"C programmers never die. They are just cast into void."

- Alan Perlis

CSE102
Computer Programming with C

2016-2018 Spring Semester

Structures

Largely adapted from J.R. Hanly, E.B. Koffman, F.E. Sevilgen, and others…

Structures

- How to define a structure?
- How to declare a variable?
- How to manipulate individual components?
- How to manipulate whole structures?
  - Assignment

Structures

- Defines a new type
  - Represents structured collection of data
    - Different type is possible
- EX: Planet type
  - Name
  - Diameter
  - Number of moons
  - Number of years to complete one solar orbit
  - Number of hours to complete one rotation.

Structure definition

```
typedef struct {
        char    name[20];
        double  diameter;
        int     moons;
        double  orbit_time,
                rotation_time;
} planet_t;

planet_t   my_planet;
```

Structure Definition

- A name chosen for a component of one structure may be the same as the name of a component of another structure or the same as the name of a variable
- The **typedef** statement itself allocates no memory
- A variable declaration is required to allocate storage space for a structured data object

```
planet_t    current_planet,
            previous_planet,
            blank_planet = {"", 0.0, 0, 0.0, 0.0};
```


Structure Definition (Cont'd)

Variable `blank_planet`, a structure of type `planet_t`

.name	\0 ? ? ? ? ? ? ? ? ?
.diameter	0.0
.moons	0
.orbit_time	0.0
.rotation_time	0.0

Structure Definition (Cont'd)

- Hierarchical structure
  - a structure containing components that are structures
- Example

```
typedef struct {
   double diameter;
   planet_t planets[9];
   char galaxy[STRSIZ];
} solar_sys_t;
```


Assigning Values

- Direct component selection operator: a dot (.) placed between a structure type variable and a component name to create a reference to the component

```
strcpy(current_planet.name, "Jupiter");
current_planet.diameter = 142800;
current_planet.moons = 16;
current_planet.orbit_time = 11.9;
current_planet.rotation_time = 9.925;
```

Variable `current_planet`, a structure of type `planet_t`

.name	J u p i t e r \0 ? ?
.diameter	142800.0
.moons	16
.orbit_time	11.9
.rotation_time	9.925

Manipulating Structures

```
printf("%s's equatorial diameter is %.1f km.\n",
            current_planet.name, current_planet.diameter);
```

→ Jupiter's equatorial diameter is 142800.0 km.

- With no component selection operator refers to the entire structure

```
previous_planet = current_planet;
```

- Direct component operator (.) has the highest precedence.

Structures as Arguments

- When a structured variable is passed as an input argument to a function, all of its component *values* are copied into the components of the function's corresponding formal parameter.
- When such a variable is used as an output argument, the address-of operator must be applied.
- The equality and inequality operators cannot be applied to a structured type as a unit.

Structured Input Parameter

print_planet(current_planet);

```
1.  /*
2.   * Displays with labels all components of a planet_t structure
3.   */
4.  void
5.  print_planet(planet_t pl) /* input - one planet structure */
6.  {
7.        printf("%s\n", pl.name);
8.        printf("  Equatorial diameter: %.0f km\n", pl.diameter);
9.        printf("  Number of moons: %d\n", pl.moons);
10.       printf("  Time to complete one orbit of the sun: %.2f years\n",
11.              pl.orbit_time);
12.       printf("  Time to complete one rotation on axis: %.4f hours\n",
13.              pl.rotation_time);
14. }
```


Comparing Two Structured Values

```
1.  #include <string.h>
2.
3.  /*
4.   * Determines whether or not the components of planet_1 and planet_2 match
5.   */
6.  int
7.  planet_equal(planet_t planet_1, /* input - planets to                    */
8.               planet_t planet_2) /*         compare                      */
9.  {
10.       return (strcmp(planet_1.name, planet_2.name) == 0   &&
11.               planet_1.diameter == planet_2.diameter      &&
12.               planet_1.moons == planet_2.moons            &&
13.               planet_1.orbit_time == planet_2.orbit_time  &&
14.               planet_1.rotation_time == planet_2.rotation_time);
15. }
```

Structured Output Argument

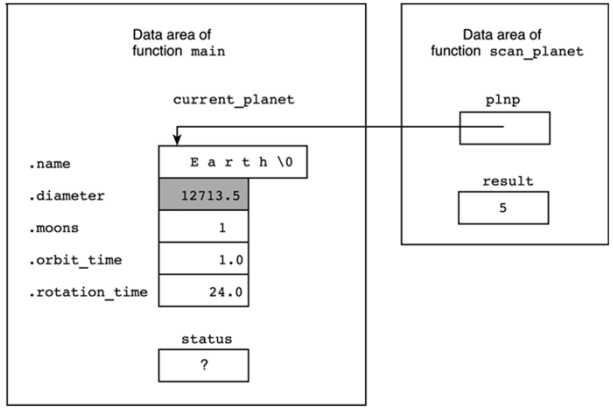
```
1.  /*
2.   * Fills a type planet_t structure with input data. Integer returned as
3.   * function result is success/failure/EOF indicator.
4.   *      1 => successful input of one planet
5.   *      0 => error encountered
6.   *      EOF => insufficient data before end of file
7.   * In case of error or EOF, value of type planet_t output argument is
8.   * undefined.
9.   */
10. int
11. scan_planet(planet_t *plnp) /* output - address of planet_t structure
12.                                to fill                              */
13. {
14.       int result;
15.
16.       result = scanf("%s%lf%d%lf%lf",  (*plnp).name,
17.                                        &(*plnp).diameter,
18.                                        &(*plnp).moons,
19.                                        &(*plnp).orbit_time,
20.                                        &(*plnp).rotation_time);
21.       if (result == 5)
22.             result = 1;
23.       else if (result != EOF)
24.             result = 0;
25.
26.       return (result);
27. }
```


Structured Output Argument (Cont'd)

TABLE 11.2 Step-by-Step Analysis of Reference &(*plnp).diameter

Reference	Type	Value
`plnp`	`planet_t *`	address of structure that `main` refers to as `current_planet`
`*plnp`	`planet_t`	structure that `main` refers to as `current_planet`
`(*plnp).diameter`	`double`	`12713.5`
`&(*plnp).diameter`	`double *`	address of colored component of structure that `main` refers to as `current_planet`

status = scan_planet(&current_planet);



Structure as Argument

- In order to use scanf to store a value in one component of the structure whose address is in plnp, we must carry out the following steps (in order):
  1. Follow the pointer in plnp to the structure.
  2. Select the component of interest.
  3. Unless this component is an array, get its address to pass to scanf.
- &*plnp.diameter would attempt step 2 before step 1.

Structure as Argument (Cont'd)

- Indirect component selection operator
  - the character sequence -> placed between a pointer variable and a component name creates a reference that follows the pointer to a structure and selects the component
- Two expressions are equivalent.

```
(*structp).component
structp->component
```


Structure as Argument (Cont'd)

-
```
result = scanf("%s%lf%d%lf%lf",
          plnp->name,
          &plnp->diameter,
          &plnp->moons,
          &plnp->orbit_time,
          &plnp->rotation_time);
```


Returning a Structured Result

- The function returns the *values* of all components.
  current_planet = get_planet();
- However, **scan_planet** with its ability to return an integer error code is the more generally useful function.

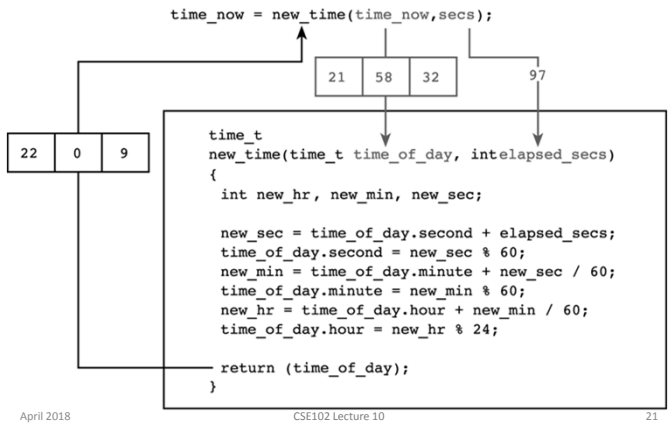
```
1.  /*
2.   * Gets and returns a planet_t structure
3.   */
4.  planet_t
5.  get_planet(void)
6.  {
7.        planet_t planet;
8.
9.        scanf("%s%lf%d%lf%lf",  planet.name,
10.                               &planet.diameter,
11.                               &planet.moons,
12.                               &planet.orbit_time,
13.                               &planet.rotation_time);
14.       return (planet);
15. }
```


Compute an Updated Time Value

```
typedef struct {
      int hour, minute, second;
   } time_t;
time_now = new_time(time_now, secs);
```

```
1.  /*
2.   * Computes a new time represented as a time_t structure
3.   * and based on time of day and elapsed seconds.
4.   */
5.  time_t
6.  new_time(time_t time_of_day,   /* input - time to be
7.                                    updated                    */
8.           int    elapsed_secs)  /* input - seconds since last update  */
9.  {
10.       int new_hr, new_min, new_sec;
11.
12.       new_sec = time_of_day.second + elapsed_secs;
13.       time_of_day.second = new_sec % 60;
14.       new_min = time_of_day.minute + new_sec / 60;
15.       time_of_day.minute = new_min % 60;
16.       new_hr = time_of_day.hour + new_min / 60;
17.       time_of_day.hour = new_hr % 24;
18.
19.       return (time_of_day);
20. }
```

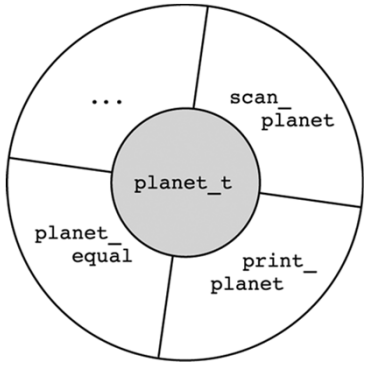
Structured Values as a Function Result



Abstract Data Type

- **Abstract Data Type (ADT)**
  a data type combined with a set of basic operations
- We must also provide basic operations for manipulating our own data types.
- If we take the time to define enough basic operations for a structure type, we then find it possible to think about a related problem at a higher level of abstraction.

Abstract Data Type



Type and Operators for Complex Numbers

```
1.  /*
2.   *  Operators to process complex numbers
3.   */
4.  #include <stdio.h>
5.  #include <math.h>
6.
7.  /*  User-defined complex number type */
8.  typedef struct {
9.        double real, imag;
10. } complex_t;
11.
12. int scan_complex(complex_t *c);
13. void print_complex(complex_t c);
14. complex_t add_complex(complex_t c1, complex_t c2);
15. complex_t subtract_complex(complex_t c1, complex_t c2);
16. complex_t multiply_complex(complex_t c1, complex_t c2);
17. complex_t divide_complex(complex_t c1, complex_t c2);
18. complex_t abs_complex(complex_t c);
19.
```

```
21. int
22. main(void)
23. {
24.       complex_t com1, com2;
25.
26.       /*  Gets two complex numbers                                    */
27.       printf("Enter the real and imaginary parts of a complex number\n");
28.       printf("separated by a space> ");
29.       scan_complex(&com1);
30.       printf("Enter a second complex number> ");
31.       scan_complex(&com2);
32.
33.       /*  Forms and displays the sum                                  */
34.       printf("\n");
35.       print_complex(com1);
36.       printf("  +  ");
37.       print_complex(com2);
38.       printf("  =  ");
39.       print_complex(add_complex(com1, com2));
40.
41.       /*  Forms and displays the difference                           */
42.       printf("\n\n");
```

(continued)


```
43.       print_complex(com1);
44.       printf("  -  ");
45.       print_complex(com2);
46.       printf("  =  ");
47.       print_complex(subtract_complex(com1, com2));
48.
49.       /*  Forms and displays the absolute value of the first number   */
50.       printf("\n\n|");
51.       print_complex(com1);
52.       printf("|  =  ");
53.       print_complex(abs_complex(com1));
54.       printf("\n");
55.
56.       return (0);
57. }
```



```
59. /*
60.  *  Complex number input function returns standard scanning error code
61.  *    1 => valid scan, 0 => error, negative EOF value => end of file
62.  */
63. int
64. scan_complex(complex_t *c) /* output - address of complex variable to
65.                                      fill                            */
66. {
67.       int status;
68.
69.       status = scanf("%lf%lf", &c->real, &c->imag);
70.       if (status == 2)
71.             status = 1;
72.       else if (status != EOF)
73.             status = 0;
74.
75.       return (status);
76. }
77.
```



```
78. /*
79.  *  Complex output function displays value as (a + bi) or (a - bi),
80.  *  dropping a or b if they round to 0 unless both round to 0
81.  */
82. void
83. print_complex(complex_t c) /* input - complex number to display   */
84. {
85.       double a, b;
86.       char   sign;
87.
88.       a = c.real;
89.       b = c.imag;
90.
91.       printf("(");
92.
93.       if (fabs(a) < .005  &&  fabs(b) < .005) {
94.             printf("%.2f", 0.0);
95.       } else if (fabs(b) < .005) {
96.             printf("%.2f", a);
97.       } else if (fabs(a) < .005) {
98.             printf("%.2fi", b);
99.       } else {
100.            if (b < 0)
101.                  sign = '-';
102.            else
103.                  sign = '+';
104.            printf("%.2f %c %.2fi", a, sign, fabs(b));
105.      }
106.
```

```
/*
 *  Returns sum of complex values c1 and c2
 */
complex_t
add_complex(complex_t c1, complex_t c2) /* input - values to add    */
{
      complex_t csum;

      csum.real = c1.real + c2.real;
      csum.imag = c1.imag + c2.imag;
      return (csum);
}
```



```
/*
 *  Returns difference c1 - c2
 */
complex_t
subtract_complex(complex_t c1, complex_t c2) /* input parameters    */
{
      complex_t cdiff;
      cdiff.real = c1.real - c2.real;
      cdiff.imag = c1.imag - c2.imag;

      return (cdiff);
}
```



```
/*  ** Stub **
 *  Returns product of complex values c1 and c2
 */
complex_t
multiply_complex(complex_t c1, complex_t c2) /* input parameters    */
{
      printf("Function multiply_complex returning first argument\n");
      return (c1);
}

/*  ** Stub **
 *  Returns quotient of complex values (c1 / c2)
 */
complex_t
divide_complex(complex_t c1, complex_t c2) /* input parameters     */
{
      printf("Function divide_complex returning first argument\n");
      return (c1);
}
```

(continued)


```
/*
 *  Returns absolute value of complex number c
 */
complex_t
abs_complex(complex_t c) /* input parameter                        */
{
      complex_t cabs;

      cabs.real = sqrt(c.real * c.real + c.imag * c.imag);
      cabs.imag = 0;

      return (cabs);
}
```

Parallel Arrays & Array of Structures

- **Parallel Arrays**

```
int id[50]; /* id numbers and */
double gpa[50]; /* gpa's of up to 50 students */

double x[NUM_PTS], /* (x,y) coordinates of */,
       y[NUM_PTS]; /* up to NUM_PTS points */
```

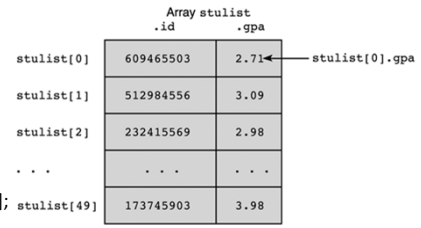
- **Array of Structures**

  A more natural and convenient organization is to group the information in a structure whose type we define.

Array of Structures

- Ex. 1

```
#define MAX_STU 50
typedef struct {
  int id;
  double gpa;
} student_t;
. . .
{
  student_t stulist[MAX_STU];
```



- Ex. 2

```
#define NUM_PTS 10
typedef struct {
  double x, y;
} point_t;
. . .
{
  point_t polygon[NUM_PTS];
```


Universal Measurement Conversion

Data file units.dat:

```
miles           mi          distance        1609.3
kilometers      km          distance        1000
yards           yd          distance        0.9144
meters          m           distance        1
quarts          qt          liquid_volume   0.94635
liters          l           liquid_volume   1
gallons         gal         liquid_volume   3.7854
milliliters     ml          liquid_volume   0.001
kilograms       kg          mass            1
grams           g           mass            0.001
slugs           slugs       mass            0.14594
```


Universal Measurement Conversion

```
/*
 *  Converts measurements given in one unit to any other unit of the same
 *  category that is listed in the database file, units.dat.
 *  Handles both names and abbreviations of units.
 */
#include <stdio.h>
#include <string.h>

#define NAME_LEN    30      /* storage allocated for a unit name              */
#define ABBREV_LEN  15      /* storage allocated for a unit abbreviation      */
#define CLASS_LEN   20      /* storage allocated for a measurement class      */
#define NOT_FOUND   -1      /* value indicating unit not found                */
#define MAX_UNITS   20      /* maximum number of different units handled      */

typedef struct {                /* unit of measurement type                   */
      char   name[NAME_LEN];    /* character string such as "milligrams"      */
      char   abbrev[ABBREV_LEN];/* shorter character string such as "mg"      */
      char   class[CLASS_LEN];  /* character string such as "pressure",
                                   "distance", "mass"                         */
      double standard;          /* number of standard units equivalent
                                   to this unit                               */
} unit_t;

int  fscan_unit(FILE *filep, unit_t *unitp);
void load_units(int unit_max, unit_t units[], int *unit_sizep);
int  search(const unit_t units[], const char *target, int n);
double convert(double quantity, double old_stand, double new_stand);
```

Universal Measurement Conversion

```
29.  int
30.  main(void)
31.  {
32.        unit_t units[MAX_UNITS];    /* units classes and conversion factors*/
33.        int    num_units;           /* number of elements of units in use  */
34.        char   old_units[NAME_LEN], /* units to convert (name or abbrev)   */
35.               new_units[NAME_LEN]; /* units to convert to (name or abbrev)*/
36.        int    status;              /* input status                        */
37.        double quantity;            /* value to convert                    */
38.
```

(continued)

Universal Measurement Conversion

```
39.        int    old_index,           /* index of units element where
40.                                       old_units found                   */
41.               new_index;           /* index where new_units found       */
42.
43.        /*  Load units of measurement database                           */
44.        load_units(MAX_UNITS, units, &num_units);
45.
46.        /*  Convert quantities to desired units until data format error
47.            (including error code returned when q is entered to quit)    */
48.        printf("Enter a conversion problem or q to quit.\n");
49.        printf("To convert 25 kilometers to miles, you would enter\n");
50.        printf("> 25 kilometers miles\n");
51.        printf("   or, alternatively,\n");
52.        printf("> 25 km mi\n> ");
53.
```



```
54.        for  (status = scanf("%lf%s%s", &quantity, old_units, new_units);
55.              status == 3;
56.              status = scanf("%lf%s%s", &quantity, old_units, new_units)) {
57.           printf("Attempting conversion of %.4f %s to %s . . .\n",
58.                  quantity, old_units, new_units);
59.           old_index = search(units, old_units, num_units);
60.           new_index = search(units, new_units, num_units);
61.           if (old_index == NOT_FOUND)
62.                 printf("Unit %s not in database\n", old_units);
63.           else if (new_index == NOT_FOUND)
64.                 printf("Unit %s not in database\n", new_units);
65.           else if (strcmp(units[old_index].class,
66.                           units[new_index].class) != 0)
67.                 printf("Cannot convert %s (%s) to %s (%s)\n",
68.                        old_units, units[old_index].class,
69.                        new_units, units[new_index].class);
70.           else
71.                 printf("%.4f%s  =  %.4f %s\n", quantity, old_units,
72.                        convert(quantity, units[old_index].standard,
73.                                units[new_index].standard),
74.                        new_units);
75.           printf("\nEnter a conversion problem or q to quit.\n> ");
76.        }
77.
78.        return (0);
79.  }
80.
```

(continued)


```
81.  /*
82.   *  Gets data from a file to fill output argument
83.   *  Returns standard error code:  1 => successful input,  0 => error,
84.   *                                negative EOF value => end of file
85.   */
86.  int
87.  fscan_unit(FILE   *filep,  /*  input - input file pointer        */
88.             unit_t *unitp)  /*  output - unit_t structure to fill */
89.  {
90.        int status;
91.
92.        status = fscanf(filep, "%s%s%s%lf", unitp->name,
93.                                            unitp->abbrev,
94.                                            unitp->class,
95.                                            &unitp->standard);
96.
97.        if (status == 4)
98.              status = 1;
99.        else if (status != EOF)
100.             status = 0;
101.
102.       return (status);
103. }
104.
```

```
/*
 *  Opens database file units.dat and gets data to place in units until end
 *  of file is encountered.  Stops input prematurely if there are more than
 *  unit_max data values in the file or if invalid data is encountered.
 */
void
load_units(int         unit_max,   /* input - declared size of units     */
           unit_t      units[],    /* output - array of data             */
           int        *unit_sizep) /* output - number of data values
                                      stored in units                    */
{
      FILE  *inp;
      unit_t data;
      int    i, status;

      /*  Gets database of units from file                               */
      inp = fopen("units.dat", "r");
      i = 0;

      for  (status = fscan_unit(inp, &data);
            status == 1  &&  i < unit_max;
            status = fscan_unit(inp, &data)) {
           units[i++] = data;
      }
      fclose(inp);

      /* Issue error message on premature exit                           */
      if (status == 0) {
            printf("\n*** Error in data format ***\n");
            printf("*** Using first %d data values ***\n", i);
      } else if (status != EOF) {
            printf("\n*** Error: too much data in file ***\n");
            printf("*** Using first %d data values ***\n", i);
      }

      /* Send back size of used portion of array                         */
      *unit_sizep = i;
}
```



```
/*
 *  Searches for target key in name and abbrev components of first n
 *     elements of array units
 *  Returns index of structure containing target or NOT_FOUND
 */
int
search(const unit_t units[],  /*  array of unit_t structures to search   */
       const char  *target,   /*  key searched for in name and abbrev
                                 components                              */
       int          n)        /*  number of array elements to search     */
{
      int i,
          found = 0,     /*  whether or not target has been found        */
          where;         /*  index where target found or NOT_FOUND       */

      /*  Compare name and abbrev components of each element to target   */
      i = 0;
      while (!found && i < n) {
          if (strcmp(units[i].name,   target) == 0  ||
              strcmp(units[i].abbrev, target) == 0)
                found = 1;
          else
                ++i;
      }
      /* Return index of element containing target or NOT_FOUND          */
      if (found)
            where = i;
      else
            where = NOT_FOUND;
      return (where);
}
```



```
/*
 *  Converts one measurement to another given the representation of both
 *  in a standard unit.  For example, to convert 24 feet to yards given a
 *  standard unit of inches:  quantity = 24, old_stand = 12 (there are 12
 *  inches in a foot), new_stand = 36 (there are 36 inches in a yard),
 *  result is 24 * 12 / 36 which equals 8
 */
double
convert(double quantity,     /* value to convert                         */
        double old_stand,    /* number of standard units in one of
                                quantity's original units                */
        double new_stand)    /* number of standard units in 1 new unit   */
{
      return (quantity * old_stand / new_stand);
}
```


Sample run:

```
Enter a conversion problem or q to quit.
To convert 25 kilometers to miles, you would enter
> 25 kilometers miles
    or, alternatively,
> 25 km mi
> 450 km miles
Attempting conversion of 450.0000 km to miles . . .
450.0000km  =  279.6247 miles

Enter a conversion problem or q to quit.
> 2.5 qt l
Attempting conversion of 2.5000 qt to l . . .
2.5000qt  =  2.3659 l

Enter a conversion problem or q to quit.
> 100 meters gallons
Attempting conversion of 100.0000 meters to gallons . . .
Cannot convert meters (distance) to gallons (liquid_volume)

Enter a conversion problem or q to quit.
> 1234 mg g
Attempting conversion of 1234.0000 mg to g . . .
Unit mg not in database

Enter a conversion problem or q to quit.
> q
```

Union Types

Union: Data object that can be interpreted in a variety of ways
- EX: a number can be real number (double) or an integer (int)

- Allows one chunk of memory to be interpreted in multiple ways

```
typedef union {
        int wears_wig;
        char color[20];
} hair_t;
hair_t hair_data;
```

- *hair_data* does not contain both *wears_wig* and *color* components, but *either* a *wears_wig* component referenced by *hair_data.wears_wig*, *or* a *color* component referenced by *hair_data.color*.

- The amount of memory is determined by the largest component of the union.

- How to determine interpretation?
  - How to determine whether to use wears_wig or color?

Union Types

- Data object that can be interpreted in a variety of ways
  - EX: number

```
typedef union {
    int wears_wig;
    char color[20];
} hair_t;

hair_t his_hair;
```

- Memory requirement is determined by the largest component.
- How to determine interpretation?
  - How to determine whether to use wears_wig or color?

Union Types

- Data object that can be interpreted in a variety of ways

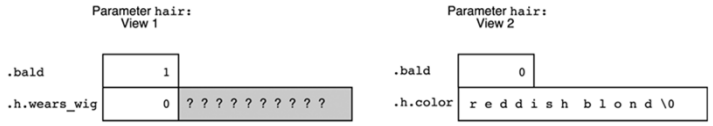
```
typedef union {
    int wears_wig;
    char color[20];
} hair_t;

typedef struct {
    int bald;
    hair_t h;
} hair_info_t;
hair_info_t his_hair;
```

- Referencing the appropriate union component is *always* the programmer's responsibility; C can do no checking of the validity of such a component reference.

Displays a Structure with a Union

```
1.  void
2.  print_hair_info(hair_info_t hair) /* input - structure to display          */
3.  {
4.      if (hair.bald) {
5.          printf("Subject is bald");
6.          if (hair.h.wears_wig)
7.              printf(", but wears a wig.\n");
8.          else
9.              printf(" and does not wear a wig.\n");
10.     } else {
11.         printf("Subject's hair color is %s.\n", hair.h.color);
12.     }
13. }
```

Two Interpretations of Parameter hair



Compute Area and Perimeter

```
1.  /*
2.   *  Computes the area and perimeter of a variety of geometric figures.
3.   */
4.
5.  #include <stdio.h>
6.  #define PI 3.14159
7.
8.  /*  Types defining the components needed to represent each shape.          */
9.  typedef struct {
10.       double area,
11.              circumference,
12.              radius;
13. } circle_t;
14.
15. typedef struct {
16.       double area,
17.              perimeter,
18.              width,
19.              height;
20. } rectangle_t;
21.
22. typedef struct {
23.       double area,
24.              perimeter,
25.              side;
26. } square_t;
```


Compute Area and Perimeter

```
28. /*  Type of a structure that can be interpreted a different way for
29.       each shape                                                    */
30. typedef union {
31.       circle_t    circle;
32.       rectangle_t rectangle;
33.       square_t    square;
34. } figure_data_t;
35.
36. /*  Type containing a structure with multiple interpretations along with
37.  *  a component whose value indicates the current valid interpretation   */
38. typedef struct {
39.       char          shape;
40.       figure_data_t fig;
41. } figure_t;
42.
43. figure_t get_figure_dimensions(void);
44. figure_t compute_area(figure_t object);
45. figure_t compute_perim(figure_t object);
46. void print_figure(figure_t object);
47.
```



```
48. int
49. main(void)
50. {
51.       figure_t onefig;
52.
53.       printf("Area and Perimeter Computation Program\n");
54.
55.       for  (onefig = get_figure_dimensions();
56.             onefig.shape != 'Q';
57.             onefig = get_figure_dimensions()) {
58.           onefig = compute_area(onefig);
59.           onefig = compute_perim(onefig);
60.           print_figure(onefig);
61.       }
62.
63.       return (0);
64. }
65.
66. /*
67.  *  Prompts for and stores the dimension data necessary to compute a
68.  *  figure's area and perimeter.  Figure returned contains a 'Q' in the
69.  *  shape component when signaling end of data.
70.  */
71. figure_t
72. get_figure_dimensions(void)
73. {
74.       figure_t object;
```

(continued)

```
75.       printf("Enter a letter to indicate the object shape or Q to quit.\n");
76.       printf("C (circle),  R (rectangle),  or S (square)> ");
77.       object.shape = getchar();
78.
79.       switch (object.shape) {
80.       case 'C':
81.       case 'c':
82.             printf("Enter radius> ");
83.             scanf("%lf", &object.fig.circle.radius);
84.             break;
85.
86.       case 'R':
87.       case 'r':
88.             printf("Enter height> ");
89.             scanf("%lf", &object.fig.rectangle.height);
90.             printf("Enter width> ");
91.             scanf("%lf", &object.fig.rectangle.width);
92.             break;
93.
94.       case 'S':
95.       case 's':
96.             printf("Enter length of a side> ");
97.             scanf("%lf", &object.fig.square.side);
98.             break;
99.
100.      default:  /*  Error is treated as a QUIT  */
101.            object.shape = 'Q';
102.      }
103.
104.      return (object);
105. }
```



```
107. /*
108.  *  Computes the area of a figure given relevant dimensions.  Returns
109.  *  figure with area component filled.
110.  *  Pre:  value of shape component is one of these letters: CcRrSs
111.  *        necessary dimension components have values
112.  */
113. figure_t
114. compute_area(figure_t object)
115. {
116.       switch (object.shape) {
117.       case 'C':
118.       case 'c':
119.             object.fig.circle.area = PI * object.fig.circle.radius *
120.                                      object.fig.circle.radius;
121.             break;
122.
123.       case 'R':
124.       case 'r':
125.             object.fig.rectangle.area = object.fig.rectangle.height *
126.                                         object.fig.rectangle.width;
127.             break;
128.
129.       case 'S':
130.       case 's':
131.             object.fig.square.area = object.fig.square.side *
132.                                      object.fig.square.side;
133.             break;
134.
135.       default:
136.             printf("Error in shape code detected in compute_area\n");
137.       }
138.
139.       return (object);
140. }
```