Structs

Chapter 16

Structs vs Arrays

- Arrays are a compound data type where all components are of the same type
- The components of an array are accessed by the array name and an integer index.
- structs are a compound data type that may have components of different types.
- The components of a struct are accessed by the struct name, a dot

Defining a Structure

- Often need to keep track of several pieces of information about a given thing.
- Example: Box
- We know its
  - length
  - width
  - height
  - weight
  - contents

Defining a Structure

- We could define a separate variable to hold each piece of information:

```
int length;
int width;
int height;
double weight;
char contents[32];
```

- If we wanted to pass the information to a function, it would have to have five parameters.

Function to do something regarding a box

```
void box_func (int length,
               int width,
               int height,
               double weight,
               char contents[])
{
  /* Do something with a box. */
  ...
}
```

The C “struct”

- The C “**struct**” is a way to combine all of this information in a single package.

```
typedef struct box
{
        int length;
        int width;
        int height;
        double weight;
        char contents[32];
} box_t;
```

Optional “tag name”

OK to omit

Other naming convention:
first letter lowercase for tag

```
    struct box
```

first letter uppercase for new type nam

```
    } Box;
```

New user defined type

Declaring struct variables

- We can declare variables of the newly defined struct type in either of two ways:

```
struct box box2;
```

Older form.

Rarely used today.

```
box_t box1;
```

Newer form using the typedef.

This is just a programming style convention.

Not significant to the compiler and not required.

Members of a struct

```
typedef struct BOX
{
        int length;
        int width;
        int height;
        double weight;
        char contents[32];
} box_t;
```

Members of the struct

Members are declared within the struct just as local variables are declared within a function.

No memory is allocated by the typedef

Note style. (Book is different.)

Members of a struct

- Members of a struct can be any previously defined type
  - Including arrays
  - Including other structs

Declaring a struct variable

- In order to have the compiler allocate memory for a struct, we have to declare a variable of that type.

```
box_t box1;
```

- We can initialize a struct with the declaration:

```
box_t box1 = {24, 12, 12, 5.3, "Fine German Wine"};
```

How to access members of a struct

- To access a member, use the name of the struct variable “dot” name of the member

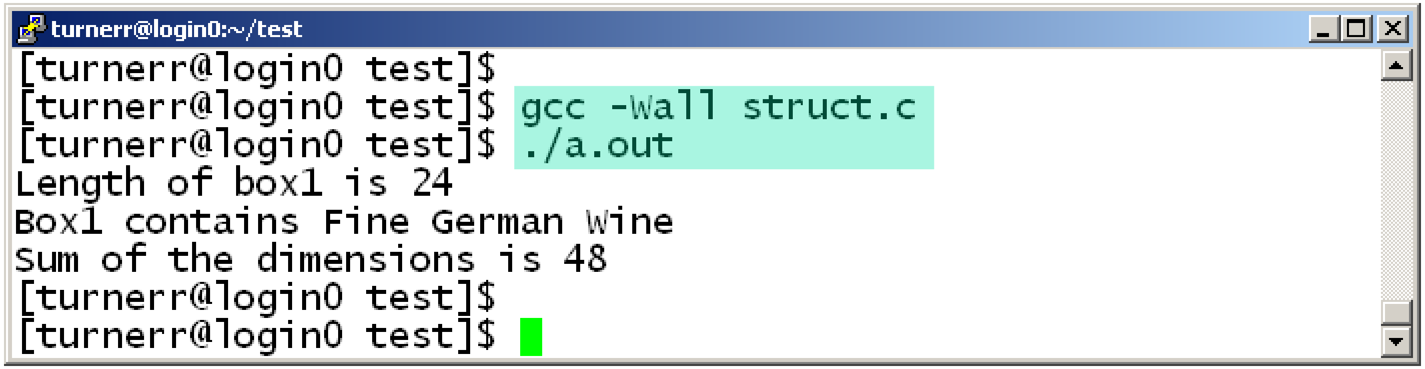
```
box1.length

box1.contents
```

- These expressions work like normal variable names.
- Note: box1 is the struct variable name, *not the type name*.

Accessing struct members

```
#include <stdio.h>

typedef struct BOX
{
    int length;
    int width;
    int height;
    double weight;
    char contents[32];
} box_t;

int main (void)
{
    int dimension_total;

    box_t box1 = {24, 12, 12, 5.3, "Fine German Wine"};

    printf ("Length of box1 is %d\n", box1.length );

    printf ("Box1 contains %s\n", box1.contents );

    dimension_total = box1.length + box1.width + box1.height;

    printf ("Sum of the dimensions is %d\n", dimension_total);

    return 0;
}
```

Accessing struct members



```
[turnerr@login0 test]$
[turnerr@login0 test]$ gcc -Wall struct.c
[turnerr@login0 test]$ ./a.out
Length of box1 is 24
Box1 contains Fine German Wine
Sum of the dimensions is 48
[turnerr@login0 test]$
[turnerr@login0 test]$
```

Using Structures

- We can do *almost* anything with structs that we can do with the built-in types.
  - Assignment.
  - Pass to a function.
  - Return from a function.
  - Create arrays of structs.
  - Use a struct as element in another struct.

Using Structures

- The one major exception is comparison.

- We can’t say
  - **`if (box1 == box2)`**

```
box_t box1 = {24, 12, 12, 5.3, "Fine German Wine"};
box_t box2 = box1;

if (box1 == box2)
{
    printf ("box1 and box2 are equal\n");
}
```

This gets a compile error.

Comparing Structures

We have to check each member individually.

```
box_t box1 = {24, 12, 12, 5.3, "Fine German Wine"};
box_t box2 = box1;

if ((box1.length == box2.length) &&
    (box1.width  == box2.width ) &&
    (box1.height == box2.height) &&
    (box1.weight == box2.weight) &&
    (strcmp(box1.contents, box2.contents) == 0))
{
    printf ("box1 and box2 are equal\n");
}
```

Passing a struct to a function

box is effectively a local variable

A *copy* of the box_t struct used as an argument by the caller.

```
void display_box ( box_t box )
{
    printf ("Box length is %d\n", box.length );
    printf ("Box width is %d\n",  box.width);
    printf ("Box height is %d\n", box.height);
    printf ("Box contains %s\n",  box.contents);
}
```

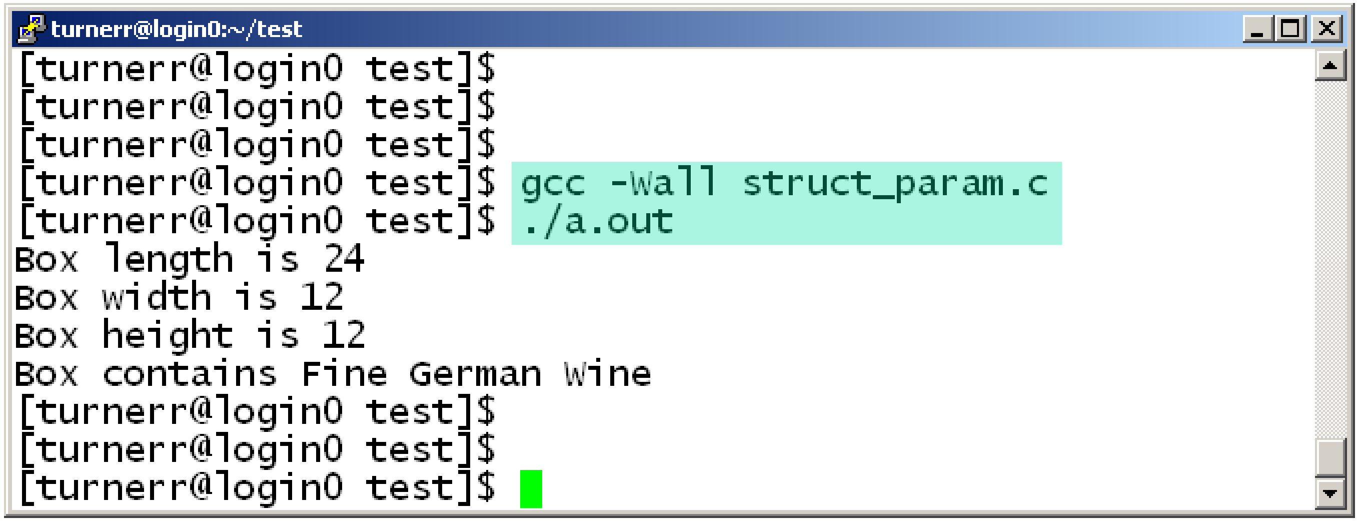
Used same as a local variable

```
int main ()
{
    box_t box1 = {24, 12, 12, 5.3, "Fine German Wine"};

    display_box(box1);

    return 0;
}
```

Just like passing an int to a function

This is an example of “Call by Value”

Passing a struct to a function



```
[turnerr@login0 test]$
[turnerr@login0 test]$
[turnerr@login0 test]$
[turnerr@login0 test]$ gcc -Wall struct_param.c
[turnerr@login0 test]$ ./a.out
Box length is 24
Box width is 12
Box height is 12
Box contains Fine German Wine
[turnerr@login0 test]$
[turnerr@login0 test]$
[turnerr@login0 test]$
```

Returning a struct from a function

Function returns a box_t struct

```
box_t new_box (int size_param)
{
    box_t box;

    box.length = size_param*3;
    box.width = size_param*2;
    box.height = size_param;
    box.weight = 3.4*size_param;
    strcpy (box.contents, "Unknown");

    return box;
}
```

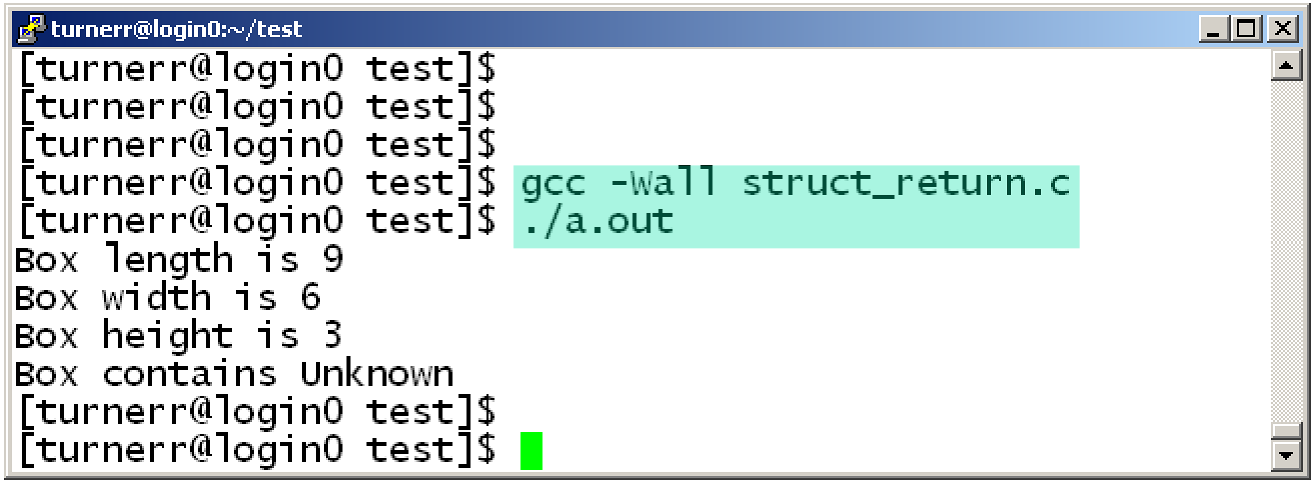
A ***copy*** of box is returned to the caller via the run time stack.

The local variable box disappears after the return.

Returning a struct from a function

```
int main (void)
{
    box_t sample_box;

    sample_box = new_box(3);

    display_box (sample_box);
    return 0;
}
```

Returning a struct from a function



```
[turnerr@login0 test]$
[turnerr@login0 test]$
[turnerr@login0 test]$
[turnerr@login0 test]$ gcc -Wall struct_return.c
[turnerr@login0 test]$ ./a.out
Box length is 9
Box width is 6
Box height is 3
Box contains Unknown
[turnerr@login0 test]$
[turnerr@login0 test]$
```

Arrays of Structs

- We can create arrays of struct types just like we do for primitive types.

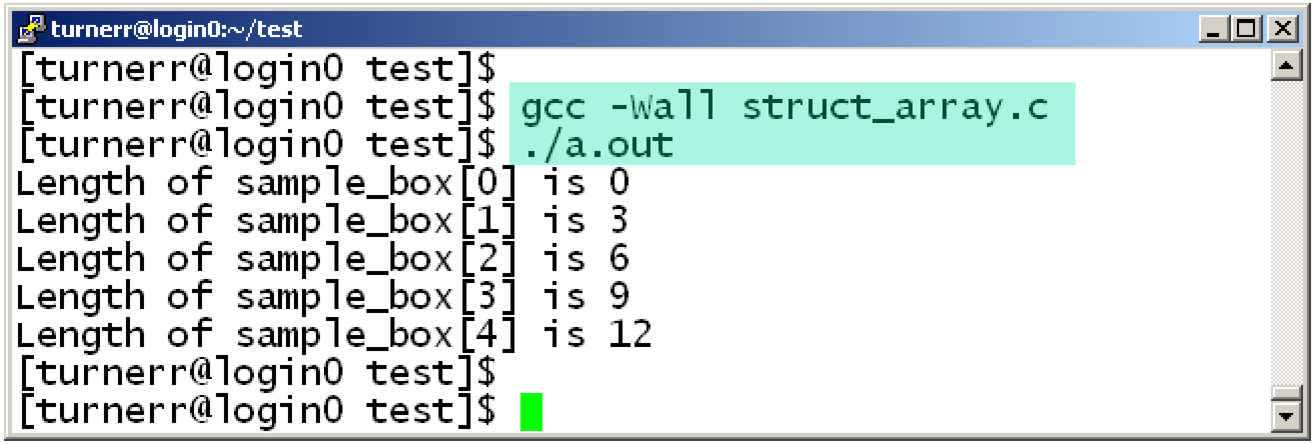
```
box_t sample_box[5];
```

- An array of five structs of type box_t

Arrays of Structs

```
int main ()
{
    box_t sample_box[5];
    int i;

    for (i = 0; i < 5; i++)
    {
        sample_box[i] = new_box(i);
    }

    for (i = 0; i < 5; i++)
    {
        printf ("Length of sample_box[%d] is %d\n",
            i, sample_box[i].length);
    }
    return 0;
}
```

Arrays of Structs



```
[turnerr@login0 test]$
[turnerr@login0 test]$ gcc -Wall struct_array.c
[turnerr@login0 test]$ ./a.out
Length of sample_box[0] is 0
Length of sample_box[1] is 3
Length of sample_box[2] is 6
Length of sample_box[3] is 9
Length of sample_box[4] is 12
[turnerr@login0 test]$
[turnerr@login0 test]$
```

Structs within Structs

- A struct can have another struct as a member

```
typedef struct
{
    int x;
    int y;
} point_t;

typedef struct
{
    point_t upper_left;
    point_t lower_right;
} rectangle_t;
```

Accessing Members of Structs within Structs

```
/* This function determines whether a point is inside
   a rectangle.  It counts being on the border as
   inside. */

int is_inside (point_t pPoint, rectangle_t pRect)
{
    return ((pPoint.x >= pRect.upper_left.x ) &&
            (pPoint.x <= pRect.lower_right.x) &&
            (pPoint.y >= pRect.upper_left.y ) &&
            (pPoint.y <= pRect.lower_right.y)    );
}
```

Returning a struct to the caller

- Get coordinates of a point from the user and return the point to the caller.

```
point_t get_point(char prompt[])
{
    point_t pt;
    printf (prompt);
    printf ("X: ");
    scanf ("%d", &pt.x);
    printf ("Y: ");
    scanf ("%d", &pt.y);
    return pt;
}
```

Returning a struct to the caller

```
rectangle_t get_rect(char prompt[])
{
    rectangle_t rect;
    printf (prompt);
    rect.upper_left = get_point("Upper left corner: \n");
    rect.lower_right = get_point("Lower right corner: \n");

    return rect;
}
```

More struct Examples

- We now define a struct to represent a date. Since a date is given by the month, the day and the year, we have:

```
typedef struct
{
        int month;
        int day;
        int year;
} date;
```

More struct Examples

- A fraction is given by its numerator and denominator:

```
typedef struct
{
        int numer;
        int denom;
} Fraction;
```

- Given the above typedef, we can now write code to perform I/O and arithmetic on fractions.

Summary

- The “struct” feature in C permits us to define data structures.
  - Any previously defined type can be a member
    - Including other structs
  - Acts as user defined type
- Use the “dot” notation to access members of a struct using the name of a struct variable.

Summary

- The assignment operator works for structs:
  - **`box2 = box1;`**

- But the comparison operator does not.
  - Can’t say **`if (box2 == box1)`** …

Summary

- Functions can have structs as parameters and can return a struct to the caller.

- Structs are passed by value.
  - Like single variables, not like arrays.
  - Function gets a *copy* of a struct passed as argument.
  - Call gets a *copy* of a struct returned as a function value.

Assignment

- Read Chapter 16
  - through section 16.3

- End of Presentation