CSE 142
Computer Programming I

Structures

Or...
There are more things in Heaven and Earth, C, than are dreamt of in your grammar.

Chapter 11

Read 11.1-11.3, & 11.7
11.1: Structure types
11.2: Structures as parameters
11.3: Structures as return values
Optional examples; skim or read:
11.4: Complex numbers

Review: Data Structures

Functions give us a way to organize programs. Data structures are needed to organize data, especially:
1. large amounts of data
2. variable amounts of data
3. sets of data where the individual pieces are related to one another
Arrays helped with points 1 and 2, but not with point 3
Example: the data describing one house in a neighborhood:
x, y, color, # windows, etc.
Example: information about one student: name, ID, GPA, etc.

Problem: Account Records
The Engulf & Devour Credit Co. Inc., Ltd. needs to keep track of insurance policies it has issued. Information recorded for each policy
Account number (integer)
Policy holder’s age (integer) and sex (‘m’ or ‘f')
Monthly premium (double)
At E&G, customers are only known by their account #, so there is no need to store their names.

Structs: Heterogeneous Structures

Collection of values of possibly differing types.
Name the collection; name the components (fields).
Example: Insurance policy information for Alice (informally)
   “alice”
account 9501234
age 23
sex ‘f’
premium 42.17

C expressions:
   alice.account 9501234
   alice.age is 23
   alice.sex is ‘f’
   2*alice.premium is 84.34

© 2000 UW CSE
Defining structs

There are several ways to define a struct in a C program. For this course:
- Define a new type specifying the fields in the struct.
- Declare variables as needed using that new type.
- The type is defined only once at the beginning of the program.
- Variables with this new type can be declared as needed.

Defining struct types

```
typedef struct { /* record for one policy:*/
    int account;      /* account number */
    int age;          /* policy holder's age */
    char sex;         /* policy holder's sex */
    double premium;   /* monthly premium */
} account_record;
```

Defines a new data type called account_record.

Does not declare (create) any variables. No storage is allocated.

Style Points in struct types

In a type definition, use comments to describe the fields, not the contents of the fields for any particular variable.
- i.e., describe the layout of an account_record, not information about Alice's account.
- typedefs normally are placed at the top of the program file.

Declaring struct Variables

Follow the usual rules:
- write the type name followed by one or more variable identifiers.
- Only difference: this time the type is defined by the programmer, not built in.

```
/*typedef students_record goes at top of program*/
...
account_record alice;
account_record bob;
```

account_record is a type; alice and bob are variables.

Both variables have the same internal layout.

Field access

A fundamental operation on struct variables is field access:
- struct_name.field_name selects the given field (variable) from the struct.

```
alice.age = 23;
alice.premium = 12.20;
alice.premium = 2 * alice.premium;
```
**Field access**

A selected field is an ordinary variable - it can be used in all the usual ways.

```c
alice.age++;  
printf("Alice is %d years old\n", alice.age);  
scanf("%lf", &alice.premium);
```

**Terminology**

The terms “struct”, “record” and “structure” mean the same thing.

“fields” are often called “components” or “members”.

---

**Why use structs?**

Collect together values that are treated as a unit (for compactness, readability, maintainability).

```c
typedef struct {  
    int hours, minutes;  
    double seconds;  
} time;

typedef struct {  
    int dollars, cents;  
} money;
```

This is an example of “abstraction”.

---

**Structs as User-Defined Types**

C provides a limited set of built-in types: int, char, double (and variants of these not discussed in these lectures).

Pointers introduced some new types.

Arrays further enrich the possible types available.

But... the objects in the real world and in computer applications are often more complex than these types allow.

With structs, we’re moving toward a way for programmers to define their own types.

---

**Some Limitations**

Like arrays, there are some restrictions on how a struct can be used compared to a simple variable (int, double, etc.)

- Can’t compare (==, !=) two structs directly.
- Can’t read or write an entire struct with `scanf`/`printf`.

But you can do these things on individual fields.

---

**struct Assignment**

Unlike arrays, entire structs can be copied in a single operation. Don’t need to copy field-by-field.

- Can assign struct values with `=`.
- Can have functions with struct result types, and can use struct values in a `return` statement.
A struct assignment copies all of the fields. If dilbert is another account_record, then
dilbert = bob;
is equivalent to
dilbert.account = bob.account;
dilbert.age = bob.age;
dilbert.sex = bob.sex;
dilbert.premium = bob.premium;

struct Assignment

account
age
sex
premium

dilbert
46532
12
m
12.95

bob
46532
12
m
12.95

structs as Parameters

structs behave like all other non-array values when used as function parameters
Can be call-by-value (copied)
Can use as pointer parameters

Midpoint Example Revisited

/* Given 2 endpoints of a line, "return" coordinates of midpoint */
void midpoint(
double x1, double y1,
double x2, double y2,
double *midxp, double *midyp )
{
  *midxp = (x1 + x2) / 2.0;
  *midyp = (y1 + y2) / 2.0;
}
double ax, ay, bx, by, mx, my;
midpoint(ax, ay, bx, by, &mx, &my);

Midpoint with points

/* return point whose coordinates are the center of the line segment with endpoints pt1 and pt2. */
point midpoint (point pt1, point pt2) {
  point mid;
  mid.x = ( pt1.x + pt2.x ) / 2.0;
  mid.y = ( pt1.y + pt2.y ) / 2.0;
  return mid;
}
...
point a = (0.0, 0.0), b = (5.0, 10.0);
... /* struct declaration and initialization */
m = midpoint (a, b); /* struct assignment */

struct initializers

A struct can be given an initial value when it is declared. List initial values for the fields in the same order they appear in the struct typedef.

account_record
ratbert = { 970142, 6, '?', 99.95 };
Execution

```c
void set_midpoint (point pt1, point pt2) {
    point mid;
    mid.x = (pt1.x + pt2.x) / 2.0;
    mid.y = (pt1.y + pt2.y) / 2.0;
    return mid;
}
```

```c
point a = {0.0, 0.0}, b = {5.0, 10.0}, m;
set_midpoint (a, b);
```

Field Access via Pointers

Function `set_midpoint` needs to access the x and y fields of its third parameter. How?

```c
void set_midpoint (point pt1, point pt2, point *mid)
...
Field access requires two steps:
  1) Dereference the pointer with *
  2) Select the desired field with .
Technicality: field selection has higher precedence
  than pointer dereference, so parentheses are needed: (*mid).x
```

Midpoint with Pointers

Instead of creating a temporary variable and returning a copy of it, we could write the function so it stores the midpoint coordinates directly in the destination variable. How? Use a pointer parameter:

```c
void set_midpoint (point pt1, point pt2, point *mid)
...
point a = {0.0, 0.0}, b = {5.0, 10.0}, m;
set_midpoint (a, b, &m);
```

Structs behave like all non-array types when used as parameters.

Midpoint with Pointers

/* Store in *mid the coordinates of the midpoint */
/* of the line segment with endpoints pt1 and pt2 */
void set_midpoint (point pt1, point pt2, point *mid) {
    (*mid).x = (pt1.x + pt2.x) / 2.0;
    (*mid).y = (pt1.y + pt2.y) / 2.0;
}

point a = {0.0, 0.0}, b = {5.0, 10.0}, m;
set_midpoint (a, b, &m);

Field Access via Pointers

Pointers Shorthand: `->`

“Follow the pointer and select a field” is a very common operation. C provides a shorthand operator to make this more convenient.

```c
structp -> component
means exactly the same thing as
("structp).component
```

-> is (sometimes) called the “indirect component selection operator”
**Pointer Shorthand: ->**

Function `set_midpoint` would normally be written like this:

```c
/* Store in *mid the coordinates of the midpoint */
/* of the line segment with endpoints pt1 and pt2 */
void set_midpoint (point pt1,
                  point pt2, point *mid)
{
    mid->x = (pt1.x + pt2.x) / 2.0;
    mid->y = (pt1.y + pt2.y) / 2.0;
}
```

---

**Summary**

- **Structs** collect variables (“fields”) possibly of differing types each field has a name . operator used to access
- **Struct** fields follow the rules for their types
- Whole structs can be assigned
- An important tool for organizing data

---

**QOTD: Frankenstein Types**

You can now create a huge variety of types – pointers to things, arrays of things, structs containing all sorts of other things.

Create the wierdest, nastiest useful type you can imagine and show how you would access each part of that type.

Now, make a function with a pointer to your type as a parameter and show what you’d do!

Need something more concrete? OK, your type should at least be a struct with an array of something inside it!