Computer Programming

Structure and Union Types

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.

Structures

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

Structure definition

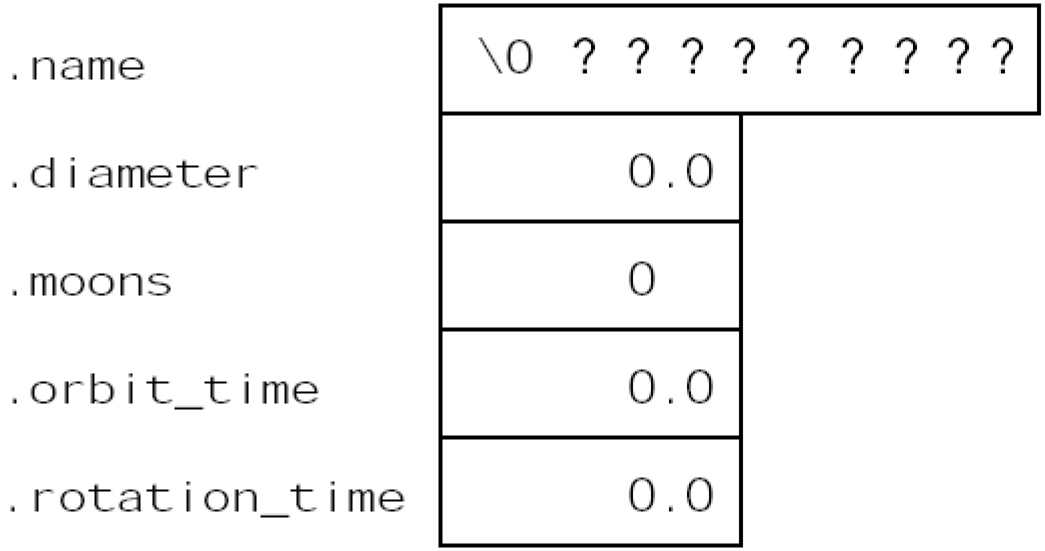
```
typedef struct {
        char      name[20];
        double    diameter;
        int       moons;
        double    orbit_time,
                  rotation_time;
} planet_t;

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};*

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

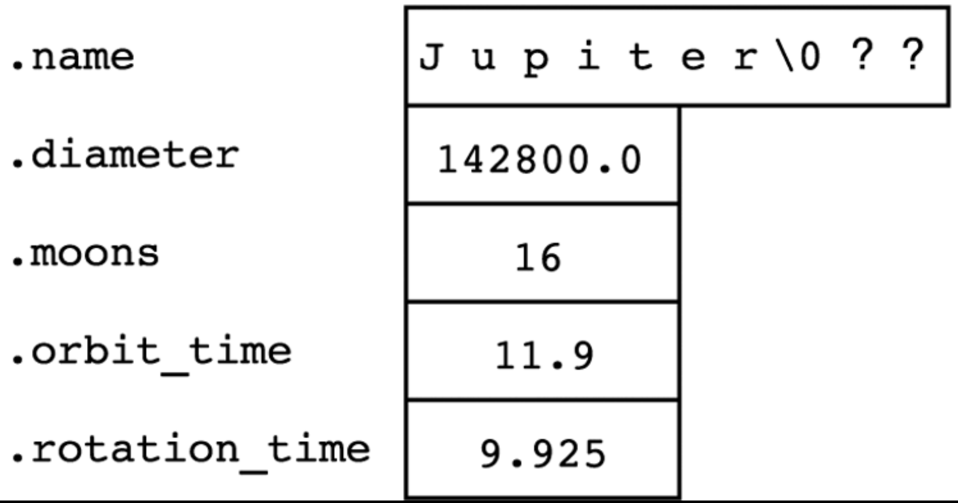
```
typedef struct {
    double diameter;
    planet_t planets[9];
    char galaxy[STRSIZ];
} solar_sys_t;
```

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;
```

- To help reduce confusion, choose user-defined type names ending in the suffix **_t**.

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

Structures as Arguments

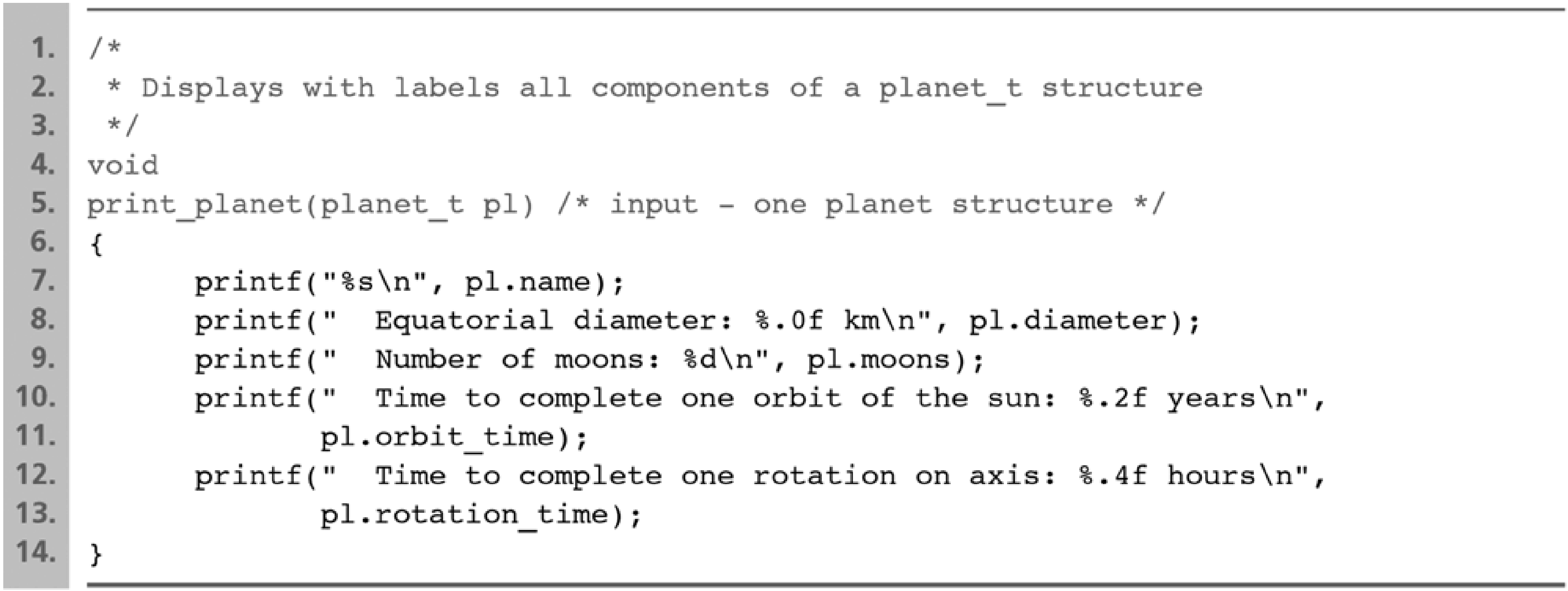
- When a structured variable is passed as an input argument to a function, all of its component *values* are copied into the components of the function’s corresponding formal parameter.

- 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.

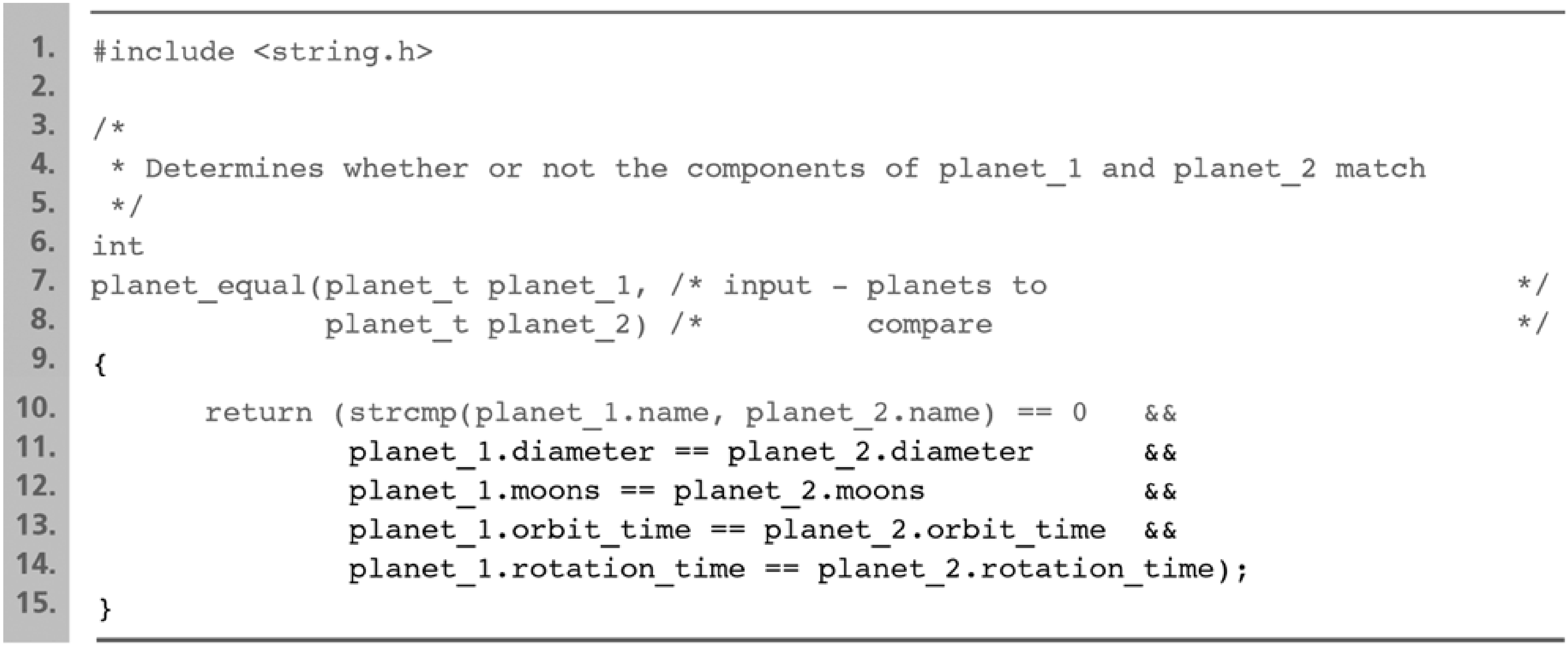
Structured Input Parameter

print_planet(current_planet);



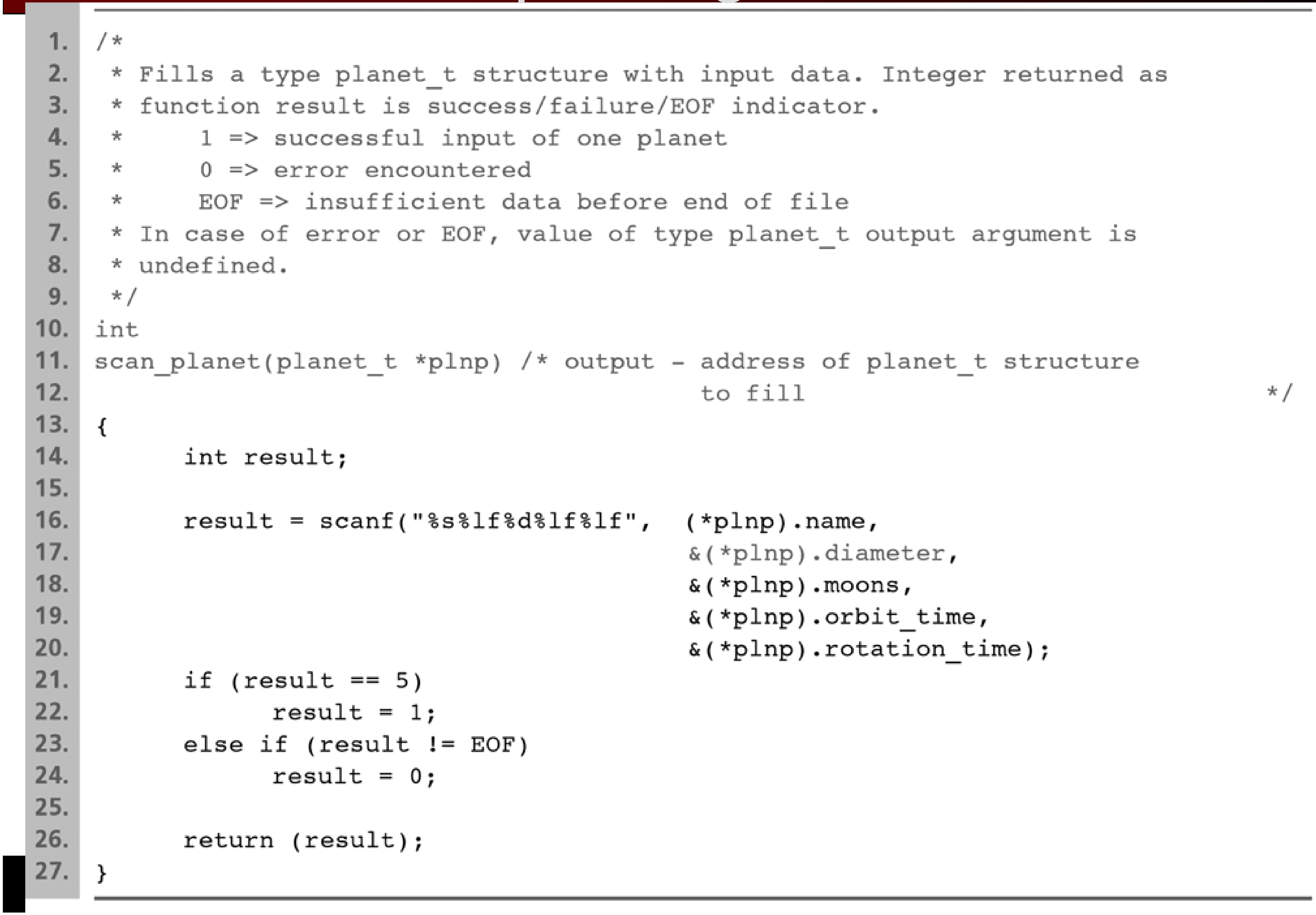
```
/*
 * Displays with labels all components of a planet_t structure
 */
void
print_planet(planet_t pl) /* input - one planet structure */
{
      printf("%s\n", pl.name);
      printf("  Equatorial diameter: %.0f km\n", pl.diameter);
      printf("  Number of moons: %d\n", pl.moons);
      printf("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

Comparing Two Structured Values



```
#include <string.h>

/*
 * Determines whether or not the components of planet_1 and planet_2 match
 */
int
planet_equal(planet_t planet_1, /* input - planets to                        */
             planet_t planet_2) /*         compare                           */
{
      return (strcmp(planet_1.name, planet_2.name) == 0   &&
              planet_1.diameter == planet_2.diameter      &&
              planet_1.moons == planet_2.moons            &&
              planet_1.orbit_time == planet_2.orbit_time  &&
              planet_1.rotation_time == planet_2.rotation_time);
}
```

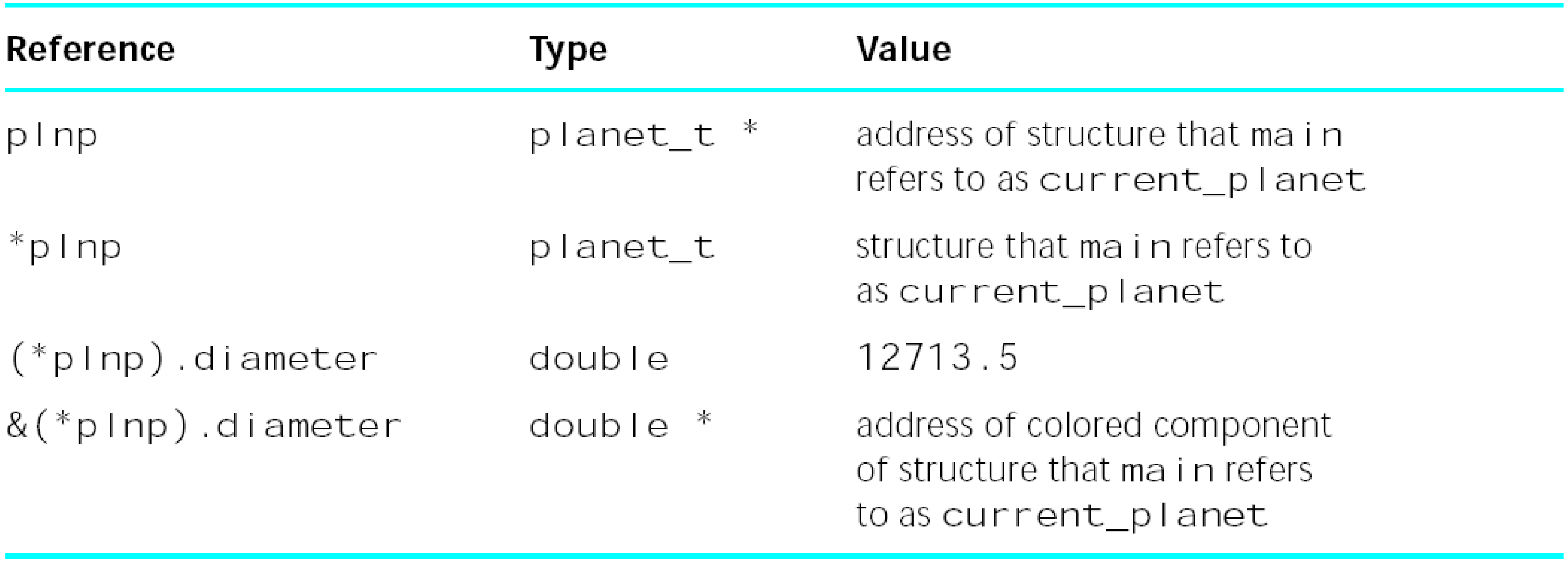
Structured Output Argument



```
/*
 * Fills a type planet_t structure with input data. Integer returned as
 * function result is success/failure/EOF indicator.
 *      1 => successful input of one planet
 *      0 => error encountered
 *      EOF => insufficient data before end of file
 * In case of error or EOF, value of type planet_t output argument is
 * undefined.
 */
int
scan_planet(planet_t *plnp) /* output - address of planet_t structure
                                        to fill                        */
{
      int result;

      result = scanf("%s%lf%d%lf%lf",  (*plnp).name,
                                       &(*plnp).diameter,
                                       &(*plnp).moons,
                                       &(*plnp).orbit_time,
                                       &(*plnp).rotation_time);
      if (result == 5)
            result = 1;
      else if (result != EOF)
            result = 0;

      return (result);
}
```

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

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**.

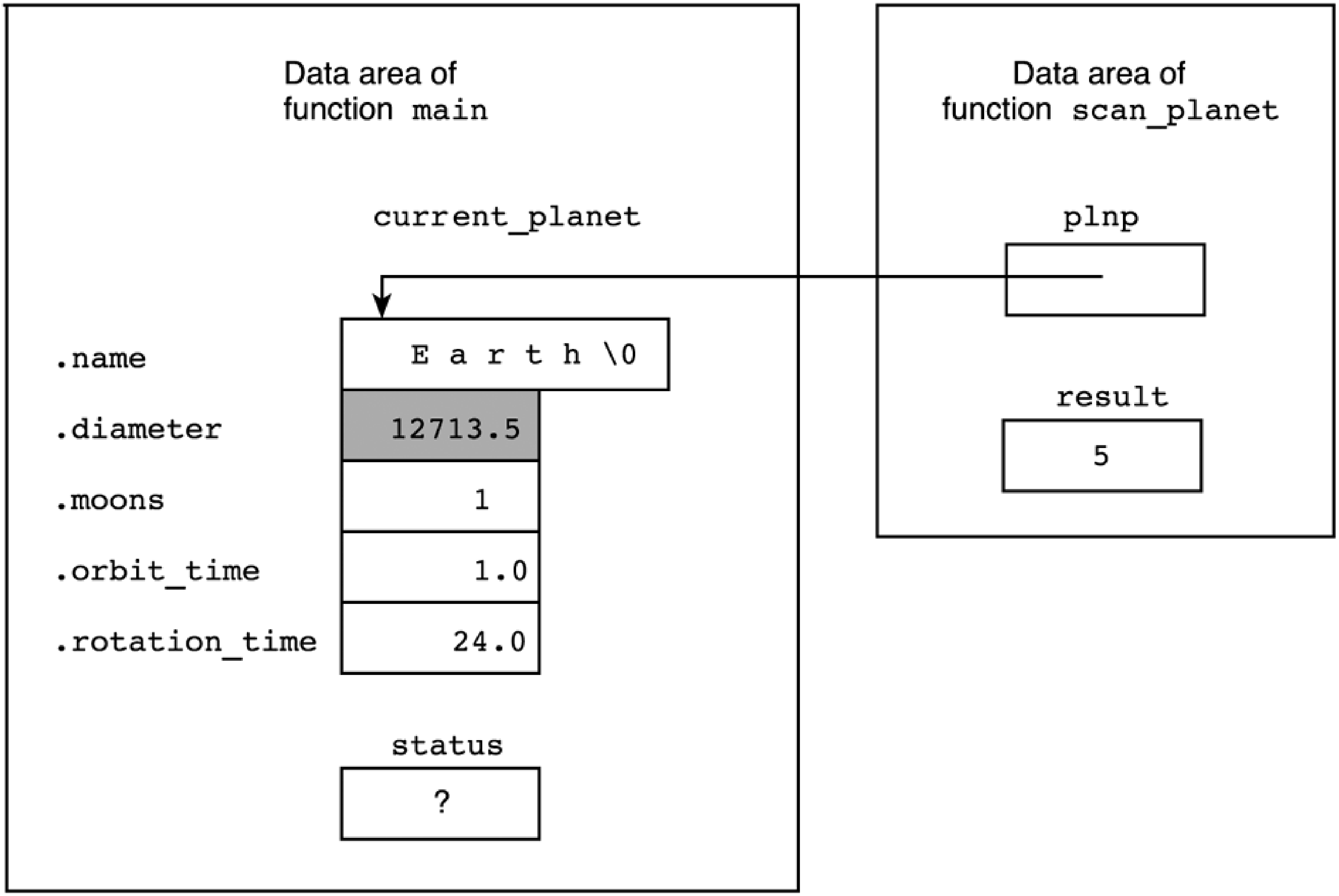
Structure as Argument (Cont'd)

- **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);
```

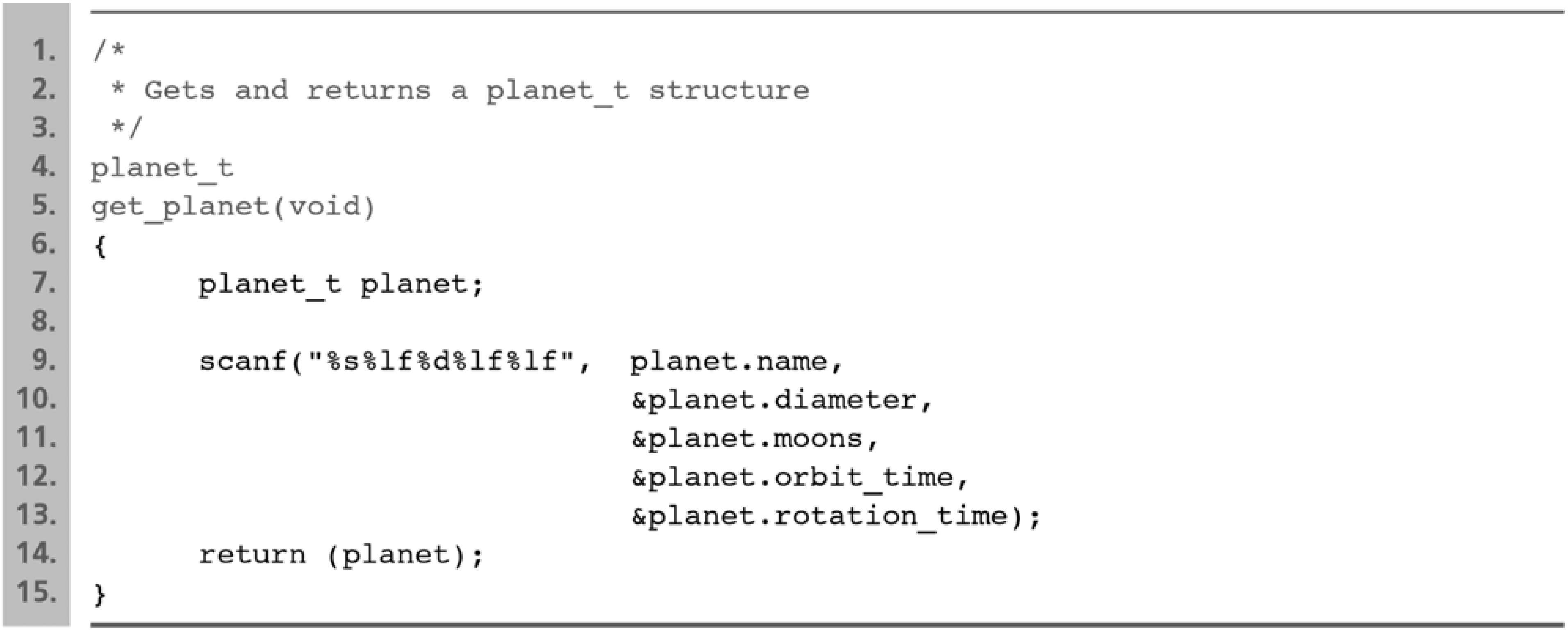
status = scan_planet(&current_planet);



Returning a Structured Result

- The function returns the *values* of all components.
- *current_planet = get_planet();*

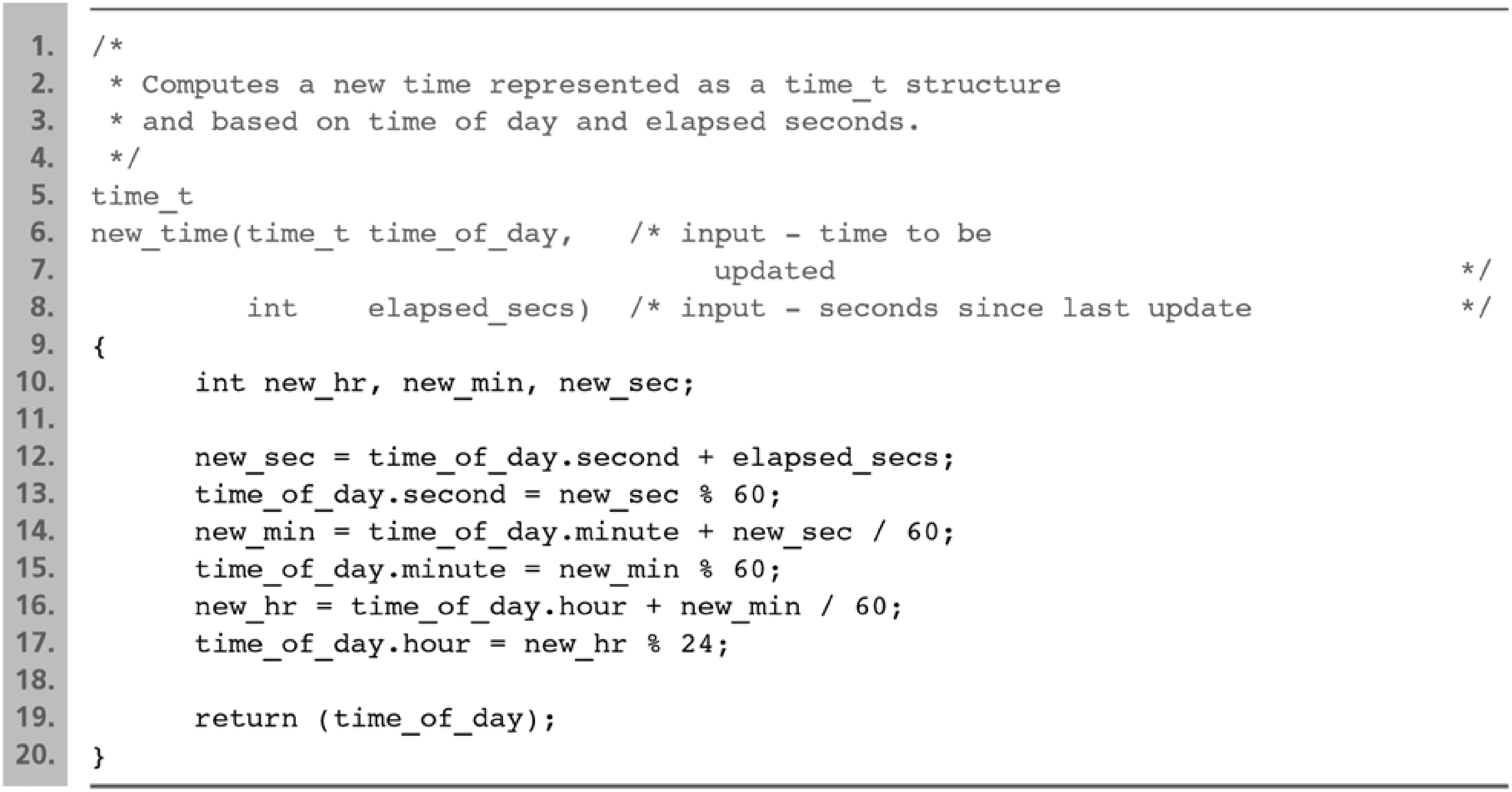
However, *scan_planet* with its ability to return an integer error code is the more generally useful function.



```
/*
 * Gets and returns a planet_t structure
 */
planet_t
get_planet(void)
{
      planet_t planet;

      scanf("%s%lf%d%lf%lf",  planet.name,
                              &planet.diameter,
                              &planet.moons,
                              &planet.orbit_time,
                              &planet.rotation_time);
      return (planet);
}
```

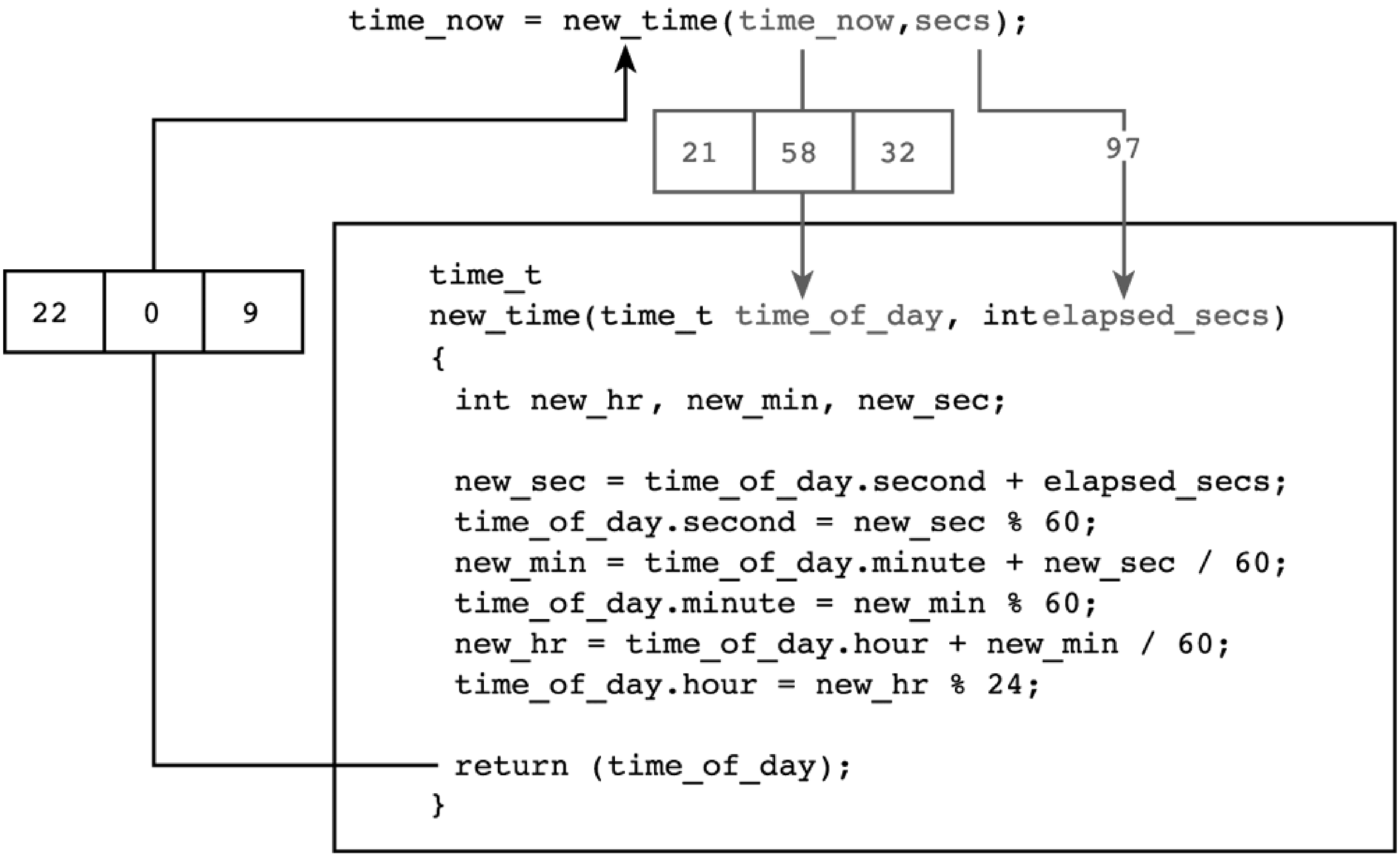
Compute an Updated Time Value

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



```
/*
 * Computes a new time represented as a time_t structure
 * and based on time of day and elapsed seconds.
 */
time_t
new_time(time_t time_of_day,   /* input - time to be
                                    updated                                   */
         int    elapsed_secs)  /* input - seconds since last update           */
{
      int new_hr, new_min, new_sec;

      new_sec = time_of_day.second + elapsed_secs;
      time_of_day.second = new_sec % 60;
      new_min = time_of_day.minute + new_sec / 60;
      time_of_day.minute = new_min % 60;
      new_hr = time_of_day.hour + new_min / 60;
      time_of_day.hour = new_hr % 24;

      return (time_of_day);
}
```

Structured Values as a Function Result



```
time_now = new_time(time_now,secs);
```

```
time_t
new_time(time_t time_of_day, intelapsed_secs)
{
 int new_hr, new_min, new_sec;

 new_sec = time_of_day.second + elapsed_secs;
 time_of_day.second = new_sec % 60;
 new_min = time_of_day.minute + new_sec / 60;
 time_of_day.minute = new_min % 60;
 new_hr = time_of_day.hour + new_min / 60;
 time_of_day.hour = new_hr % 24;

 return (time_of_day);
}
```

Abstract Data Type

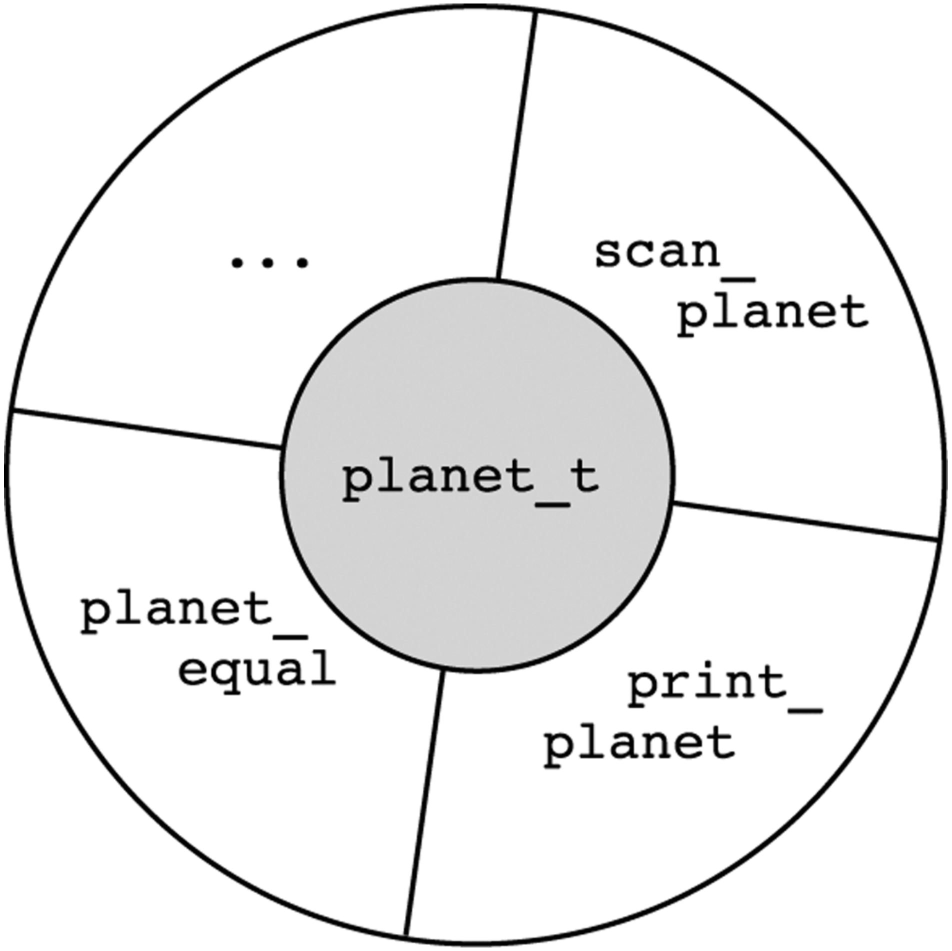
- **Abstract Data Type (ADT)**

  a data type combined with a set of basic operations

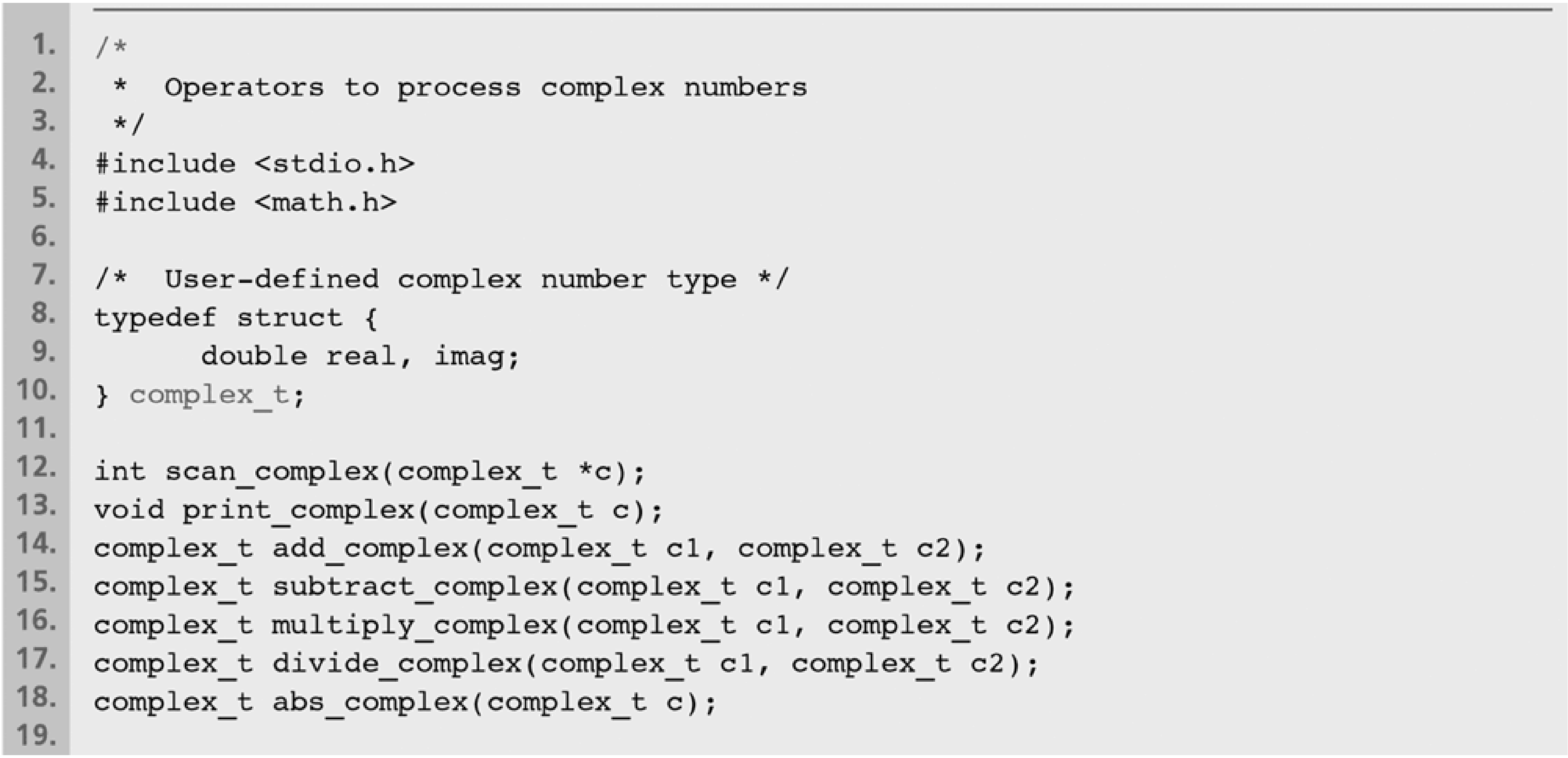
- We must also provide basic operations for manipulating our own data types.

- 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.

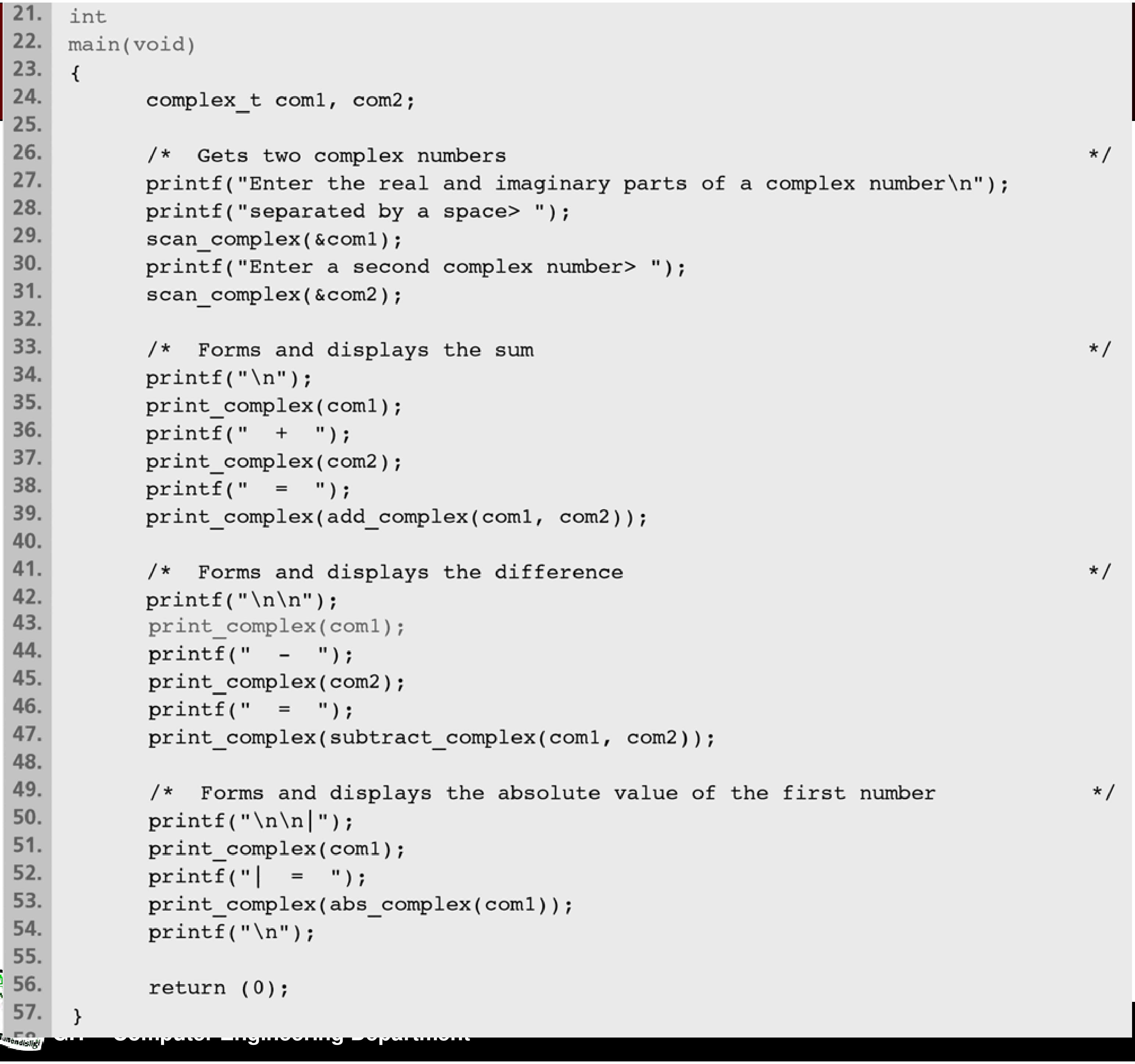
Abstract Data Type



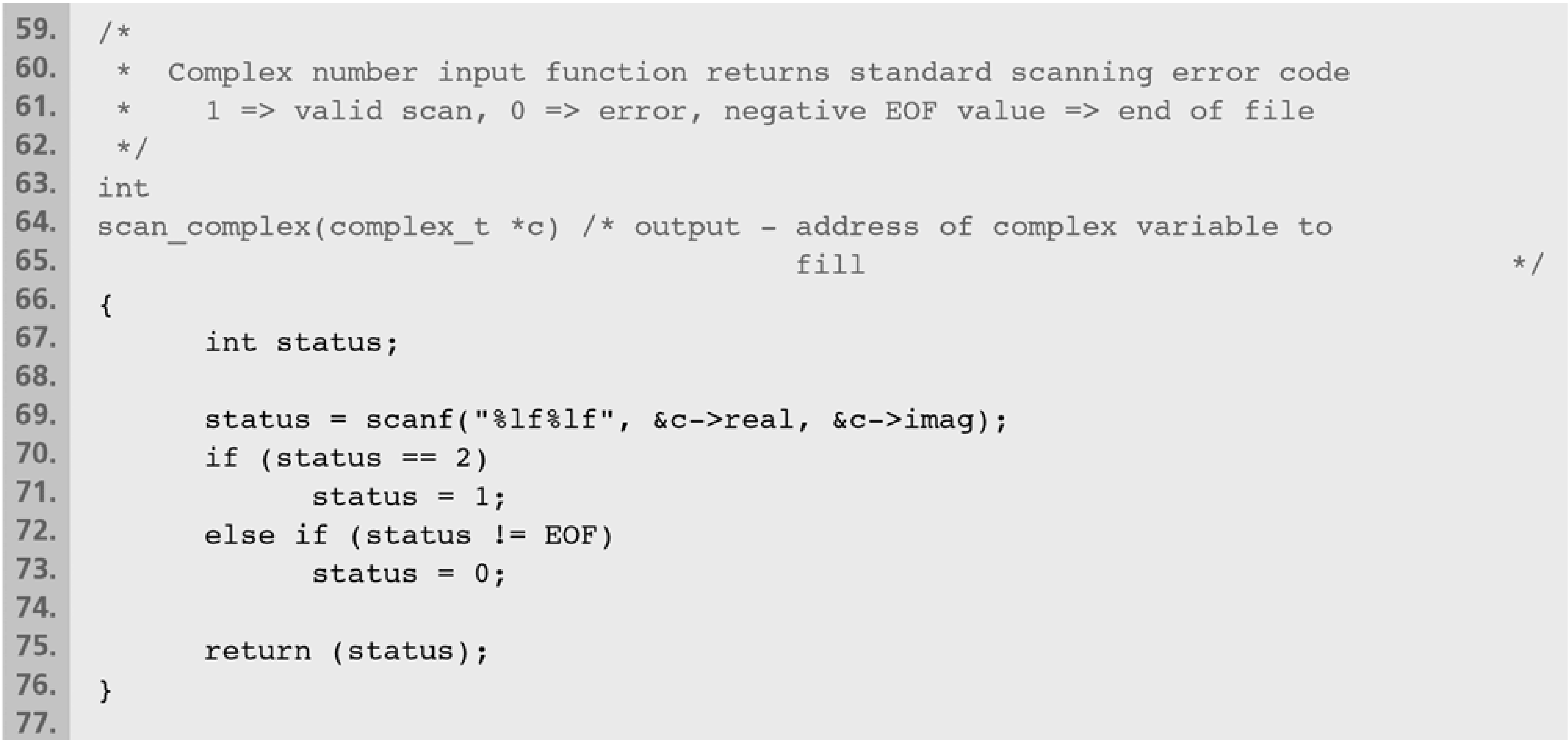
Type and Operators for Complex Numbers



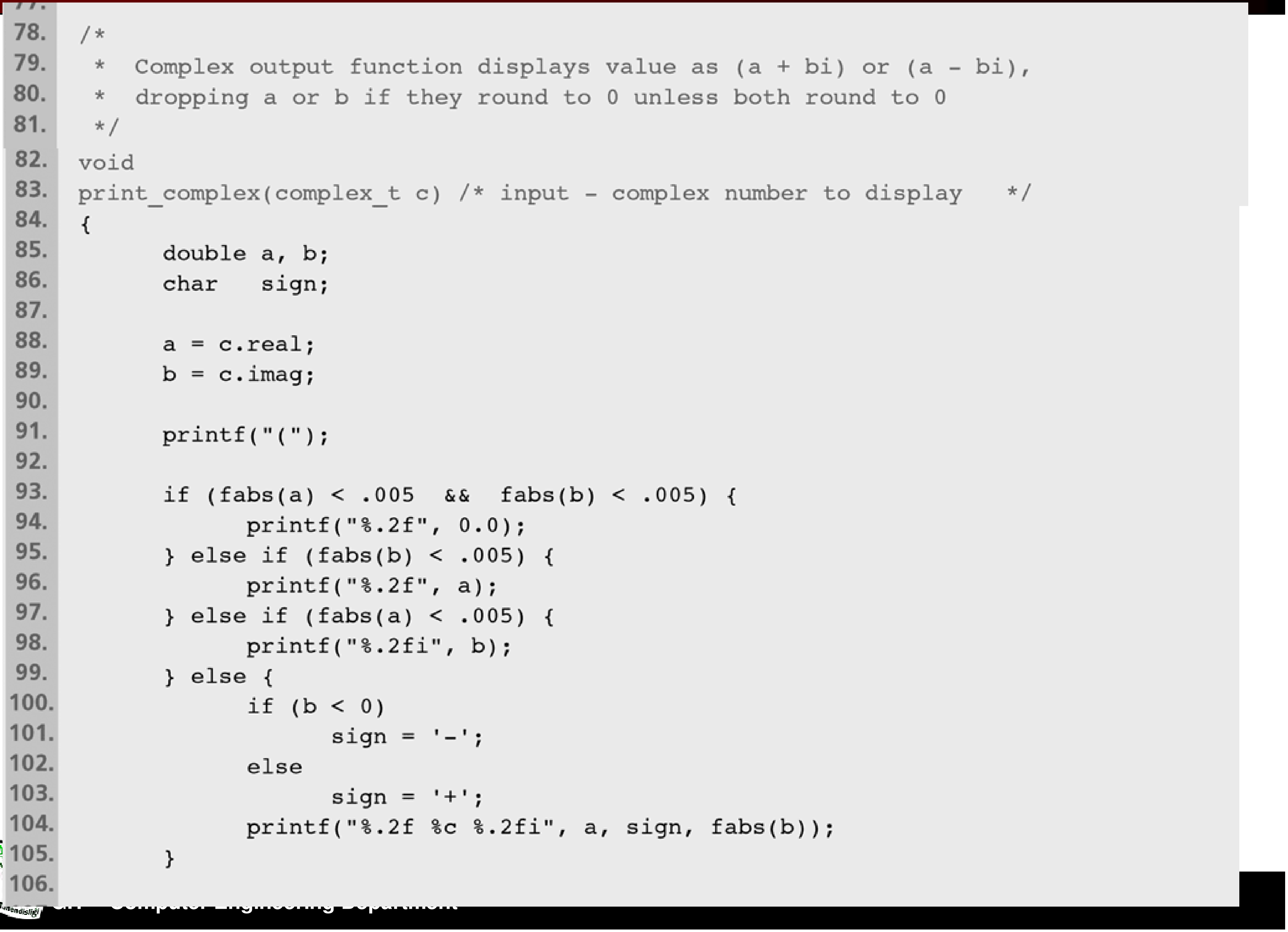
```c
/*
 *  Operators to process complex numbers
 */
#include <stdio.h>
#include <math.h>

/*  User-defined complex number type */
typedef struct {
      double real, imag;
} complex_t;

int scan_complex(complex_t *c);
void print_complex(complex_t c);
complex_t add_complex(complex_t c1, complex_t c2);
complex_t subtract_complex(complex_t c1, complex_t c2);
complex_t multiply_complex(complex_t c1, complex_t c2);
complex_t divide_complex(complex_t c1, complex_t c2);
complex_t abs_complex(complex_t c);
```



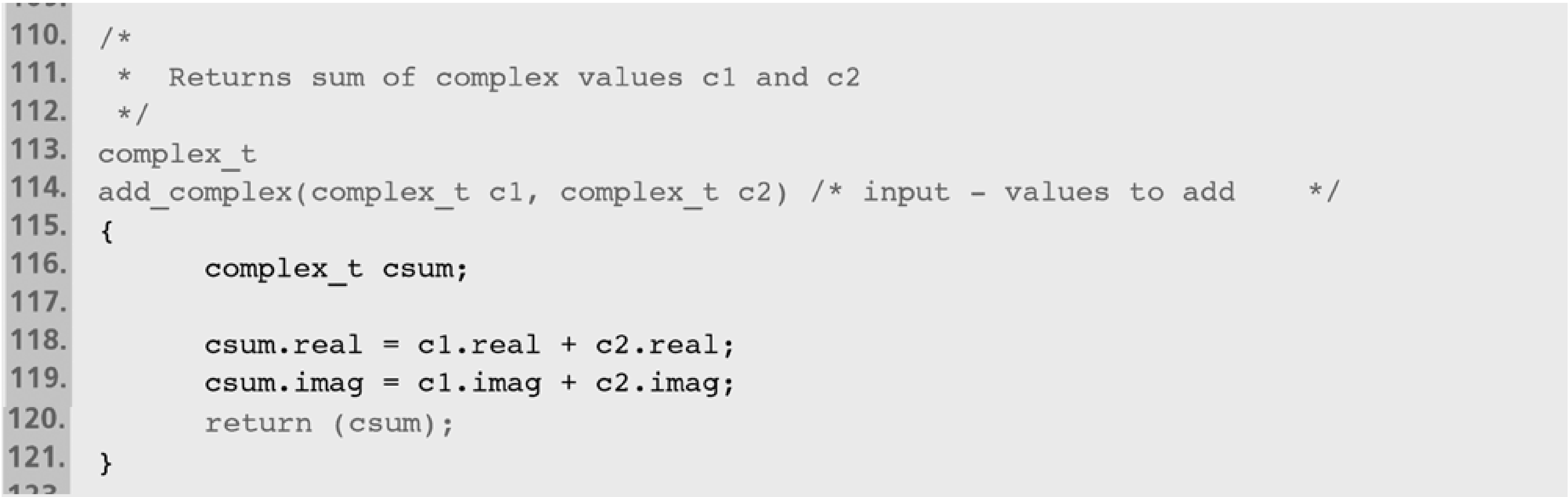
```c
int
main(void)
{
      complex_t com1, com2;

      /*  Gets two complex numbers                                             */
      printf("Enter the real and imaginary parts of a complex number\n");
      printf("separated by a space> ");
      scan_complex(&com1);
      printf("Enter a second complex number> ");
      scan_complex(&com2);

      /*  Forms and displays the sum                                           */
      printf("\n");
      print_complex(com1);
      printf("  +  ");
      print_complex(com2);
      printf("  =  ");
      print_complex(add_complex(com1, com2));

      /*  Forms and displays the difference                                    */
      printf("\n\n");
      print_complex(com1);
      printf("  -  ");
      print_complex(com2);
      printf("  =  ");
      print_complex(subtract_complex(com1, com2));

      /*  Forms and displays the absolute value of the first number            */
      printf("\n\n|");
      print_complex(com1);
      printf("|  =  ");
      print_complex(abs_complex(com1));
      printf("\n");

      return (0);
}
```



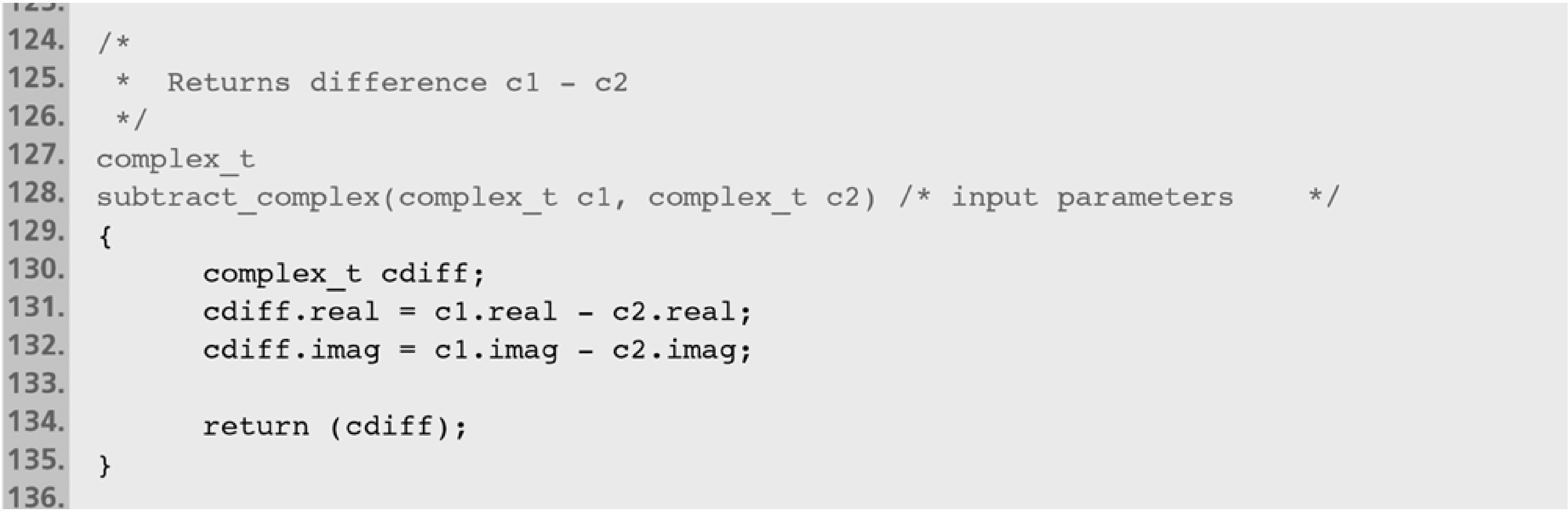
```
/*
 *  Complex number input function returns standard scanning error code
 *     1 => valid scan, 0 => error, negative EOF value => end of file
 */
int
scan_complex(complex_t *c) /* output - address of complex variable to
                                        fill                                    */
{
      int status;

      status = scanf("%lf%lf", &c->real, &c->imag);
      if (status == 2)
            status = 1;
      else if (status != EOF)
            status = 0;

      return (status);
}
```



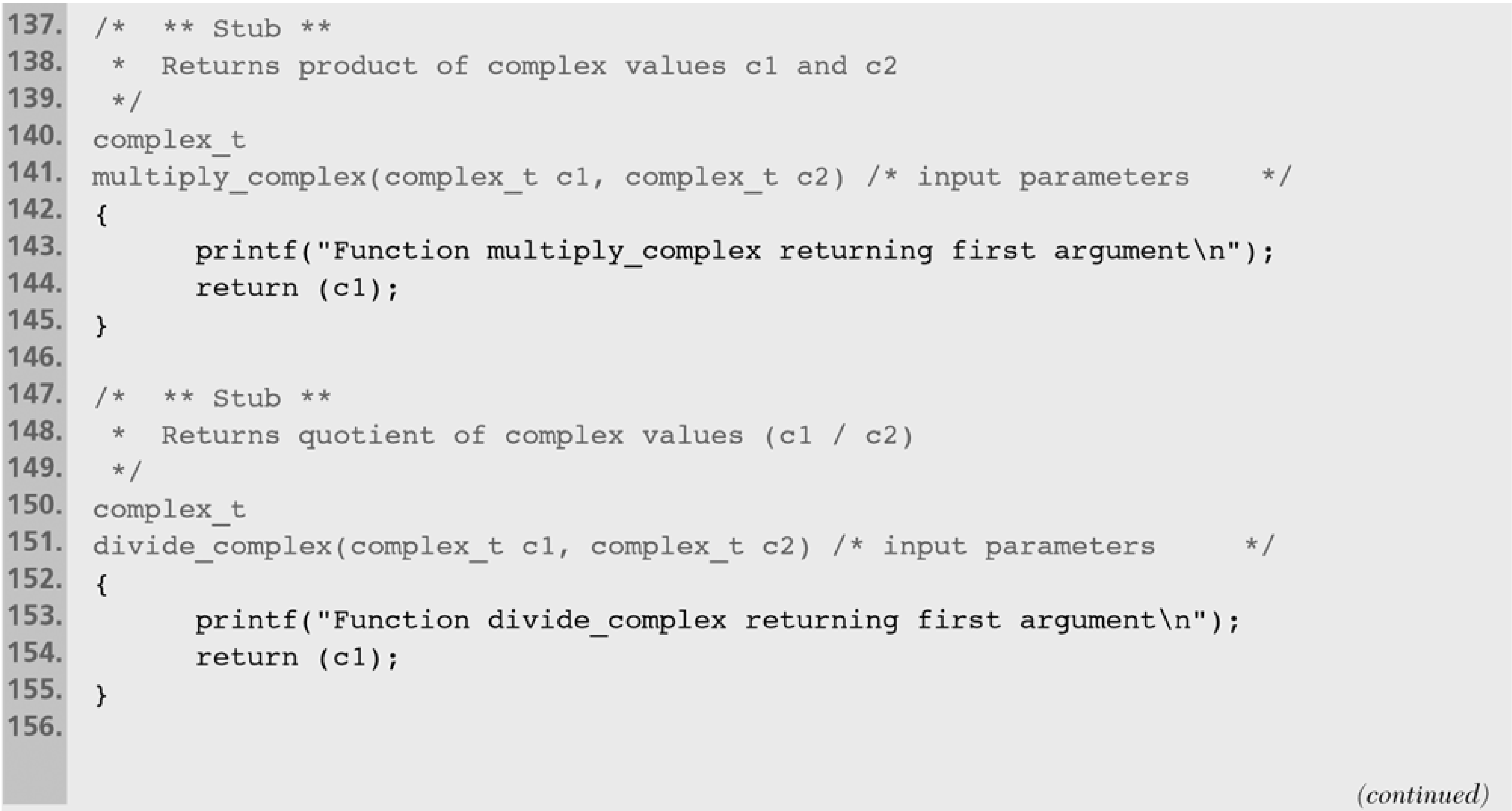
```
/*
 *  Complex output function displays value as (a + bi) or (a - bi),
 *  dropping a or b if they round to 0 unless both round to 0
 */
void
print_complex(complex_t c) /* input - complex number to display   */
{
      double a, b;
      char   sign;

      a = c.real;
      b = c.imag;

      printf("(");

      if (fabs(a) < .005  &&  fabs(b) < .005) {
            printf("%.2f", 0.0);
      } else if (fabs(b) < .005) {
            printf("%.2f", a);
      } else if (fabs(a) < .005) {
            printf("%.2fi", b);
      } else {
            if (b < 0)
                  sign = '-';
            else
                  sign = '+';
            printf("%.2f %c %.2fi", a, sign, fabs(b));
      }

```



```c
/*
 *  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);
}
```

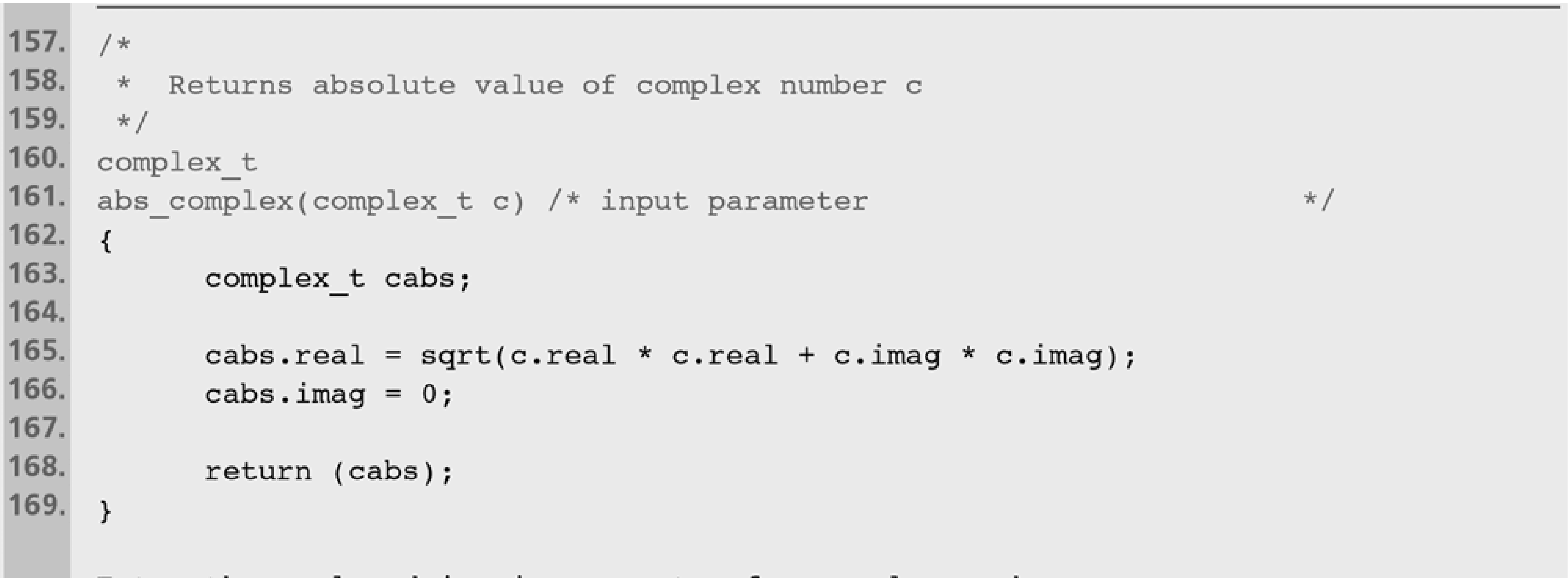


```
/*
 *  Returns difference c1 - c2
 */
complex_t
subtract_complex(complex_t c1, complex_t c2) /* input parameters    */
{
        complex_t cdiff;
        cdiff.real = c1.real - c2.real;
        cdiff.imag = c1.imag - c2.imag;

        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.

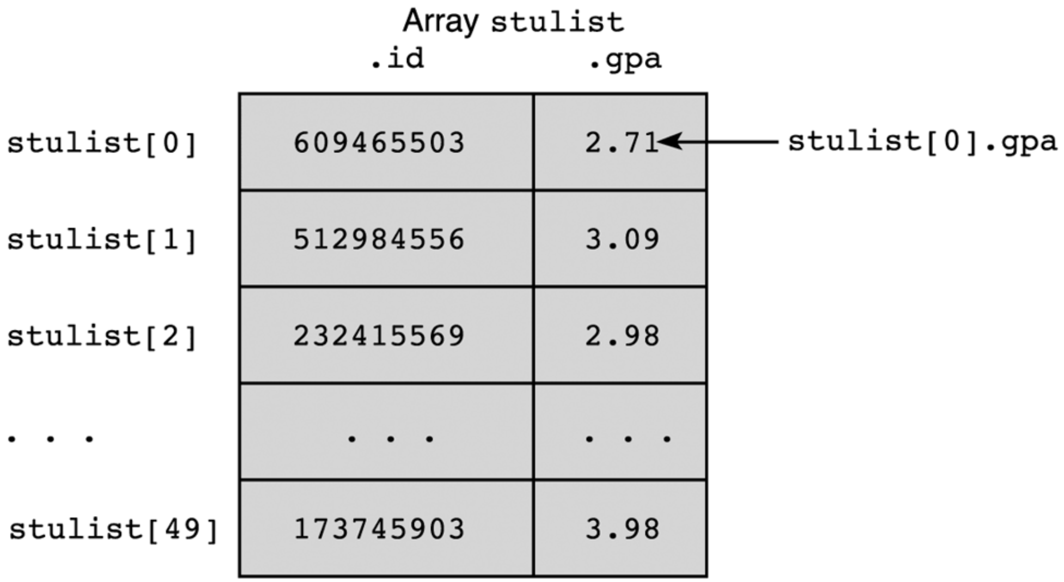
Array of Structures

- Ex. 1

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

- Ex. 2

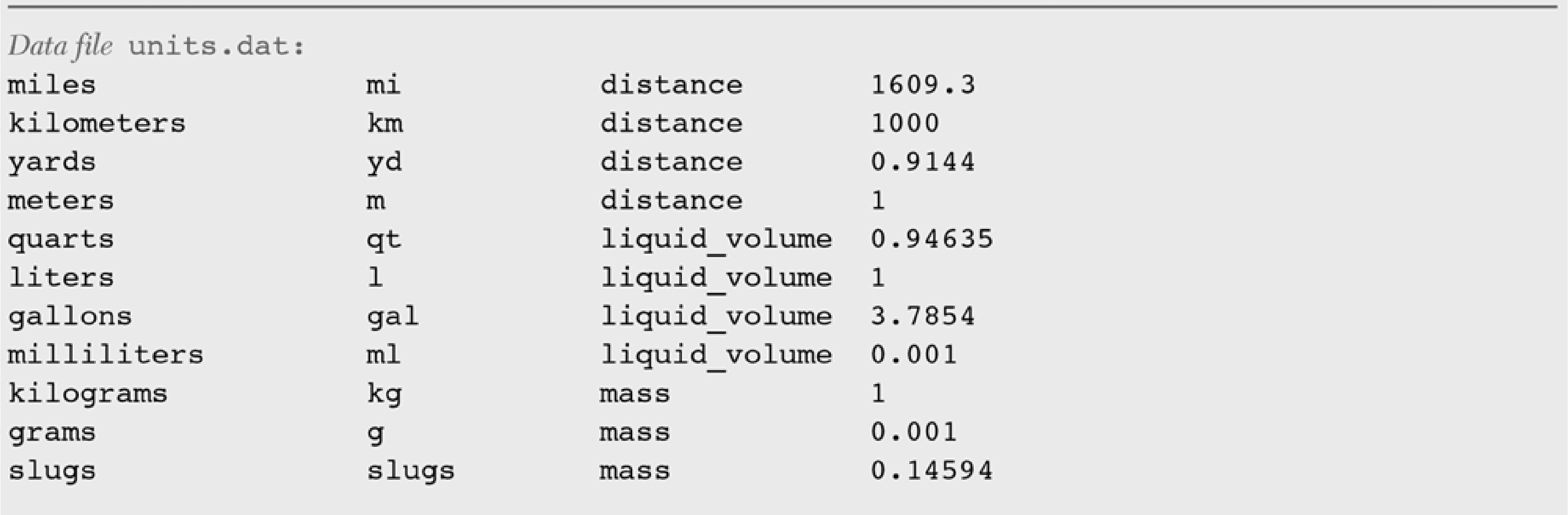
```
#define NUM_PTS 10
typedef struct {
    double x, y;
} point_t;
. . .
{
point_t polygon[NUM_PTS];
```



Array stulist

	.id	.gpa	
stulist[0]	609465503	2.71	← stulist[0].gpa
stulist[1]	512984556	3.09	
stulist[2]	232415569	2.98	
. . .	. . .	. . .	
stulist[49]	173745903	3.98	

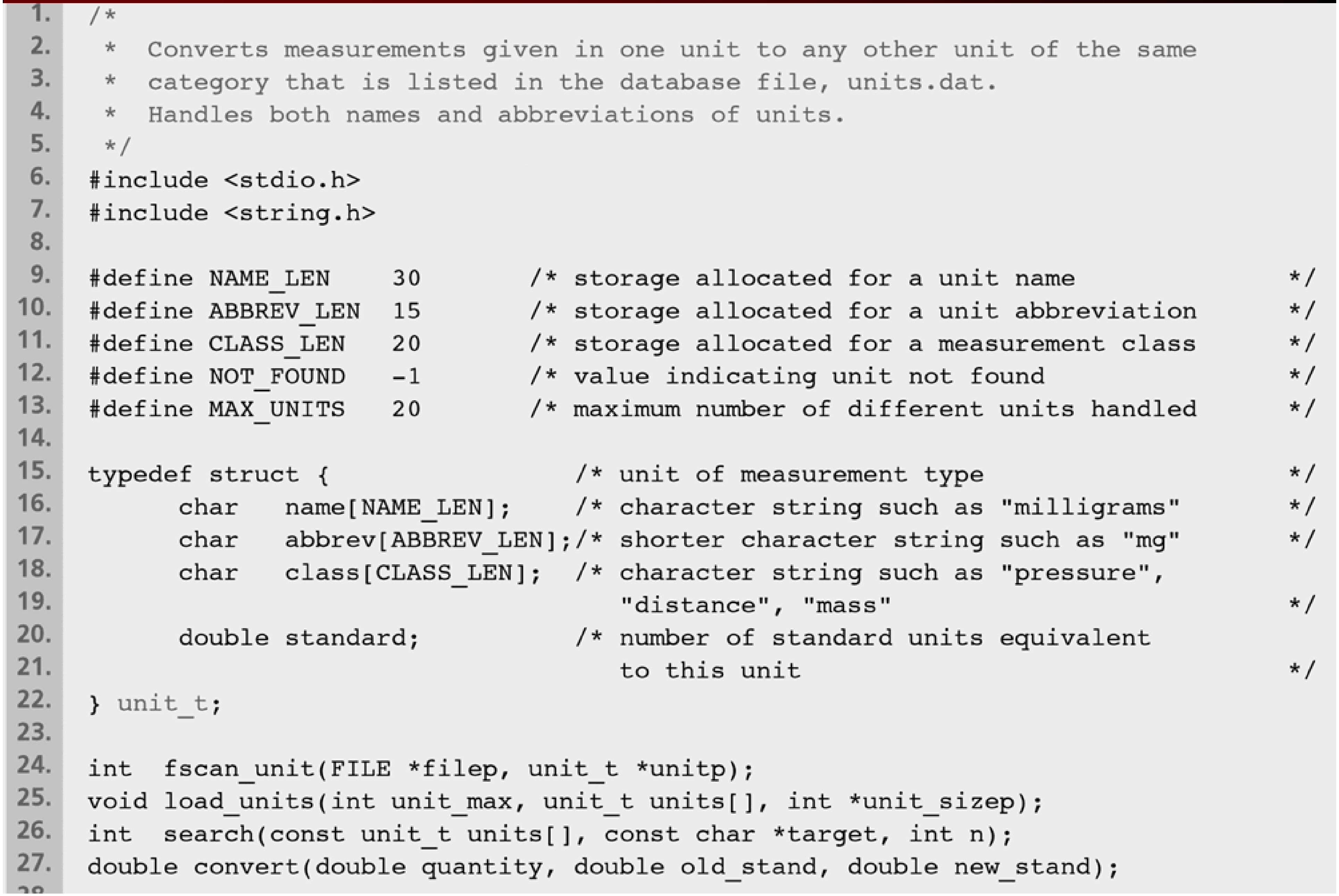
Universal Measurement Conversion



Data file units.dat:

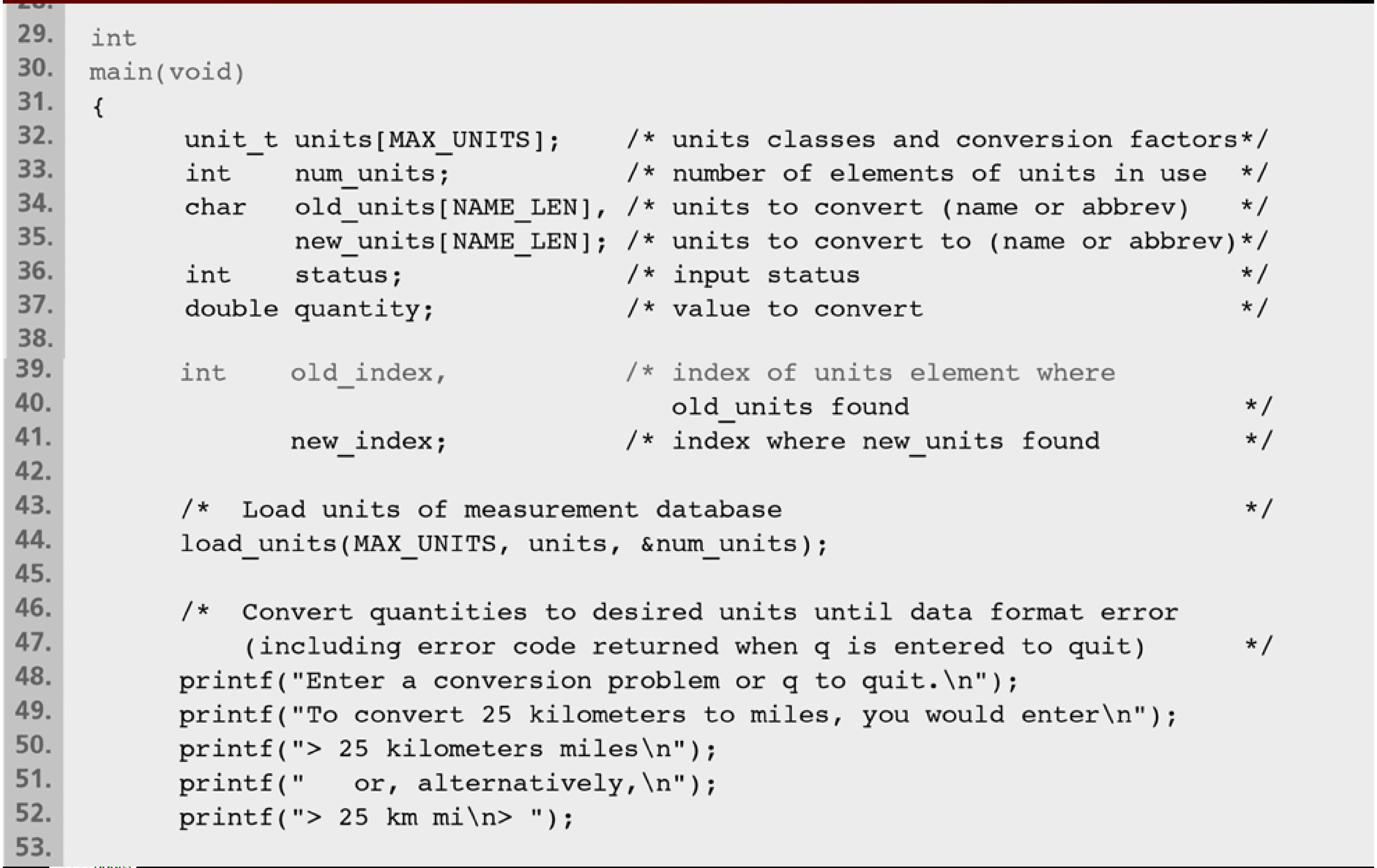
```
miles          mi       distance      1609.3
kilometers     km       distance      1000
yards          yd       distance      0.9144
meters         m        distance      1
quarts         qt       liquid_volume 0.94635
liters         l        liquid_volume 1
gallons        gal      liquid_volume 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

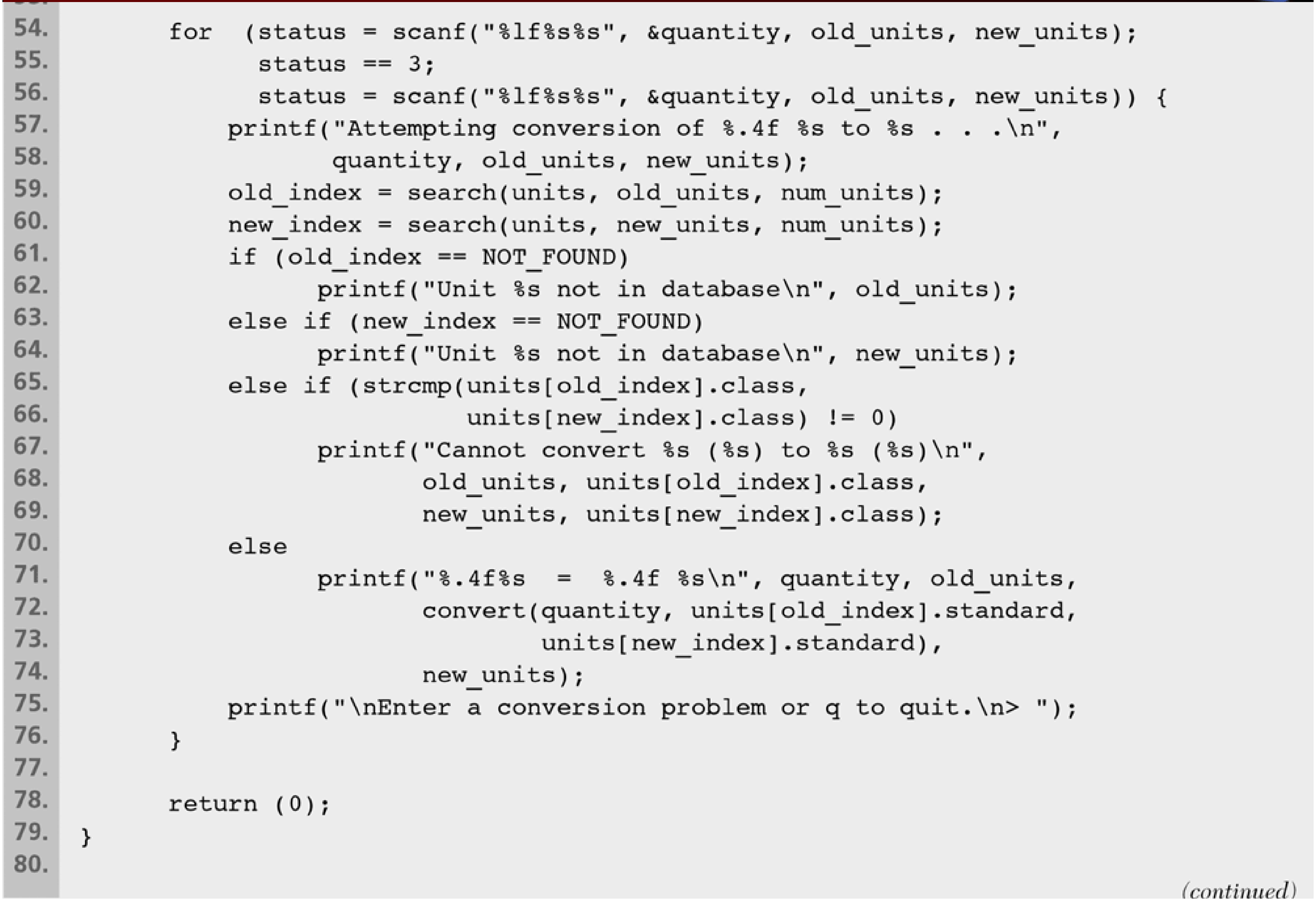


```c
/*
 *  Converts measurements given in one unit to any other unit of the same
 *  category that is listed in the database file, units.dat.
 *  Handles both names and abbreviations of units.
 */
#include <stdio.h>
#include <string.h>

#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

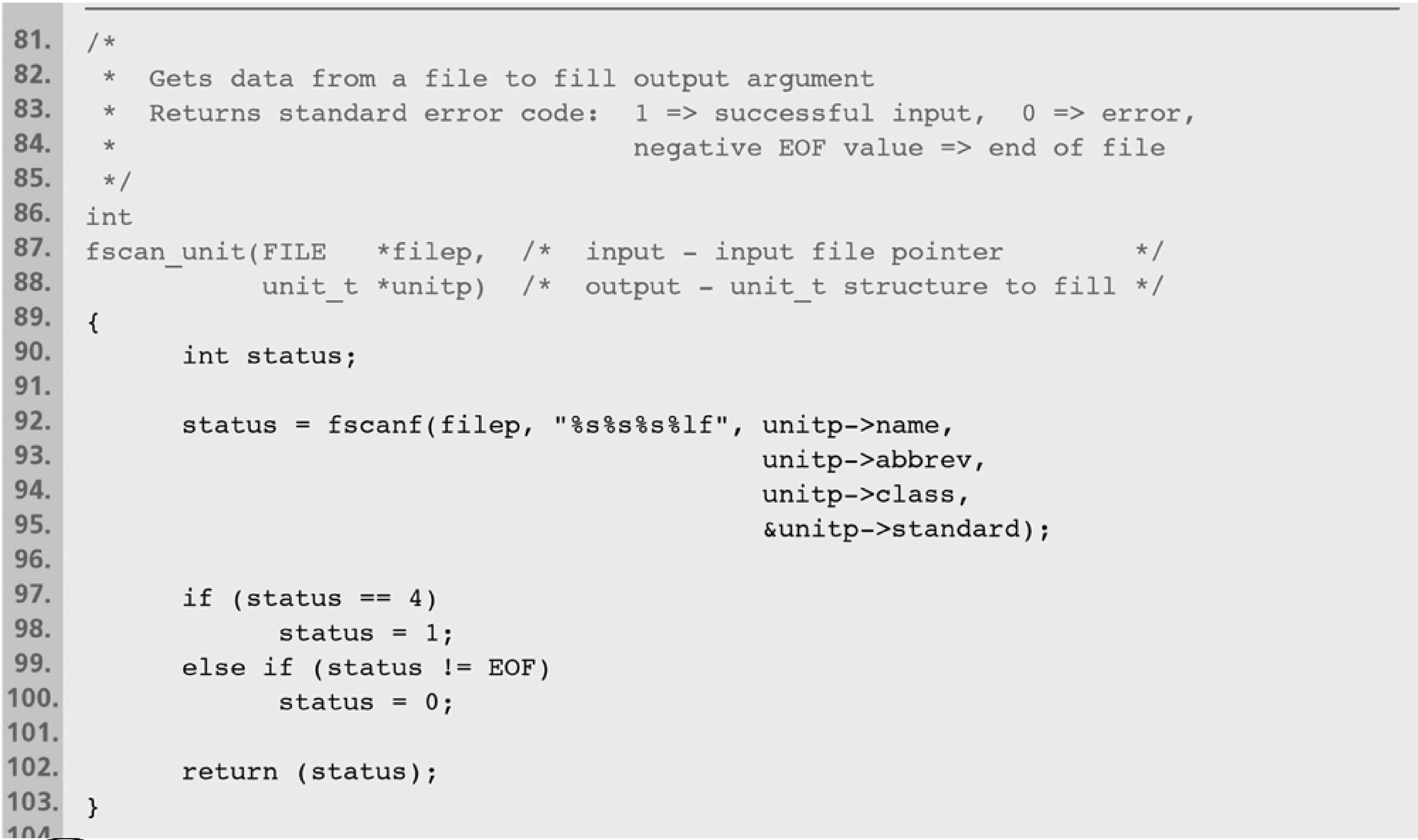


```
int
main(void)
{
      unit_t units[MAX_UNITS];    /* units classes and conversion factors*/
      int    num_units;           /* number of elements of units in use  */
      char   old_units[NAME_LEN], /* units to convert (name or abbrev)   */
             new_units[NAME_LEN]; /* units to convert to (name or abbrev)*/
      int    status;              /* input status                        */
      double quantity;            /* value to convert                    */

      int    old_index,           /* index of units element where
                                     old_units found                     */
             new_index;           /* index where new_units found         */

      /*  Load units of measurement database                             */
      load_units(MAX_UNITS, units, &num_units);

      /*  Convert quantities to desired units until data format error
          (including error code returned when q is entered to quit)      */
      printf("Enter a conversion problem or q to quit.\n");
      printf("To convert 25 kilometers to miles, you would enter\n");
      printf("> 25 kilometers miles\n");
      printf("   or, alternatively,\n");
      printf("> 25 km mi\n> ");
```

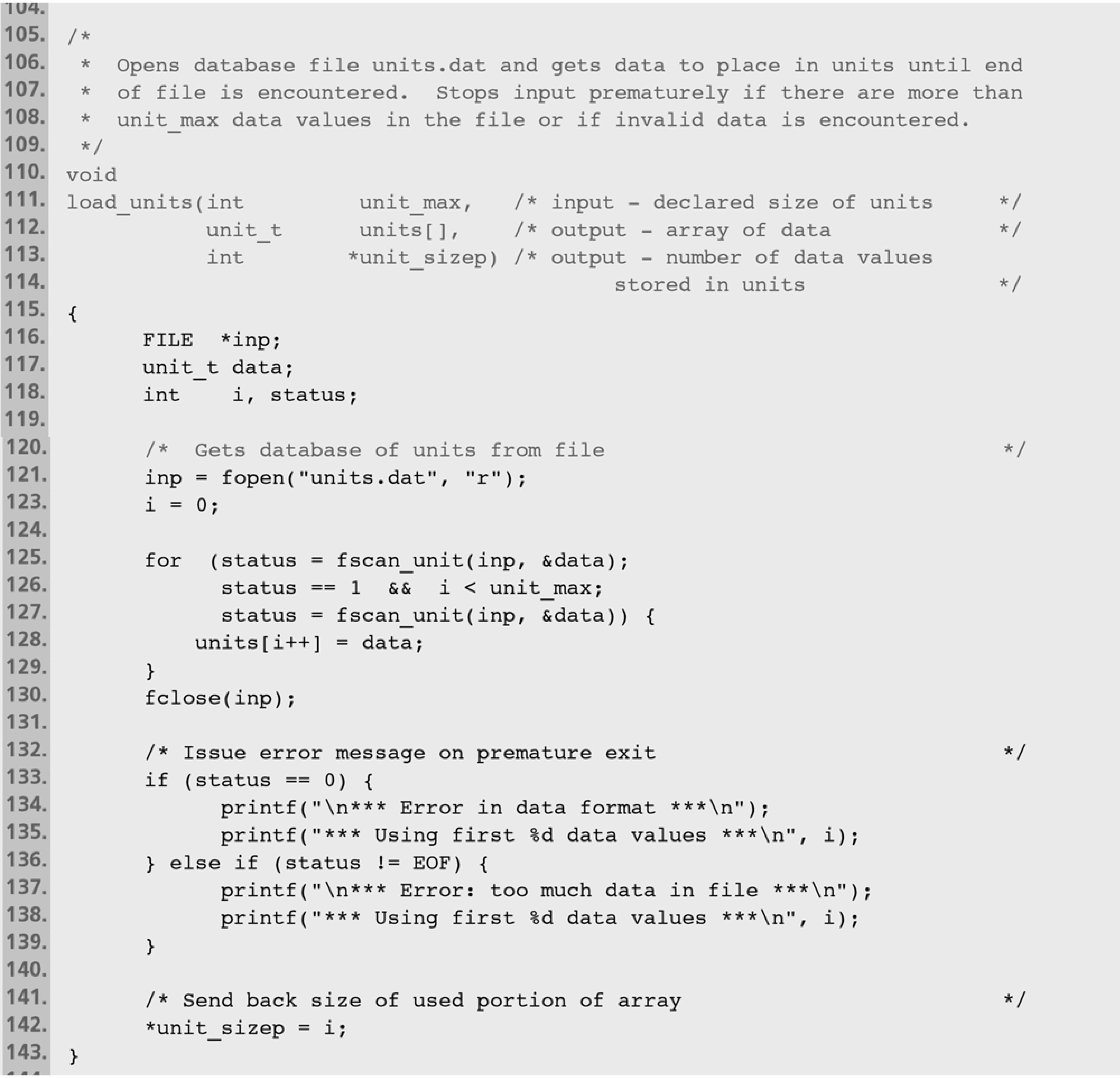


```
      for  (status = scanf("%lf%s%s", &quantity, old_units, new_units);
            status == 3;
            status = scanf("%lf%s%s", &quantity, old_units, new_units)) {
          printf("Attempting conversion of %.4f %s to %s . . .\n",
                 quantity, old_units, new_units);
          old_index = search(units, old_units, num_units);
          new_index = search(units, new_units, num_units);
          if (old_index == NOT_FOUND)
                printf("Unit %s not in database\n", old_units);
          else if (new_index == NOT_FOUND)
                printf("Unit %s not in database\n", new_units);
          else if (strcmp(units[old_index].class,
                          units[new_index].class) != 0)
                printf("Cannot convert %s (%s) to %s (%s)\n",
                       old_units, units[old_index].class,
                       new_units, units[new_index].class);
          else
                printf("%.4f%s  =  %.4f %s\n", quantity, old_units,
                       convert(quantity, units[old_index].standard,
                               units[new_index].standard),
                       new_units);
          printf("\nEnter a conversion problem or q to quit.\n> ");
      }

      return (0);
}
```

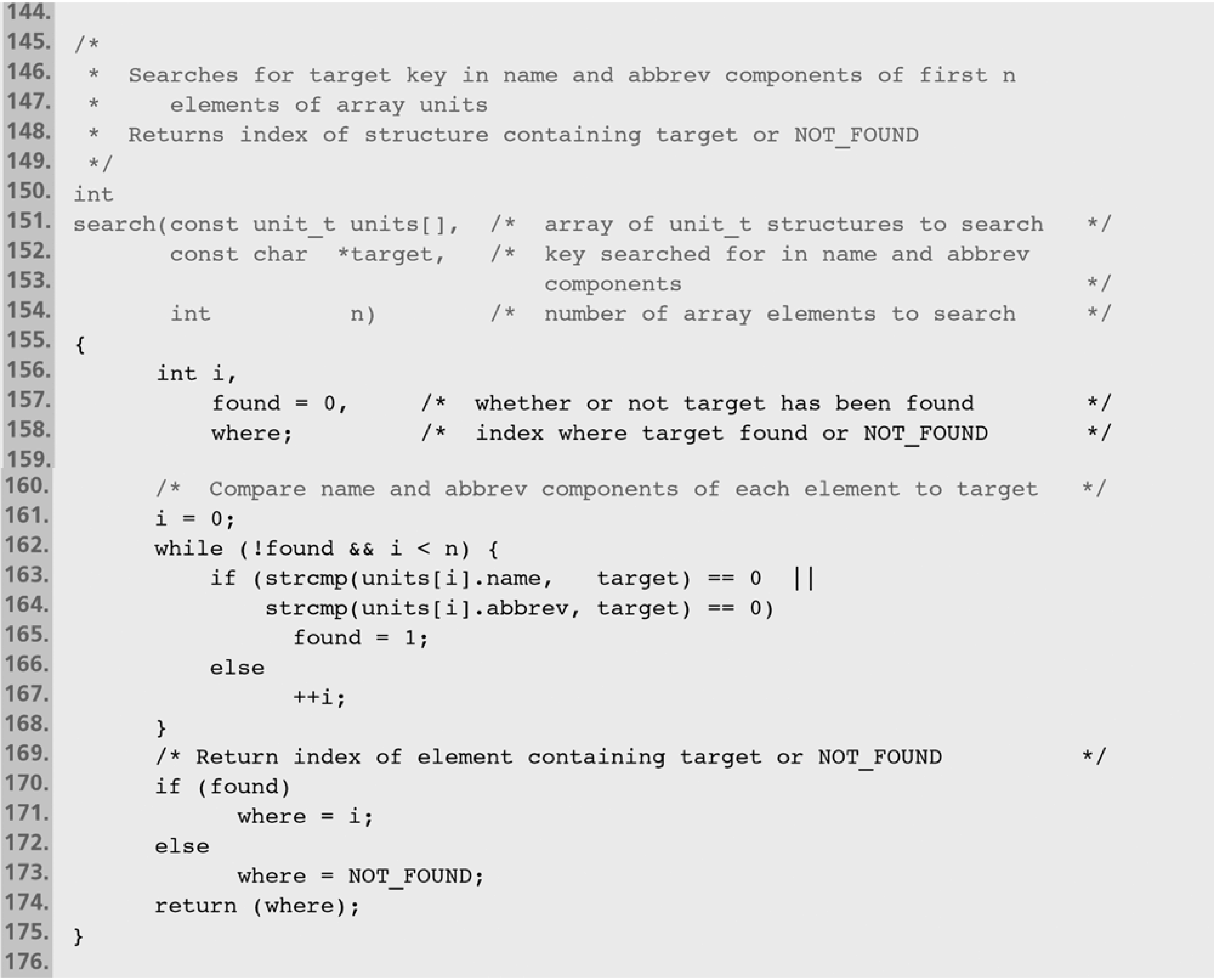
(continued)



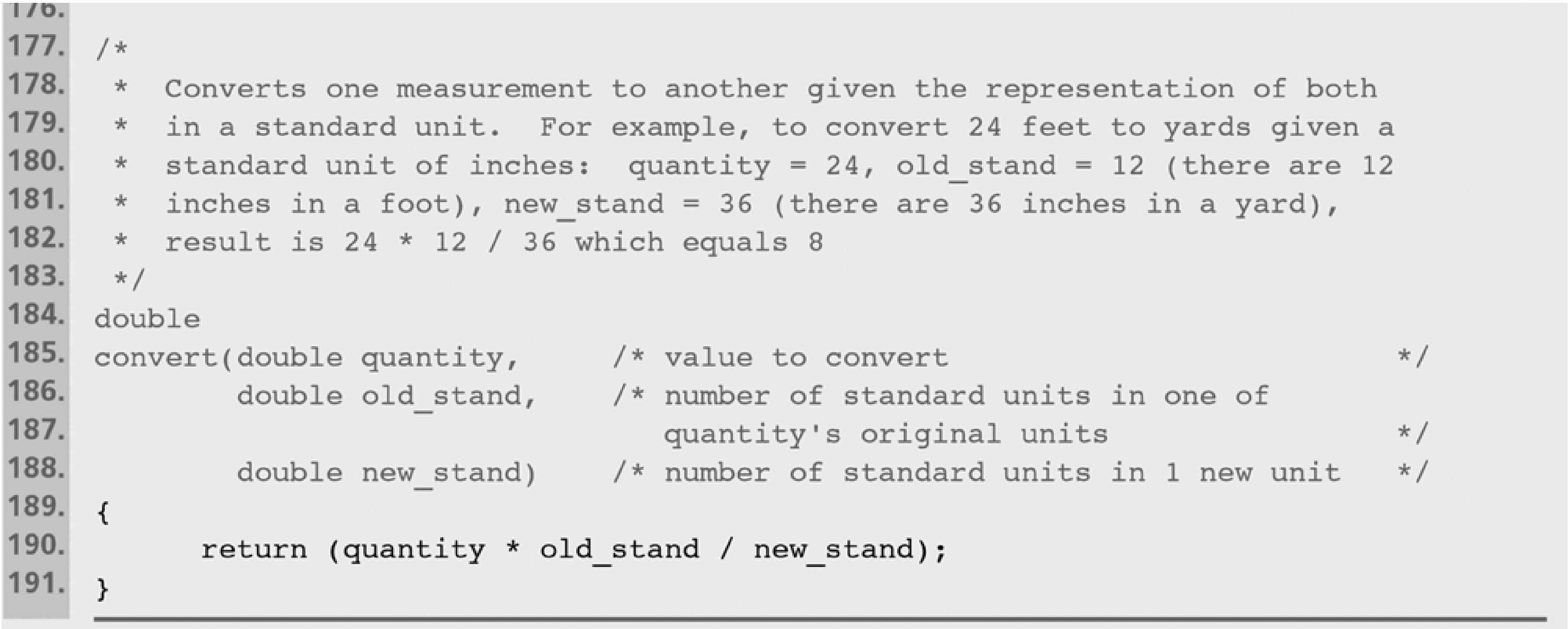
```c
/*
 *  Gets data from a file to fill output argument
 *  Returns standard error code:  1 => successful input,  0 => error,
 *                                negative EOF value => end of file
 */
int
fscan_unit(FILE   *filep,  /*  input - input file pointer         */
           unit_t *unitp)  /*  output - unit_t structure to fill */
{
      int status;

      status = fscanf(filep, "%s%s%s%lf", unitp->name,
                                          unitp->abbrev,
                                          unitp->class,
                                          &unitp->standard);

      if (status == 4)
            status = 1;
      else if (status != EOF)
            status = 0;

      return (status);
}
```



```
/*
 *  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;
}
```



```c
/*
 *  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);
}
```



```c
/*
 *  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?

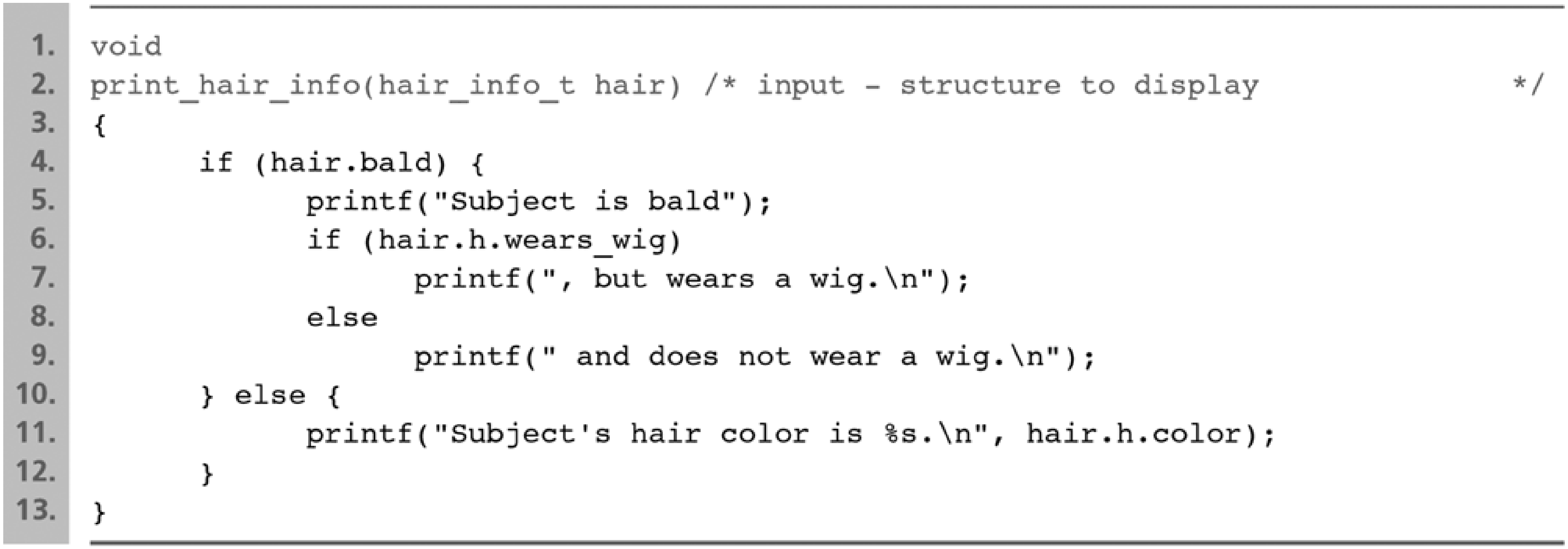
Union Types

- 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;
```

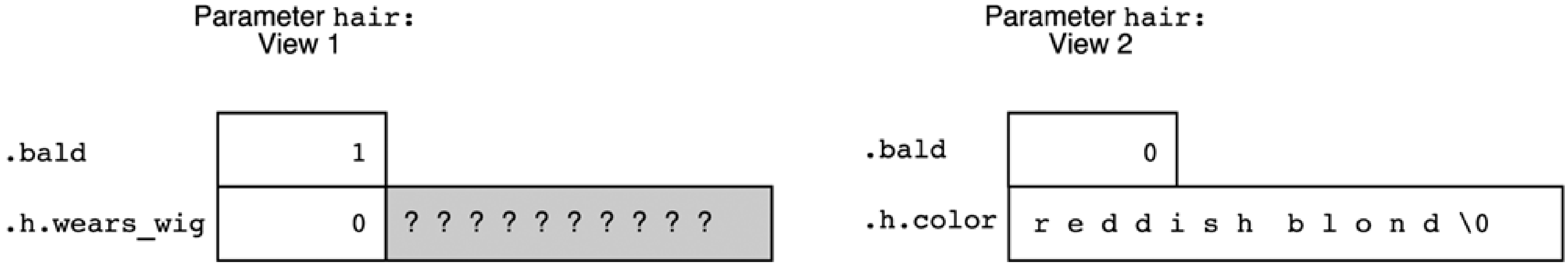
- Referencing the appropriate union component is *always* the programmer’s responsibility; C can do no checking of the validity of such a component reference.

Displays a Structure with a Union



```
void
print_hair_info(hair_info_t hair) /* input - structure to display              */
{
      if (hair.bald) {
            printf("Subject is bald");
            if (hair.h.wears_wig)
                  printf(", but wears a wig.\n");
            else
                  printf(" and does not wear a wig.\n");
      } else {
            printf("Subject's hair color is %s.\n", hair.h.color);
      }
}
```

Two Interpretations of Parameter hair



Parameter hair:
View 1

.bald 1

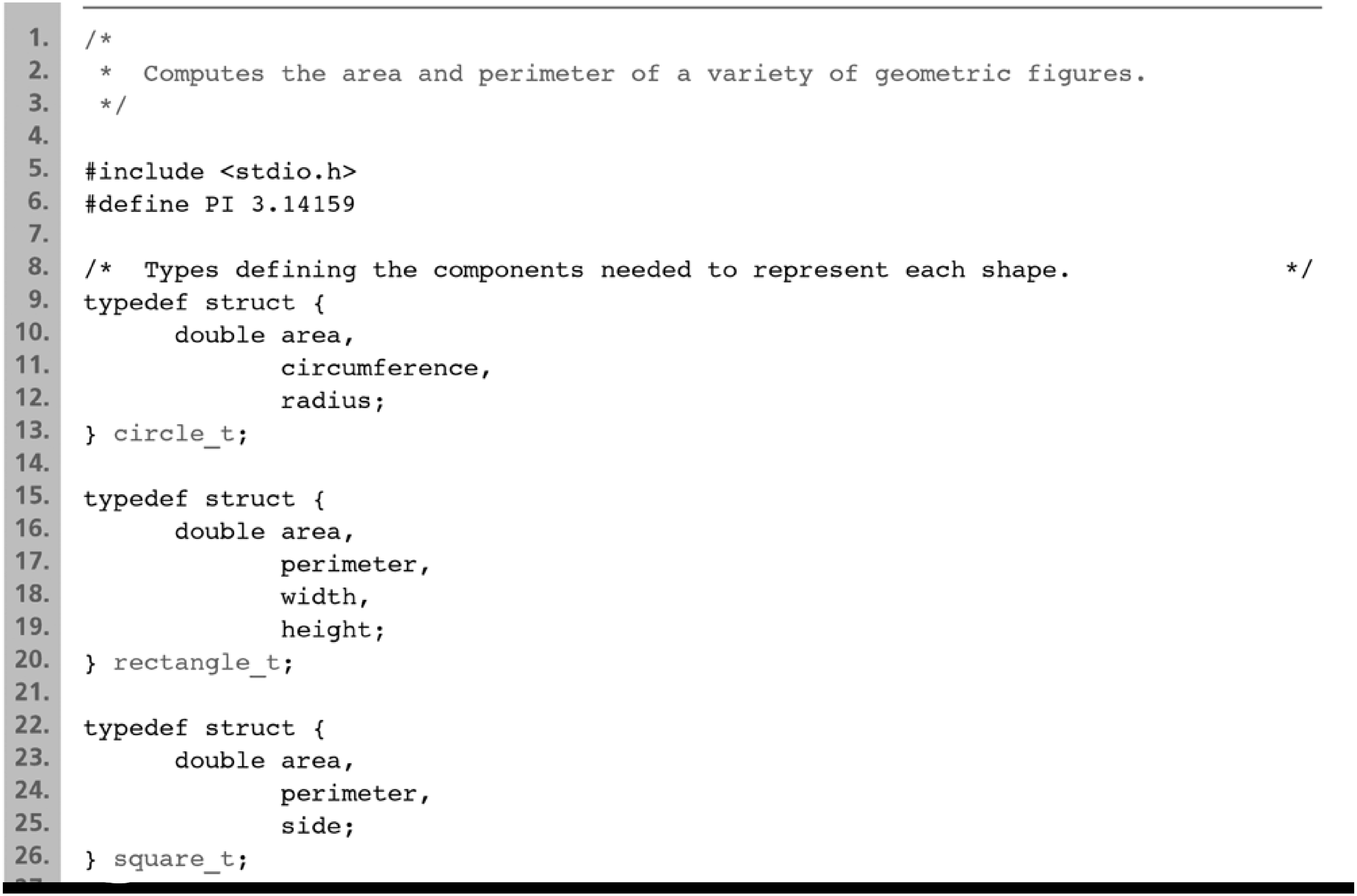
.h.wears_wig 0 ? ? ? ? ? ? ? ? ? ?

Parameter hair:
View 2

.bald 0

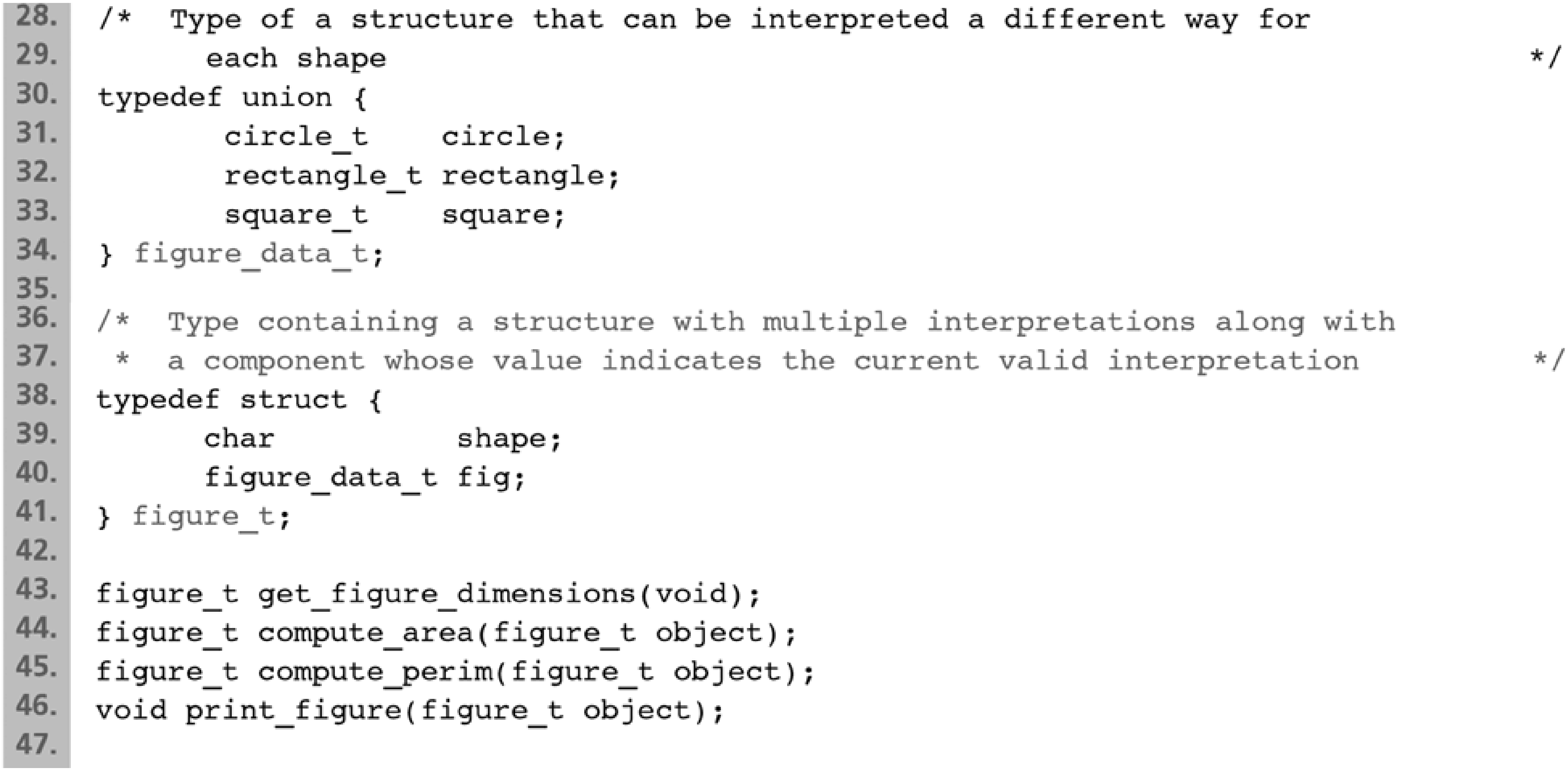
.h.color r e d d i s h b l o n d \0

Compute Area and Perimeter

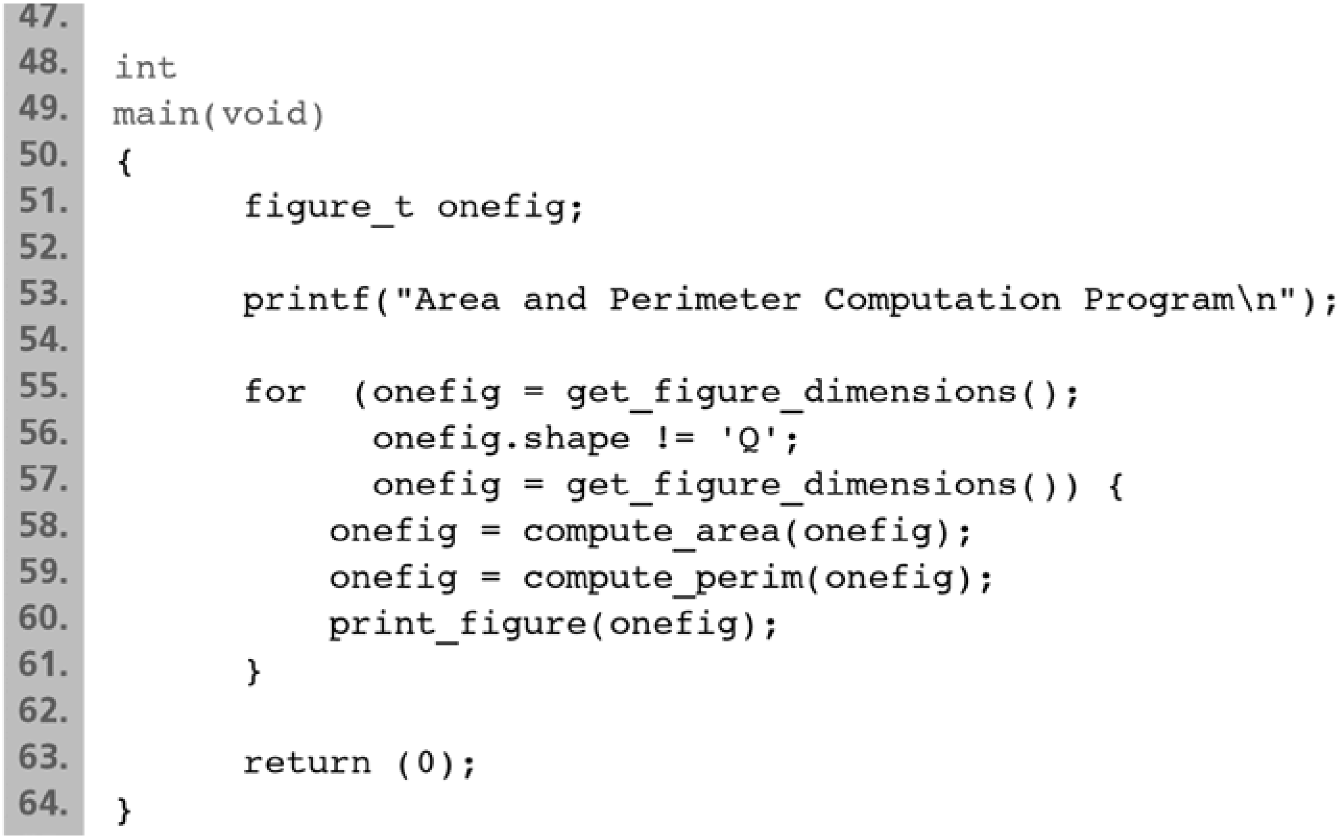


```c
/*
 *  Computes the area and perimeter of a variety of geometric figures.
 */

#include <stdio.h>
#define PI 3.14159

/*  Types defining the components needed to represent each shape.              */
typedef struct {
      double area,
             circumference,
             radius;
} circle_t;

typedef struct {
      double area,
             perimeter,
             width,
             height;
} rectangle_t;

typedef struct {
      double area,
             perimeter,
             side;
} square_t;
```

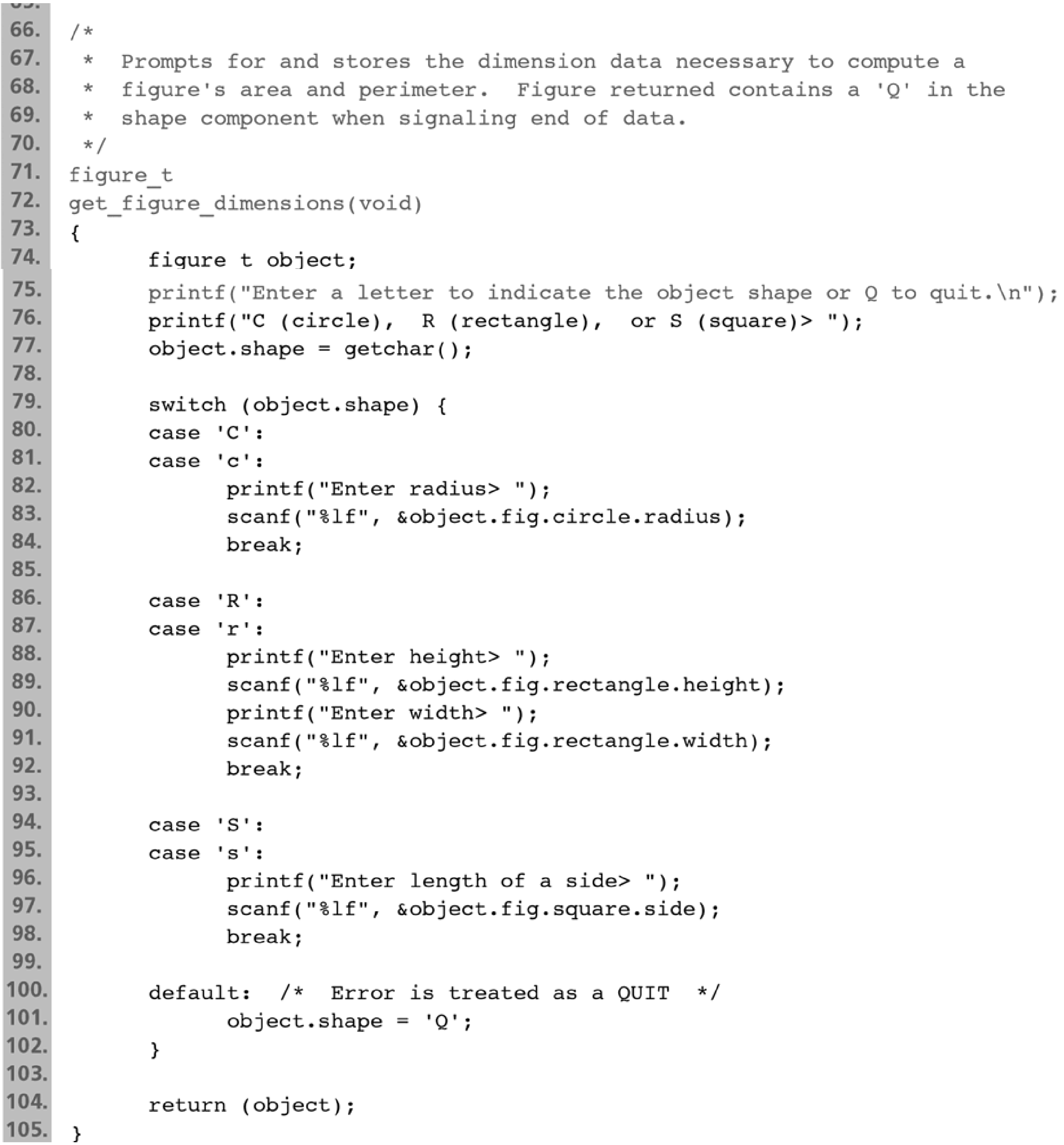
Compute Area and Perimeter



```
/*  Type of a structure that can be interpreted a different way for
       each shape                                                          */
typedef union {
        circle_t    circle;
        rectangle_t rectangle;
        square_t    square;
} figure_data_t;

/*  Type containing a structure with multiple interpretations along with
 *  a component whose value indicates the current valid interpretation     */
typedef struct {
      char          shape;
      figure_data_t fig;
} figure_t;

figure_t get_figure_dimensions(void);
figure_t compute_area(figure_t object);
figure_t compute_perim(figure_t object);
void print_figure(figure_t object);
```



```c

int
main(void)
{
        figure_t onefig;

        printf("Area and Perimeter Computation Program\n");

        for  (onefig = get_figure_dimensions();
              onefig.shape != 'Q';
              onefig = get_figure_dimensions()) {
            onefig = compute_area(onefig);
            onefig = compute_perim(onefig);
            print_figure(onefig);
        }

        return (0);
}
```



```c
/*
 *  Prompts for and stores the dimension data necessary to compute a
 *  figure's area and perimeter.  Figure returned contains a 'Q' in the
 *  shape component when signaling end of data.
 */
figure_t
get_figure_dimensions(void)
{
      figure t object;
      printf("Enter a letter to indicate the object shape or Q to quit.\n");
      printf("C (circle),  R (rectangle),  or S (square)> ");
      object.shape = getchar();

      switch (object.shape) {
      case 'C':
      case 'c':
            printf("Enter radius> ");
            scanf("%lf", &object.fig.circle.radius);
            break;

      case 'R':
      case 'r':
            printf("Enter height> ");
            scanf("%lf", &object.fig.rectangle.height);
            printf("Enter width> ");
            scanf("%lf", &object.fig.rectangle.width);
            break;

      case 'S':
      case 's':
            printf("Enter length of a side> ");
            scanf("%lf", &object.fig.square.side);
            break;

      default:  /*  Error is treated as a QUIT  */
            object.shape = 'Q';
      }

      return (object);
}
```

```c
/*
 *  Computes the area of a figure given relevant dimensions.  Returns
 *  figure with area component filled.
 *  Pre:  value of shape component is one of these letters: CcRrSs
 *        necessary dimension components have values
 */
figure_t
compute_area(figure_t object)
{
        switch (object.shape) {
        case 'C':
        case 'c':
                object.fig.circle.area = PI * object.fig.circle.radius *
                                         object.fig.circle.radius;
                break;

        case 'R':
        case 'r':
                object.fig.rectangle.area = object.fig.rectangle.height *
                                            object.fig.rectangle.width;
                break;

        case 'S':
        case 's':
                object.fig.square.area = object.fig.square.side *
                                         object.fig.square.side;
                break;

        default:
                printf("Error in shape code detected in compute_area\n");
        }

        return (object);
}
```