CSC 357 Lecture Notes Week 2, Part 3

Pointers, Arrays, and Structs, Cont’d
I. **Structures (K&R chapter 6).**

A. We’ve seen structs in lecture and lab examples.

B. A struct is set of variables collected under a common name; the vars are *fields* of the struct.

C. Compared to Java, a struct is equivalent to a class with all public data fields and no method defs.
II. Basics of structures (K&R Section 6.1).

A. The syntax of a structure declaration is

```
struct struct-tag {
  fields
}
```

where `struct-tag` is a name, and `fields` are variable declarations; the tag is optional.
Basics, cont’d

B. Structure fields are also referred to as *members*; the two terms are synonymous.

C. Here’s a simple example:

```c
struct point {
    int x;
    int y;
}
```
Basics, cont’d

D. A struct declaration defines a type, and so can be used directly to declare struct-type variables.

1. I.e.,

    struct { ... } x, y, z;

    is syntactically analogous to

    int x, y, z;
Basics, cont’d

2. If a struct declaration contains a tag, then it can be used in subsequentdecls, as in

   struct point pt;

(but cleaner-looking naming is with typedef)
Basics, cont’d

E. Structs can be initialized in a declaration, as in

```c
struct point maxpt = {320, 200};
```
Basics, cont’d

F. Struct fields are accessed with ’.’ operator, as in

```c
pt.x = 10;
pt.y = 20;
printf("%d,%d", pt.x, pt.y);
```
Basics, cont’d

G. Nested `struct` defs, as in

```c
struct rect {
    struct point pt1;
    struct point pt2;
};
```
Basics, cont’d

H. If we declare

```c
struct rect screen;
then
```

```c
screen.pt1.x
```

refers to the x coordinate of the `pt1` field.
III. Structures and functions (K&R Section 6.2).

A. Legal operations on structs are assignment, address-of, and member access.

B. For large structs, passing a struct pointer as a parameter is more efficient.

C. Pointers to structs are also necessary when creating dynamically-linked data structures.
Structs and functions, cont’d

D. There are two notations for accessing the fields of a pointed-to struct, such as

```c
struct point *pp;
```

1. Expression `(*pp).x` accesses the `x` field.
2. Alternative equivalent notation is `pp->x`. 
IV. Arrays of structures (K&R Section 6.3).

A. Arrays of structs are an important working data structure in C.

B. For example, a very simple word-count table:
Arrays of structs, cont’d

```c
#define MAXWORDS 100

struct {
    char* word;
    int count;
} wordtab[MAXWORDS];
```
Arrays of structs, cont’d

C. Assuming the fields of the ith table element have been properly initialized:

```c
wordtab[i].word[j] = getchar();
wordtab[i].count++;
```
V. Pointers to structures (K&R Section 6.4).

A. When an array of structs is sparse, an array of pointers to structs can be more efficient.

B. Consider the following declarations:
Pointers to structs, cont’d

```c
struct wordcnt {  
    char* word;  
    int count;  
};

struct wordcnt wordtab[MAXWORDS];  
struct wordcnt* wordtabp[MAXWORDS];
```
Pointers to structs, cont’d

C. Before any elements of `wordtabp` have been set, `wordtabp` is half as big as `wordtab`.

D. When contents of a table may be partially unfilled, using struct pointers is advantageous.
VI. Self-referential structures (K&R Section 6.5).

A. C allows a struct field to be declared as a pointer to the struct itself.

B. E.g.,
Self-referential structs, cont’d

```c
struct tnode {
    char* word;
    int count;
    struct tnode* left;
    struct tnode* right;
};
```
Self-referential structs, cont’d

C. This is a recursive data type def.
VII. Table lookup (K&R Section 6.6).

A. This section of K&R defines a simple hash table.

B. Have a look.
VIII. Typedefs (K&R Section 6.7).

A. Typedef provides a convenient way to give a mnemonic name to a data type definition.

B. The typedef can be as simple as

```c
typedef int Length;
```

used in declarations like

```c
Length len, maxlen;
Length getLength(...);
```
Typedefs, cont’d

C. Typedefs also add readability to struct defs

```c
typedef struct {
    char* word;
    int count;
} WordCount;

WordCount wordtab[MAXWORDS];
WordCount* wordtabp[MAXWORDS];
```
Typedefs, cont’d

D. When non-recursive struct is typedef’d, the struct tag need not be present.

E. But for recursive types, tag must be present for self-referencing
typedef struct tnode {
    char* word;
    int count;
    struct tnode* left;
    struct tnode* right;
} TreeNode;

TreeNode* tree;
Typedefs, cont’d

F. The following equivalent-looking definition does NOT work:

typedef struct tnode {
    char* word;
    int count;
    TreeNode* left;  /* INVALID */
    TreeNode* right; /* INVALID */
} TreeNode;
IX. Unions (K&R Section 6.8).

A. A union var may hold values of different types.

B. Suppose we want a variable that can hold one of an int, double, string, or boolean.
Unions, cont’d

typedef union {
    int int_val;
    double double_val;
    char* string_val;
    unsigned char bool_val;
} GenericValue;
Unions, cont’d

C. Syntactically, unions are declared and accessed in precisely the same way as structs.

1. Union fields are accessed with ‘.’.

2. Pointer-to-union fields are accessed with ‘->’.
Unions, cont’d

D. The semantic difference is that a struct value contains all its data fields, whereas a union value contains one of its data fields.
Unions, cont’d

E. As explained on pages 147-148 of K&R, "It is the programmer’s responsibility to keep track of which type is currently stored in a union; ..."

F. For this reason, union types are often tagged to keep track of the current value.
Unions, cont’d

1. Union tags are frequently implemented with enums.

2. E.g.,
Unions, cont’d

typedef enum {
    INT, DOUBLE, STRING, BOOL
} ValueTag;

typedef struct {
    ValueTag tag;
    GenericValue val;
} TaggedGenericValue;
Unions, cont’d

3. Some example usage
Unions, cont’d

```c
void PrintTaggedGenericValue(TaggedGenericValue v) {
    switch (v.tag) {
    case INT:
        printf("%d0, v.val.int_val);
        break;
    case DOUBLE:
        printf("%f0, v.val.double_val);
        break;
    case STRING:
        printf("%s0, v.val.string_val);
        break;
    case BOOL:
        printf("%s0, v.val.bool_val ? "true" : "false"
    }
}
```
main() {
    TaggedGenericValue tval;

    tval.val.int_val = 10;
    tval.tag = INT;
    PrintTaggedGenericValue(tval);

    tval.val.bool_val = 0;
    tval.tag = BOOL;
    PrintTaggedGenericValue(tval);
}

4. The idea is that the union type appears in the context of a struct that has information indicating which union value is current.
X. Bit-fields (K&R Section 6.9).

A. Bit-fields provide access to individual binary bits in a word of memory.

B. Such access can save space, e.g., bool as bit.

C. Also provide direct access to hardware devices.

D. We’ll talk more about bit-fields later.
XI. **A culminating example.**

A. Attached code listings illustrate key concepts.

B. The commenting style is `doxygen`-compliant.

C. There is `doxygen`-generated doc online.
Culminating example, cont’d

D. PLEASE NOTE: There are some unanswered questions in the testing file `person-record-test.c`; these will be topics in next week’s lab.