Structs and Unions

C Structures, Unions, Example Code
Review

- Introduction to C
- Functions and Macros
- Separate Compilation
- Arrays
- Strings
- Pointers
Structs vs Objects

• C is not an OOP language
  ▫ No way to combine data and code into a single entity
• Struct – C method for combining related data
  ▫ All data in a struct can be accessed by any code

Coming from an object-oriented programming background, think of classes as an extension of struct. Classes have data members but allow you to restrict access to them while providing a mechanism to organize and bundle a set of related functions. You can think of a struct as an OOP class in which all data members are public, and which has no methods, not even a constructor.
Structs

• A struct represents a block of memory where a set of variables are stored
  ▫ Each member of struct has offset from beginning of struct block determining where data is located

• General form of structure definition:
  ```
  struct example{
    type ex1;
    type ex2;
  }
  ```

• Note the semicolon at the end of the definition
Struct Example

• A point in the Euclidean coordinate plane
  ▫ struct point{
  ▪ int x;  //x-coordinate
  ▪ int y;  //y-coordinate
  ▫ }

• To create point data types:
  ▫ struct point p1,p2;

• To access point members:
  ▫ p2.x, p2.y
Passing Structs to Functions

• Like other variable types, struct variables (e.g. p1, p2) may be passed to function as parameter and returned as parameters
  ▫ The ability to return a struct variable provides option to bundle multiple return values
• Members of a struct are variables and may be used like any other variable
  ▫ i.e. p1.x can be used like any other integer
// struct point is a function parameter
void printPoint( struct point aPoint) {
    printf("( %2d, %2d )", aPoint.x, aPoint.y);
}

// struct point is the return type
struct point inputPoint( ) {
    struct point p;
    printf("please input the x-and y-coordinates: ");
    scanf("%d %d", &p.x, &p.y);
    return p;
}

int main( ) {
    struct point endpoint; // endpoint is a struct point variable
    endpoint = inputPoint( );
    printPoint( endpoint );
    return 0;
}
Initializing a Struct

- Struct variables may be initialized when it is declared by providing the initial values for each member
  - E.g. struct point p1 = {-5,7};
- Struct variables may be declared at the same time the struct is defined
  - Struct point{ int x, y;} startpoint, endpoint;
    - Defines structure point, and point variables startpoint and endpoint
Typedef and Structs

- It's common to use a typedef for the name of a struct to make code more concise
  - `typedef struct point{
    int x, y;
  } POINT_t;`

- This defines the structure `point`, and allows declaration of point variables using either struct `point`, or just `POINT_t`
  - E.g. `struct point endpoint; POINT_t startpoint;`
  - Same can be done with Enums
    - `typedef enum months{} MONTHS_e;`
Struct Assignment

- Contents of struct variable may be copied to another struct variable using assignment (=)
  - `POINT_t p1, p2;`
  - `p1.x=15;`
  - `p1.y = -12;`
  - `p2 = p1;  // same as p2.x = p1.x; p2.y = p1.y`

- Assignment represents copying a block of memory with multiple variables
Struct Within a Struct

• A data element in a struct may be another struct
  ▫ Similar to class composition in OOP

• E.g line composed of points
  ▫ typedef struct line { POINT_t start, end } LINE_t;

• Given declarations below, how do you access x and y coordinates of line
• LINE_t line, line1, line2;
  ▫ Line.start.x = 13
Arrays of Struct

- Since struct is a variable type, arrays of structs may be created like any other type
  - E.g. `LINE_t lines[5];`
- Code to loop through and print each lines start point

```c
for(int i = 0; i<5; i++){
    printf(\%d,\%d\n\", lines[i].start.x, lines[i].start.y);
```
/* assume same point and line struct definitions */

```c
int main() {
    struct line lines[5];  // same as LINE_t lines[5];
    int k;
    /* Code to initialize all data members to zero */
    for (k = 0; k < 5; k++) {
        lines[k].start.x = 0;
        lines[k].start.y = 0;
        lines[k].end.x = 0;
        lines[k].end.y = 0;
    }
    /* call the printPoint() function to print
     ** the end point of the 3rd line */
    printPoint( lines[2].end);
    return 0;
}
```
Arrays Within a Struct

• Structs may contain arrays as well as primitives
  typedef struct month{
    int nrDays;
    char name[3+1];
  }MONTH_t;
  MONTH_t january = {31,"JAN"};

• Note: january.name[2] is ‘N’
Example Struct with Arrays

```c
struct month allMonths[12] =
{31, "JAN"}, {28, "FEB"}, {31, "MAR"},
{30, "APR"}, {31, "MAY"}, {30, "JUN"},
{31, "JUL"}, {31, "AUG"}, {30, "SEP"},
{31, "OCT"}, {30, "NOV"}, {31, "DEC"}
}; //Same as MONTH_t allMonths[12]=...;
// write the code to print the data for September
printf("%s has %d days\n", allMonths[8].name, allMonths[8].nrDays);
// what is the value of allMonths[3].name[1]
printf("%c\n", allMonths[3].name[1]);
P
printf("%s\n", allMonths[3].name);
APR
```
Bit Fields

• When saving space in memory or a communications message is important, we need to pack lots of information into a small space
• Struct syntax can be used to define “variables” which are as small as 1 bit in size
  ▫ Known as “bit fields”

Struct weather{
  unsigned int temperature : 5;
  unsigned int windSpeed : 6;
  unsigned int isRaining : 1;
  unsigned int isSunny : 1;
  unsigned int isSnowing : 1;
};
Using Bit Fields

- Bit fields are referenced like any other struct member

```c
struct weather todaysWeather;
todaysWeather.isSnowing = 0;
todaysWeather.windSpeed = 23;
// etc
If(todaysWeather.isRaining)
    printf("%s\n", "Take your umbrella");
```
More on Bit Fields

• Almost everything about bit fields is implementation specific
  ▫ Machine and compiler specific
• Bit fields may only be defined as (unsigned) ints
• Bit fields do not have addresses
  ▫ & operator may not be applied to them
Unions

- A union is a variable type that may hold different types of members of different sizes, but only one type at a time
  - All members of the union share the SAME memory
  - Compiler assigns enough memory for the largest of the member types
  - Syntax of a union and using its members is the same as for a struct
Union Definition

• General form of a union definition is
  ```
  Union ex{
    type member1;
    type member2;
  };
  ```

• Note that the format is the same as for a struct
• Only member1 or member2 will be in that memory location
Application of Unions

```c
struct square { int length; };
struct circle { int radius; };
struct rectangle { int width; int height; };
enum shapeType {SQUARE, CIRCLE, RECTANGLE };
union shapes {
    struct square aSquare;
    struct circle aCircle;
    struct rectangle aRectangle;
};
struct shape {
    enum shapeType type;
    union shapes theShape;
};
```
Application of Unions

double area( struct shape s) {
    switch( s.type ) {
        case SQUARE:
            return s.theShape.aSquare.length * s.theShape.aSquare.length;
        case CIRCLE:
            return 3.14 * s.theShape.aCircle.radius * s.theShape.aCircle.radius;
        case RECTANGLE:
            return s.theShape.aRectangle.height * s.theShape.aRectangle.width;
    }
}
Union vs. Struct

• Similarities
  ▫ Definition syntax nearly identical
  ▫ Member access syntax identical

• Differences
  ▫ Members of a struct each have their own address in memory
  ▫ Size of a struct is at least as big as the sum of the sizes of the members
  ▫ Members of a union SHARE the same memory
  ▫ The size of the union is the size of the largest member
Struct Storage in memory

- Struct elements are stored in the order they are declared in.
- Total size reserved for a struct variable is not necessarily the sum of the size of the elements.
  - Some systems require some variables to be aligned at certain memory addresses (usually small power of 2).
    - Requires some padding between members in memory = wasted space.
  - If members are reordered, it may reduce total number of padding bytes required.
    - Usually rule of thumb is to place larger members at the beginning of definition, and small types (char) last.
  - Special compiler options may allow packing, reducing, or eliminating padding but may come at a cost in speed as data must be manipulated.
  - In 8-Bit AVR with single-byte memory access there will be no padding.
How to Print the Bytes of a Structure to See Padding

Code:
```c
#include <stdio.h>
#include <stdlib.h>

typedef struct dummy_tag1 {
    signed char c1;
    int i1;
    signed char c2;
} big_t;

typedef struct dummy_tag2 {
    int i1;
    signed char c1;
    signed char c1;
} small_t;

int main(){
    big_t big = {1,-1,1};
    small_t small = {-1,1,1};

    unsigned char * ptrByte; //pointer for accessing individual bytes
    ptrByte = (unsigned char *)&big;
    printf("BIG:(%d bytes): \n", sizeof(big_t));
    for (int i=0; i<sizeof(big_t); i++){
        printf("%02x\n", *ptrByte);
        ptrByte++;
    }

    ptrByte = (unsigned char *)&small;
    printf("SMALL(%d bytes): \n", sizeof(small_t));
    for (int i=0; i<sizeof(small_t); i++){
        printf("%02x\n", *ptrByte);
        ptrByte++;
    }

    return 0;
}
```

Compile:
```
$ gcc -Wall -std=c99 ./test.c
```

First Call
```
$ ./a.out
BIG:(12 bytes):
01
00
00
00
ff
ff
ff
ff
ff
ff
01
00
00
00
00
SMALL(8 bytes):
ff
ff
ff
ff
ff
ff
01
01
5e
57
```

Second Call
```
$ ./a.out
BIG:(12 bytes):
01
00
00
00
ff
ff
ff
ff
ff
ff
01
00
00
00
00
SMALL(8 bytes):
ff
ff
ff
ff
ff
ff
01
01
5e
c9
```

Wasted Space for Padding is highlighted red (platform dependent). The last two bytes of small are garbage values, illustrated by the juxtaposition of two successive runs.
Examining Bytes of a Union

**Code:**

```c
#include <stdio.h>
#include <stdlib.h>

typedef union dummy_tag1 {
    signed char c1;
    int i1;
} T;

int main() {
    T myUnion;
    unsigned char *ptrByte; //variable for printing bytes

    printf("sizeof(unsigned char): %d byte\n", sizeof(unsigned char));
    printf("sizeof(int): %d bytes\n", sizeof(int));
    printf("sizeof(T): %d bytes\n", sizeof(T));

    myUnion.i1 = 0; //clear all the b
    printf("Cleared Bytes of Union Variable:\n");
    ptrByte = (unsigned char *)&myUnion;
    for (int i=0; i<sizeof(T); i++){
        printf("%02x\n", *ptrByte);
        ptrByte++;
    }

    myUnion.c1 = -1;
    printf("After setting member c1 to -1:\n");
    ptrByte = (unsigned char *)&myUnion;
    for (int i=0; i<sizeof(T); i++){
        printf("%02x\n", *ptrByte);
        ptrByte++;
    }

    myUnion.i1 = -1;
    printf("After setting member i1 to -1:\n");
    ptrByte = (unsigned char *)&myUnion;
    for (int i=0; i<sizeof(T); i++){
        printf("%02x\n", *ptrByte);
        ptrByte++;
    }
    return 0;
}
```

**Compile:**

```
$ gcc -Wall -std=c99 ./test.c
```

**Run**

```
$ ./a.out
sizeof(unsigned char): 1 byte
sizeof(int): 4 bytes
sizeof(T): 4 bytes
Cleared bytes of union variable:
00
00
00
00
After setting member c1 to -1:
ff
00
00
00
After setting member i1 to -1:
ff
ff
ff
ff
```

Slide borrowed from Dr. Robucci’s 311 course