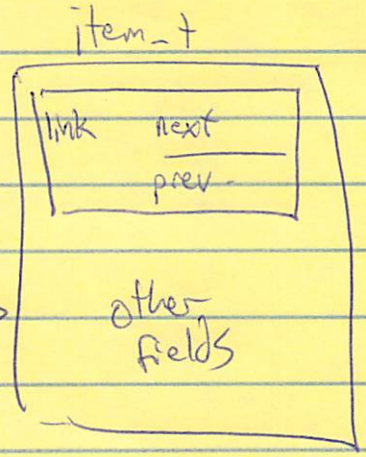
We can use that idea to create hierarchies of different types of structures

- Common initial fields
- different overall

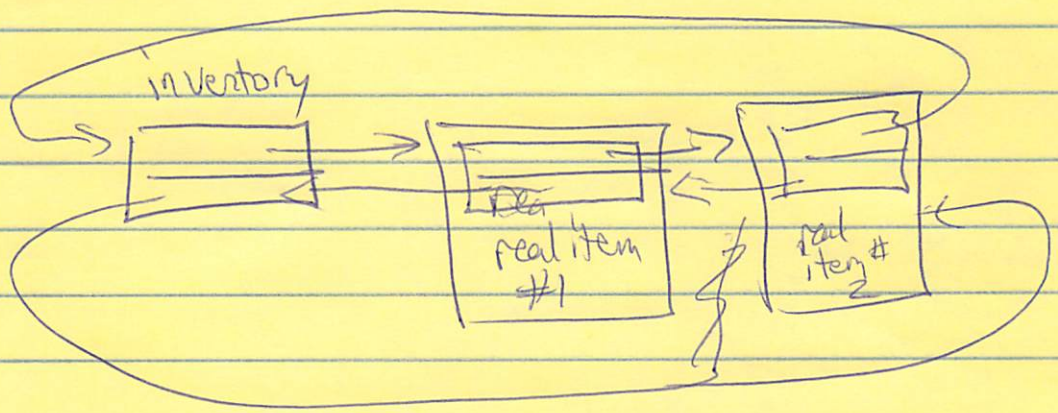
Example: just the pointers in a sentinel (Prog 10)

```
struct double_list_t {
        double_list_t* next;
        double_list_t*  prev;
};
```

```
struct item_t {
        double_list_t links;
        item fields
};
```



now "inventory" can be a double_list_t

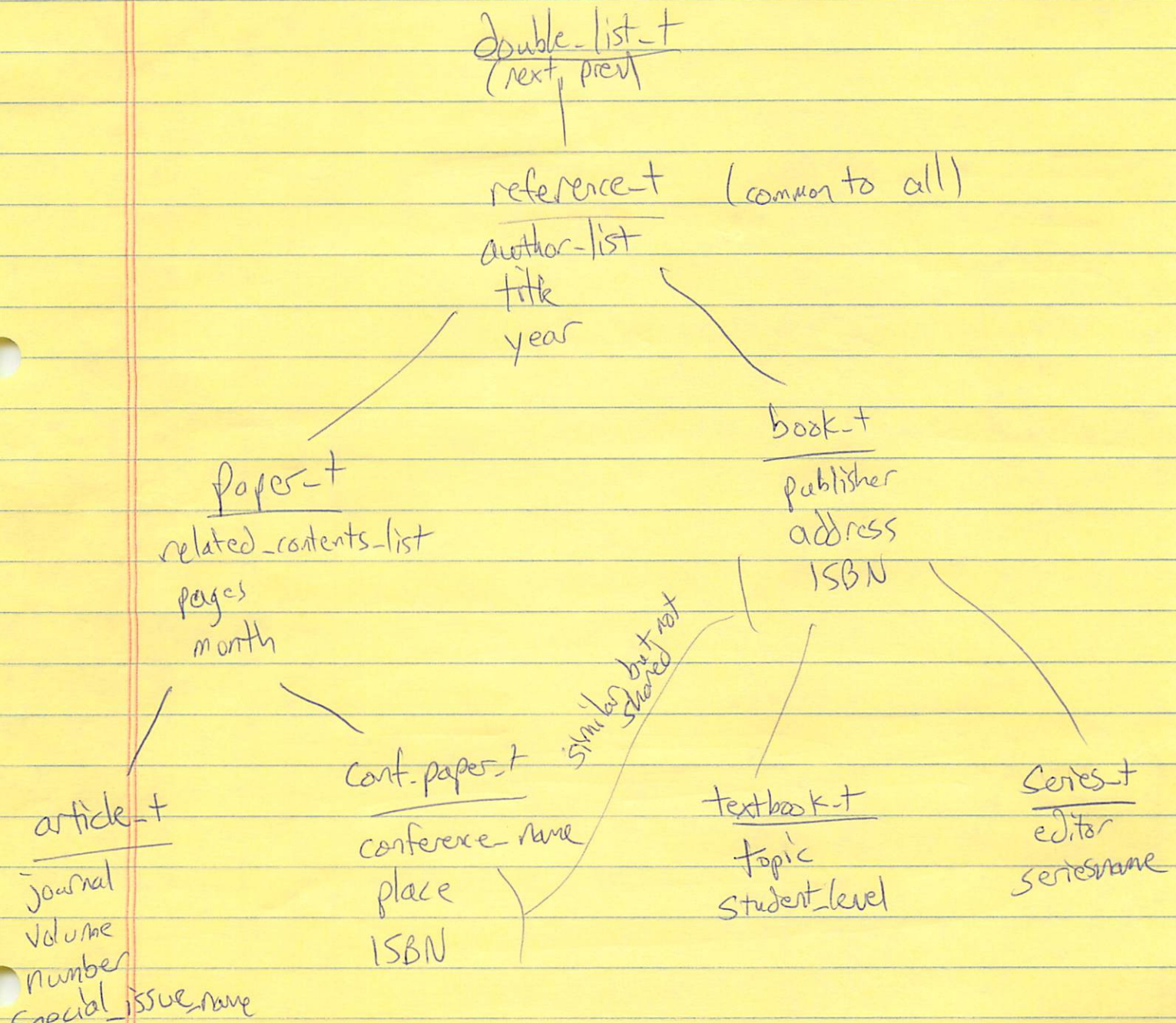


\+ saves a little space

\- need lots of explicit pointer casts in C
(compiler cannot check these!)

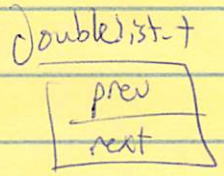
more useful: combine a variety of types

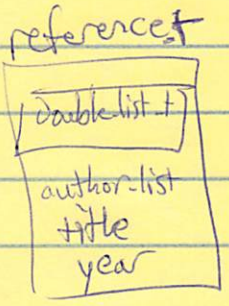
example: citation / bibliographic database

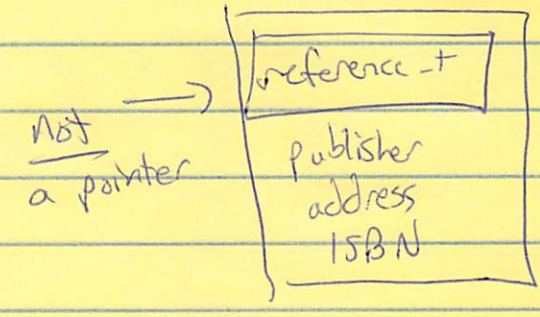


(omitting typedefs)

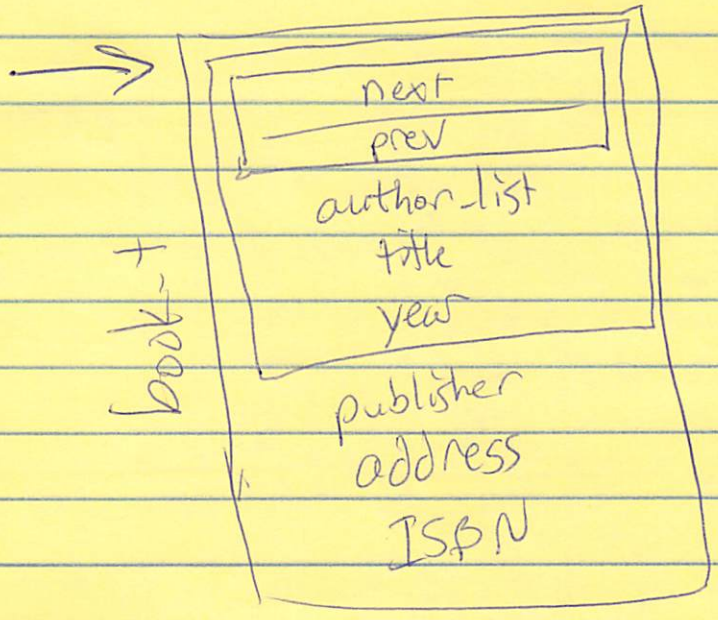
```
struct double-list_t {
        double-list_t* next;
        double-list_t* prev;
};

struct reference_t {
        double_list_t link;
        char* author_list;
        char* title;
        int year;
};

struct book_t {
        reference_t ref;
        char* publisher;
        char* address;
    unsigned long long ISBN;
};
```







NOTICE
any pointer to a book_t is ALSO a pointer to a reference_t AND a pointer to a double-list_t.



void print_citation (reference_t* ref);

Let's write a loop that prints citations for all items in a library...

static double_list_t library; ← a sentinel

```
double_list_t* elt;
for( elt= library.next; &library != elt;
                elt = elt->next) {
    print_citation ((reference_t*) elt);
}
```

ok if all of our list elements are reference_t's.