C: Structures

Department of Computer Science
College of Engineering
Boise State University

August 25, 2017

Structures

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. The variables contained in a structure are known as its members.
- Use the `struct` keyword followed by an optional tag to declare a structure. For example, in the following declaration, `point` is the tag and `x, y` are members of the structure.

```
struct point {
    int x;
    int y;
};

struct point pt;
```

- A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct point pt = { 100, 200 };
```

The Dot Operator

- A member of a particular structure is referred to in an expression by a construction of the form structure-name.member
- The structure member operator . (dot) connects the structure name and the member name. For example:

```
printf("%d,%d", pt.x, pt.y);
```

- Structures can contain references to other structures (similar to a class in Java). For example:

```
struct rect {
    struct point bottomLeft;
    struct point topRight;
};

struct rect screen;
```

Then `screen.topRight.x` refers to the `x` coordinate of the `topRight` member of screen.

Structures as function arguments

- Structures are passed by value, just like any other primitive type.

```
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p2.x += p2.y;
    return p1;
}
```

- This could be used in the following way

```
strcut point p1 = { 100, 200 };
strcut point p2 = { 150, 100 };
struct point p3 = addpoint(p1, p2);
```

- Note that when we pass by value, we create a copy of the entire structure. When we return a struct, we are returning a complete copy of the structure.
- It is generally more efficient to pass/return a pointer than to copy the whole structure.

Operations on Structures

- The only legal operations on a structure are
  - copying it or assigning to it as a unit,
  - taking its address with &,
  - and accessing its members using the dot . operator.
- Copy and assignment include passing arguments to functions and returning values from functions as well.
- Structures may not be compared.
- A structure may be initialized by a list of constant member values.

Pointers to Structures

- Structure pointers are just like pointers to ordinary variables. For example:

```
struct point *pp;
```

says that pp is a pointer to a structure of type `struct point`. If pp points to a point structure, `*pp` is the structure, and `(*pp).x` and `(*pp).y` are the members. To use pp, we might write, for example,

```
struct point origin, *pp;
pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

- Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then *p->member-of-structure* refers to the particular member. So we could write instead

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Precedence of . and -> operators

- Both . and -> associate from left to right, so if we have

```
struct rect r, *rp = &r;
//then these four expressions are equivalent:
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

- The structure operators . and ->, together with () for function calls and [] for subscripts, are at the top of the precedence hierarchy and thus bind very tightly.
- For example, given the declaration:

```
struct {
    int len;
    char *str;
} *p;
```

  What happens with: `++p->len`, `(++p)->len`, `(p++)->len`.
  What happens with: `*p->str`, `*p->str++`, `(*p->str)++`, `*p++->str`.
- See example lab/C-examples/structures/structs-ex0.c

More Structure Examples

- The following declaration creates a new type `struct record`.

```
struct record {
    char *name;
    long int studentId;
    char *major;
};
```

- Now we can use the new type to declare variables, arrays, pointers etc. For example:

```
struct record r;  /* static declaration */
struct record students1[1000]; /*static 1-d array*/
struct record students2[100][1000]; /*static 2-d
   array*/
struct record *student; /* pointer to a struct*/

n = 1000;
student = (struct record *) malloc(sizeof(struct
   record)*n);

for (i=0; i<n; i++) {
    student[i].studentId = i; /* give a fake id */
    student[i].major = NULL; /* no major yet */
    student[i].name = (char *) malloc(sizeof(char)
                      * MAX_NAME_LENGTH);
}
```

Self-referential Structures

- It is valid for a structure to contain a pointer to itself! For example, here is the declaration of a structure to represent a node in a singly linked list:

```
struct node {
    int item;
    struct node *next;
};
```

- Here is some code that uses the above node structure.

```
struct node *head = (struct node *) malloc(sizeof(struct node));
(*head).item = 10;
/* the following is equivalent to the above */
head->item = 10;
/* undefined as malloc doesn't initialize
   pointers inside the struct */
head->next = ????
```

Union (1)

- A union is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements.
- Consider the following union declaration:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

- Then if we have a variable named utype to keep track of the type of data stored, we could write the following code:

```
if (utype == INT)
    printf("%d\n", u.ival);
if (utype == FLOAT)
    printf("%f\n", u.fval);
if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("bad type %d in utype\n", utype);
```

Union (2)

- The intended purpose of union is to save memory by using the same memory region for storing different objects at different times.
- In effect, a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the "widest" member, and the alignment is appropriate for all of the types in the union.
- The same operations are permitted on unions as on structures: assignment to or copying as a unit, taking the address, and accessing a member.
- A union may only be initialized with a value of the type of its first member.

Another Union Example (1)

- We will use a structure with a union inside it to store one of three types of data values.

```
enum utype{
  INT,
  DOUBLE,
  STRING
};
struct data {
    union udata {
        int ival;
        double dval;
        char *sval;
    } data;
    enum utype storedType;
};
```

- Without the union, the structure would need to store three separate variables, one for each type.

Another Union Example (2)

- Now we can use a switch statement to access the right type of values.

```
void printData(struct data value)
{
    switch (value.storedType) {
        case INT: printf("ival = %d \n", value.data.ival
    ); break;
        case DOUBLE: printf("fval = %f \n", value.data.
    dval); break;
        case STRING : printf("sval = %s \n", value.data.
    sval); break;
    }
}
```

- The following shows a sample usage:

```
  myCloset.data.ival = 10;
  myCloset.storedType = INT;
  printData(myCloset);

  myCloset.data.dval = 3.14159;
  myCloset.storedType = DOUBLE;
  printData(myCloset);

  myCloset.data.sval = "Cool!";
  myCloset.storedType = STRING;
  printData(myCloset);
```

Recommended Assignments

- Read Sections 6.1 through 6.9. Skim though Sections 6.5 and 6.6 as they are more difficult.
- Techniques introduced:
  - Using *doxygen* tool for javadoc style comments.
  - Using `valgrind` tool to check for memory errors and leaks.