CSC 180 lab 7B: Structs

Thurs Oct 25 or Mon Oct 29, 2001

Note: For week 7, there are two half-labs; this week's lab assignment is a combination of two separate, short topics. Please also see lab 7A, "Command-line arguments".

Last week's lab examined arrays, an "aggregate" data type. This week's lab examines another aggregate data type, called "records" in most programming languages, but called "struct" (for "structure") in C.

A struct is not uniform like an array. It contains a specified list of various data types, and each item gets a *name*, not an index.

1. A struct can be used to represent a time interval of the format 12:34:56, as follows:

```
struct timeinfo {
    int hours;
    int mins;
    int secs;
};
```

The "timeinfo" identifier there is called the struct "tag"; this creates a type called "struct timeinfo", so that we can later declare variables such as "struct timeinfo x;". (I didn't call it "time" because there is already a C library function called "time".)

If *x* is of type "struct timeinfo", then its members can be referred to as x.hours, x.mins, and x.secs.

2. If we want to pass structs to functions, or return them from functions, we can do so in the obvious way. But we frequently pass *pointers* to structs instead, so that the called function can modify the struct members. If we declare a function

```
void normalize(struct timeinfo *p)
```

then we could call this function by saying

```
normalize(&x);
```

and it could adjust the hours, mins, and secs members. Within the function normalize(), it would have to say things like (*p).hours. The precedence of '*' and '.' is such that those parentheses are necessary. However, there is an alternate syntax, which is "−>" (a minus sign and a greater-than sign, but we think of it as an arrow). So "something−>otherthing" is short for "(*something).otherthing". Thus inside normalize, we could refer to p−>hours, p−>mins, and p−>secs.

3. Write the function "normalize". It has the following specification, referring to the "old" values of hours, mins, and secs before the call, and the "new" values after it returns. All of the following should be made true:

- (oldhours * 60 + oldmins) * 60 + oldsecs = (newhours * 60 + newmins) * 60 + newsecs
- $0 \le$ newsecs < 60
- $0 \le$ newmins < 60

(Thus negative-forty minutes is represented by −1 hours and 20 minutes! Odd, but this makes it not a special case.)

4. Write a little test program to try your normalize function. The test program should read in the three numbers (you can pass things like &x.hours to scanf()), call normalize(), then output the results. For example, an input of 0 hours, 0 minutes, and 3601 seconds should output 1 hours, 0 minutes, and 1 seconds.

5. Given the normalize function, a function to add two times together is easy. If you have time, after doing lab 7A, write a function which takes two struct parameters (not pointers to struct) and returns a struct (again, not a pointer to a struct) which is the sum of the two times. It can declare a result struct; set its hours value to the sum of the two argument hours, its minutes value to the sum of the two argument minutes, and its seconds value to the sum of the two argument seconds; then normalize this value; then return it.