FCPA 2022  
  
Structured Types

Student Workbook 08

January 19, 2022

Author

Paul Kimball  
Interface Associates

1. Structures

Drowning in a sea of bytes

* Bytes, chars, pointers, arrays, NULL-terminated strings...
  + They're all just bytes in memory!
* We need to impose some structure!

struct

union

enum

struct

* A *struct* represents a record
  + A set of related members that describe a larger concept
  + Each member has a type and a name
  + The overall structure has a fixed size that is large enough to hold all its members
* Example - a value reported by a sensor
  + In this case, an accelerometer

**struct accelerometer\_event**

**{**

    short sensor\_id;

    float accel\_x;

    float accel\_y;

    float accel\_z;

**};**

* A struct type is identified by the keyword *struct* + its *tag name*
  + You can declare a variable using the struct type:

**struct accelerometer\_event** event;

Typedef

* A *typedef* statement defines a new name for an existing type
  + We saw this used with enumerated types
* A typedef makes the struct seem more natural as a data type

**typedef** struct accelerometer\_event **accelerometer\_event**;

* + Best practice in embedded C is for the tag name and the type name to be the same
  + Declare a variable simply, using your new type

**accelerometer\_event** event;

* You can combine the typedef statement with the struct type declaration if you want:

**typedef**

struct accelerometer\_event

{

    short sensor\_id;

    float accel\_x;

    float accel\_y;

    float accel\_z;

} **accelerometer\_event**;

**accelerometer\_event** event;

Accessing members of a struct

* In a struct variable, members are accessed using the "dot" operator

accelerometer\_event event;

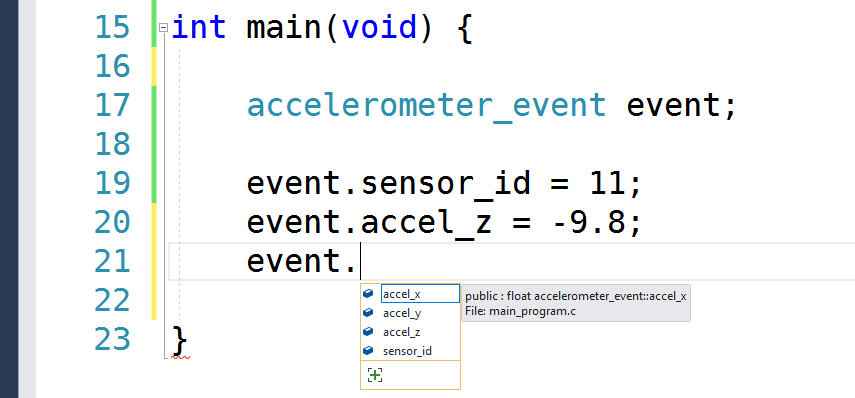
**event.sensor\_id** = 11;

event**.**accel\_x = 0.0;

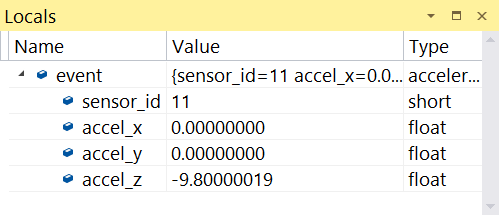
event**.**accel\_y = 0.0;

event**.**accel\_z = -9.8;

* Your editor understands this, and will make suggestions:



* So does your debugger, when reporting values



Initializing a struct variable

* You can initialize a struct variable with an *initializer*:

accelerometer\_event event = **{11, 0., 0., -9.8}**;

* + Values for members must be in the order declared in the struct
* As of C99, you can use a *designated initializer*, which is easier to read:

accelerometer\_event slow\_decel = **{**

**.sensor\_id = 11,**

**.accel\_z = -9.8,**

**.accel\_y = 0.0,**

**.accel\_x = -1.**0 };

* + Values need not be in order
* You can also initialize from an existing struct

accelerometer\_event impact\_event = **event**;

impact\_event.accel\_y = -22.4;

Size of structs

* A struct is always large enough to contain all its members
* You can determine the size with sizeof:

sizeof(accelerometer\_event)

// or

sizeof(slow\_decel)

* The compiler is allowed to add padding between members to meet requirements for alignment, word size etc.

Assigning structs

* Assignment between structs of the same type copies all the memory in the structure

event = impact\_event;

* As of C99, you can assign from a *compound literal*:
  + Looks like a cast + an initializer list

event = (accelerometer\_event) {11, 0., 0., -9.8};

Passing a structure by value

* Structures can be passed by value
* As a parameter to a function

void print\_event(**accelerometer\_event event**) {

printf("Sensor %d: accel [%f, %f, %f]",

event.sensor\_id, event.accel\_x,

event.accel\_y, event.accel\_z);

}

* As a return value

**accelerometer\_event** get\_next\_event() {

accelerometer\_event ev;

ev.sensor\_id = read\_sensor\_id();

ev.accel\_x = read\_x\_val();

ev.accel\_y = read\_y\_val();

ev.accel\_z = read\_z\_val();

**return ev;**

}

Passing a struct by reference

* The address operator can be used to pass a struct by reference

void react(**accelerometer\_event \*event\_ptr**) {

if (fabsf( **event\_ptr->accel\_y** ) > 2.0) {

puts("Wow! this ride is exciting!");

}

if (( **(\*event\_ptr).accel\_z** ) > 3.0) {

puts("Wow! this ride is terrifying!");

}

}

* When you have a pointer-to-structure, you can use the *arrow operator* -> to access members:

event\_ptr**->**accel\_z

* + Is equivalent to:

(\*event\_ptr).accel\_z

Arrays inside structs

* An array can be a member of a structure
* The array must be a fixed size

struct bom\_part\_count {

**char part\_id[48];**

int quantity;

}

* If the array is being treated as a NULL-terminated string, remember to use the functions in <string.h> to copy, compare, and concatenate

Pointer types in structs

* A pointer can be a member of a struct
* The pointer value (that is, the address in the pointer member) is copied when the surrounding structure is copied

**BUT**

* The value that it points to is NOT copied automatically
* Sometimes this is not a problem
  + It is a pointer to a constant value that never changes
  + It is a pointer to an object that is safe to share
  + The pointer value is always updated correctly when the surrounding struct is manipulated (e.g., a linked list)
* Sometimes this introduces BIG issues
  + Complex copy semantics
  + Read/write synchronization
  + Lifetime of values
  + We'll discuss these later

1. Unions

Union

* Syntax is just like a struct, but all union members are mapped to the same chunk of memory
  + A union is always large enough to contain its largest member

**union sensor\_event**

{

    struct accelerometer\_event sv\_accelerometer;

    struct magnetometer\_event sv\_magnetometer;

    struct proximity\_event sv\_proximity;

};

**typedef union sensor\_event sensor\_event;**

## Union members are accessed using the dot . or arrow operators as they are in structs

sensor\_event ev;

**ev.sv\_accelerometer** = get\_accel\_reading(11);

## A union provides a way to reuse the *same* memory for different data types, *at different times*

### Can save memory when dealing with types that are of similar size but contain different values.

## The member retrieved must be the member most recently stored

### Writing to a member invalidates all other members

### However, the union does not track changes, nor remember what was placed in it most recently

### It is up to the programmer to enforce correct usage. Hmmm....

### And, there is an exception to this rule...

Union as a Family of types

* if a union contains several structures that share a common initial sequence, it is permitted to inspect the common initial part of any of them
* A union can be used to represent a family of related types that share common features
  + An enum is a natural choice to keep track of what is currently stored in the union

enum sensor\_type

{

kUnknown,

    kAccelerometer,

    kMagnetometer,

    kProximity

};

struct any\_event {

**enum sensor\_type type;**

**int timestamp;**

**short sensor\_id;**

};

struct accelerometer\_event {

**enum sensor\_type type;**

**int timestamp;**

**short sensor\_id;**

    float accel\_x;

    float accel\_y;

    float accel\_z;

};

// other types of events declared here...

**typedef**

**union sensor\_event**

**{**

**enum sensor\_type type;**

struct any\_event sv\_any;

    struct accelerometer\_event sv\_accelerometer;

    struct magnetometer\_event sv\_magnetometer;

    struct proximity\_event sv\_proximity;

**} sensor\_event;**

Union as a generic type

* This function does not need to know the details of any particular event type

void sensor\_loop(void)

{

**sensor\_event event;**

    for (;;)

    {

**event = getNextSensorValue();**

        switch (**event.type**)

        {

        case kAccelerometer:

            handleAccelerometer(event);

            break;

        case kMagnetometer:

            handleMagnetometer(event);

            break;

        case kProximity:

            handleProximity(event);

            break;

        default:

            handleAnyEvent(event);

        }

    }

}

* This function can easily access the enclosed object

void handleAccelerometer(sensor\_event event)

{

    if (event.type != kAccelerometer)

        return;

    puts("Accelerometer event from %d",

**event.sv\_accelerometer.sensor\_id** );

);

Type Punning

* One traditional use of a union is "type punning"

**union \_HollerithChars**

**{**

**char h[8];**

**unsigned long int n;**

**};**

...

union \_HollerithChars buf;

**buf.n = 0x0000216f6c6c6548;** //NUL NUL ! o l l e H

printf("You say '%lx' \n", buf.n);

printf("I say '%s' \n", buf.h);

printf("Starts with '%c' and ends with '%c'\n",

buf.h[0], buf.h[6]);

* Not recommended; and NOT portable (but people do it anyway);

Arrays of Structs/Unions

* An array of structs or unions is easy to declare
  + This allocates memory for the objects
  + But how are they initialized?

sensor\_event event\_list[20];

* Array of pointers to struct is also easy to declare, but must be initialized explicitly to reference allocated objects

sensor\_event \*event\_ptr\_list[20];

// Assumes that all events in event\_list are valid

for (int i=0; i<20; i++){

event\_ptr\_list[i] = &event\_list[i];