String library functions, cont.

- Comparing strings:
  ```c
  int strcmp(char s1[], char s2[]);
  ```
  - works just the same as `s1.compareTo(s2)` did in Java:
    - returns negative number if `s1` is less than `s2`
    - returns positive number if `s1` is greater than `s2`
    - returns 0 if `s1` and `s2` match character for character

  - Example:
    ```c
    if (strcmp(str1, "hello") == 0)
        printf("str1 is "hello"");
    ```
A possible `strcmp()` implementation

```c
int strcmp(char s1[], char s2[]) {
    int i;
    for (i = 0; s1[i] && s2[i]; i++)
        if (s1[i] != s2[i])
            break;
    return s1[i] - s2[i];
}
```

• Notice the return statement subtracts characters; remember that `char` is an integer type

String library functions, cont.

• Copying strings:

```c
void strcpy(char dst[], char src[]);
```

- copies the string in `src` to `dst`
- it is up to the programmer to ensure that `dst` is an array with enough characters to hold the string
  • being lazy with this function can result in buffer overflows
- Example:

```c
char str[] = "cherry";
char str2[10] = "milkshake";
strcpy(str2, str);
```

\[ \text{str2} \begin{array}{c} c h e r r y \ \backslash 0 k e \ \backslash 0 \end{array} \]

A possible `strcpy()` implementation

```c
void strcpy(char dst[], char src[]) {
    int i = 0;
    while (src[i]) {
        dst[i] = src[i];
        i++;
    }
    dst[i] = '\0';
}
```

• What expression gives the minimum size of the array `dst` (to ensure safe execution)?
Structures

- Like arrays, hold multiple items
- Items need not be of the same type
- Items referred to by field names, not numerical indices
- You can assign the value of another structure to a structure
- You cannot use == or !=
- Similar to a Java class with all public fields and no methods

Creating structures

- Example:
  ```c
  struct employee {
    int id_number;
    char last_name[10];
    char first_name[10];
    double salary;
  } emp1, emp2;
  ```
  Declares two variables (emp1 and emp2) of type struct employee
- `employee` is called the tag of these two structs
  - used to differentiate between different kinds of structs

Structure declarations

- More formally, this is the syntax for declaring structures (or structure types):
  ```c
  struct tag { member-list } variable-list;
  ```
- Omitting `variable-list` creates a new type
- Omitting `member-list` (and `{}`) declares variables of an existing struct type
- Omitting `tag` means you create a unique type for the variables listed
  - even if `member-lists` are the same
  - prevents use of those structs as function arguments

Accessing fields of a structure

- Dot operator:
  ```c
  struct point {
    int x, y;
  };
  ```
  ```c
  struct point p1, p2, points[5];
  p1.x = 17;
  p2.y = 22;
  points[0].x = 13;
  points[0].x++;
  ```
### The `typedef` keyword

- You can give types new names
  - eases readability and maintainability
- `typedef existing-type new-name;`
  - the type may be created along with the `typedef` usage, as we'll see with structures
- `typedef double Dollars;`  
  `Dollars x, y = 1.25;`
  - now you know that `x` and `y` shouldn't be assigned values like `sqrt(15)`
- Using caps to start `typedef`'d names helps set them apart from other types

### Combining `typedef` and `struct`

- Combining the two keywords:
  ```c
  typedef struct {
    int i;
    char ch;
  } Ex_struct;
  ...
  Ex_struct a[10], b;
  ```
- Structure definitions (either form) should be placed in header files if the structures are used across multiple files

### Structure storage

- How much space does a structure use in memory?
  ```c
  struct one {
    double b;
    int a;
  } s1;
  ```
  ```c
  printf("%d\n", sizeof(s1));
  ```
- Assuming `ints` use 4 bytes and `doubles` 8 bytes each, prints "12"

### Structure storage, cont.

- But due to alignment issues, things aren't always that simple:
  ```c
  struct two {
    char a;
    int b;
  } s2;
  ```
- `ints` must begin at 4-byte boundaries, so `s2` must be 8 bytes, not 5.
  ```c
  printf("%d\n", sizeof(s2));
  ```
- To minimize unused space, order fields from longest to shortest
Assigning and comparing structures

- Each field is copied for an assignment
  ```c
  struct ex_struct a, b;
  ...
  a = b;
  ```
- Is `a == b` true now? Two issues:
  - `a` and `b` are still separate objects in memory
  - Can't just compare bits - what if there's unused space?
- Because it doesn't make sense to do these types of comparisons, the `==` won't compile

Structure initialization

- Much like array initialization
- The items listed in the initializer are assigned to the fields in order
- Zeroes used to fill uninitialized fields when an initializer is used
- Example:
  ```c
  typedef struct {
    int i;
    char ch;
    double d;
  } Ex_struct;
  Ex_struct a = {4, 's', 3.5};
  Ex_struct b = {5, 'g'};
  ```

Nested structure example

```c
/* a Section contains a number like 0101, and
 * how many students are enrolled */
typedef struct {
  int number;
  int num_students;
  int start_time;
} Section;

/* a Course contains a number like 313,
 * and two Sections */
typedef struct {
  int course_number;
  Section section1, section2;
} Course;

Section s = {101, 10, 1400};
Course c = {213, {}, {201, 30, 1200}};
...
c.section1 = s;    /* referring to a whole Section */
c.section2.num_students = 29;   /* referring to one field of a 
   Section in the Course */
```
Aside: parameters are variables, too!

- Because arguments are passed and returned by value, you can use the parameters as variables:

```c
Section add_students(Section sec, int students_to_add) {
    sec.num_students += students_to_add;
    return sec;
}
Section s = { 0101, 10, 1400 }, t;
...t = add_students(s, 26);
```

Unions

- Look much like structures
- But all fields share the same memory space, so are only as large as largest field
- Only one field valid at a time

```c
typedef union {
    int i;
    double d;
} Number;
...Number a, b;
a.i = 2;
b.d = 3.14159;
printf("%d\n", b.i);
```