Lecture 13: Data structures in C
C structures

- We are going to discuss defining more complex data structures
  - Structures are “user-defined types”
- Data structure design is a very important part of programming
  - We learned many program design concepts
    - Use of control flow structures such as loops and if statements
    - Use of functions and recursion
  - Data structures design is as important as any of these concepts
C structures

• For representing the data in our program, so far we used
  – basic data types
    • \texttt{int} \ a; \texttt{char} \ c; \texttt{double} \ x;
  – arrays
    • \texttt{int} \ a[10][10];
  – pointers
    • \texttt{int} \ *b;
• Sometimes we need a richer data representation
  – For example:
    • For each student, we may want to keep the following data: perm number, first name, last name
    • So each data item consists of an integer value and two string values
C structures – some basics

- Structures are user-defined types with multiple data fields – e.g., define structure to hold a char and a double:

  ```c
  struct example {
    char c;
    double d;
  }; /* the semicolon is mandatory */
  ```

- Create objects and assign pointers like any other type:

  ```c
  struct example e, *ep = &e;
  ```

- Effect of assignment statement just like other data types:

  ```c
  struct example e2 = e;
  /* makes copy of whole object */
  struct example *ep2 = ep;
  /* now both ep and ep2 point at e */
  ```
More structure basics

- Access internal fields with the dot ‘.’ operator
  - By using the structure’s name: `e.d = 2.5;`
  - Or the pointer:
    `(*ep).c = 'a'; /* parenthesis needed */`

- Easier to use the arrow ‘-＞’ operator for pointers
  - `ep-＞c = 'a';`

- Can initialize all fields in one step:
  - `struct example e2 = { 'c', 97.14 };`

- Note – size of structure >= sum of field sizes:
  - `sizeof e >= sizeof(char) + sizeof(double)`
We can give names to types

- We can give names to complicated types to make it easier to use them later on in declarations.
- Precede a declaration with `typedef` to give a new name to a type.
- Useful for complicated type declarations.
We can give names to types

- Example, we define a new type "structure temperature"

```c
struct temperature {
    char unit; /* e.g., 'F' or 'C' */
    double value;
};
```

- Now we can give it a new name (like a nickname) "Temperature":

```c
typedef struct temperature Temperature;
Temperature t, *tp;
```

- Can even use one defined type to define another:

```c
typedef Temperature *TemperaturePointer;
TemperaturePointer tp; /* tp same as above */
```
Why need user-defined types?

• Answer: to *represent* more complicated things than numbers, characters and other primitives

• For example, a structure like the one in our example could represent temperatures for particular units:

```c
struct temperature {
    char unit; /* e.g., 'F' or 'C' */
    double value;
};
struct temperature bodyNormal = {'F', 98.6};
```

• Example from the book: *struct hurricane hurricane.c, and hurricane2.c*
Passing structures to functions

• Actually passing a copy of the whole structure – just like assignment statement
  – Unlike passing array name (which passes the pointer value, not the contents)

• Example: hurricane3.c

• Often want to pass pointers instead
  – Avoids copy costs – some structures very big
  – Gives the function access to internal fields of the original object (by \rightarrow)
  – Careful though: sometimes it’s not what you want
Returning structures

• Function get_info() for example (end of Section 7.2)
  struct tsunami get_info(void) {...}
  –Returns temporary struct (calling program copies it by assignment)

• Often return pointers too:
  struct tsunami* aFunction(void) {...}
  –But only okay if dynamic memory allocation used – do not return a pointer to a local structure variable!

• BTW: notice how defined types aid readability:
  –First: typedef struct tsunami Tsunami;
  –Above becomes:
    Tsunami* aFunction(void) {...}
Collections of structures

• Arrays of structures – an alternative to “parallel arrays”
  – Mostly handle like all other array types

Temperature temps[7];
   /* store a week’s worth */
temps[0].value = 74;
   /* sets field inside first element */

• Example hurricane4.c

• Also note: you can have arrays in structures

• Often use array of pointers to structures (with malloc)