CMPE 252
C PROGRAMMING

SPRING 2021

WEEK 8-9

ENUM, STRUCTURE AND UNION TYPES

CHAPTER 10

Problem Solving & Program Design in C

Eighth Edition
Global Edition

Jeri R. Hanly & Elliot B. Koffman

Chapter Objectives

- To learn how to declare and use your own data types, enum
- To learn how to declare a struct data type which consists of several data fields, each with its own name and data type
- To understand how to use a struct to store data for a structured object or record
- To learn how to use dot notation to process individual fields of a structured object
- To learn how to use structs as function parameters and to return function results

Chapter Objectives

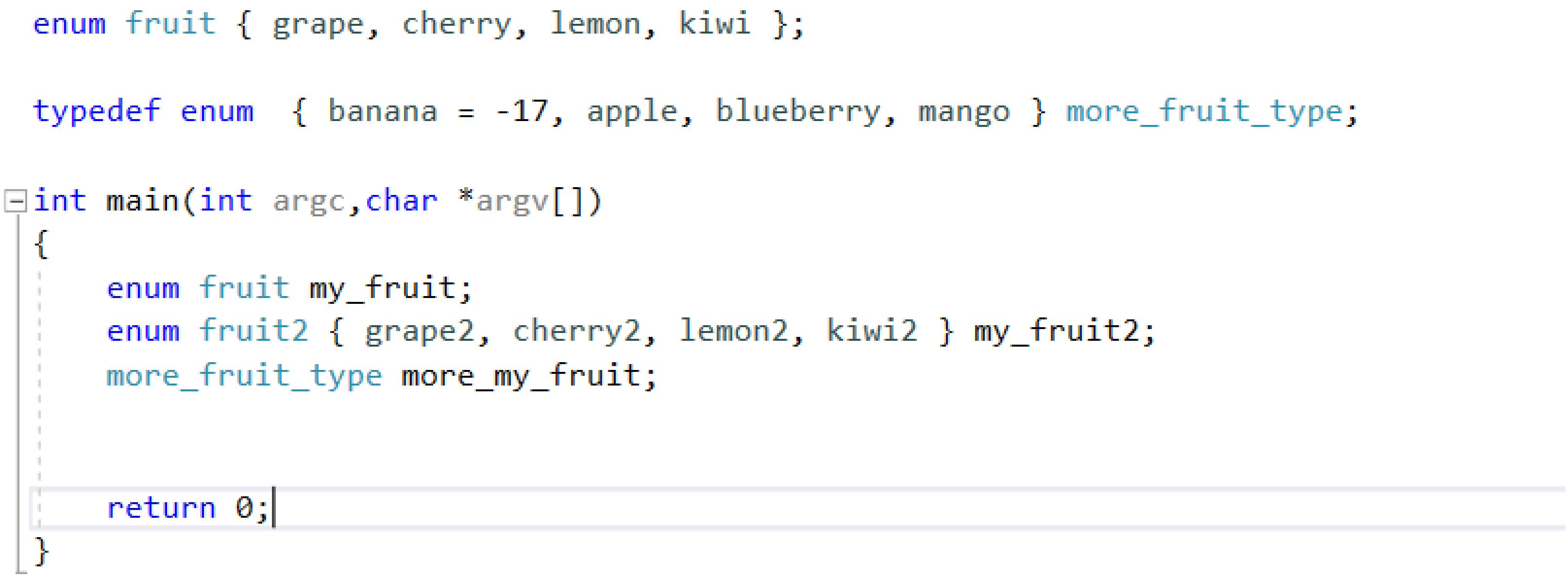
- To see how to create a struct data type for representing complex numbers and how to write functions that perform arithmetic operations on complex numbers
- To understand the relationship between parallel arrays and arrays of structured objects
- To learn about union data types and how they differ form structs

Enumerated Types

- enumerated type
  - a data type whose list of values is specified by the programmer in a type declaration
  - Special form of integers
- enumeration constant
  - an identifier that is one of the values of an enumerated type
  - Monday: integer 0, Tuesday: integer 1, so on..

```
typedef enum
        { Monday, Tuesday, Wednesday, Thursday,
          Friday, Saturday, Sunday } day_t;
```

Alternative ways



```
enum fruit { grape, cherry, lemon, kiwi };

typedef enum  { banana = -17, apple, blueberry, mango } more_fruit_type;

int main(int argc,char *argv[])
{
    enum fruit my_fruit;
    enum fruit2 { grape2, cherry2, lemon2, kiwi2 } my_fruit2;
    more_fruit_type more_my_fruit;


    return 0;
}
```

Typedef Basics

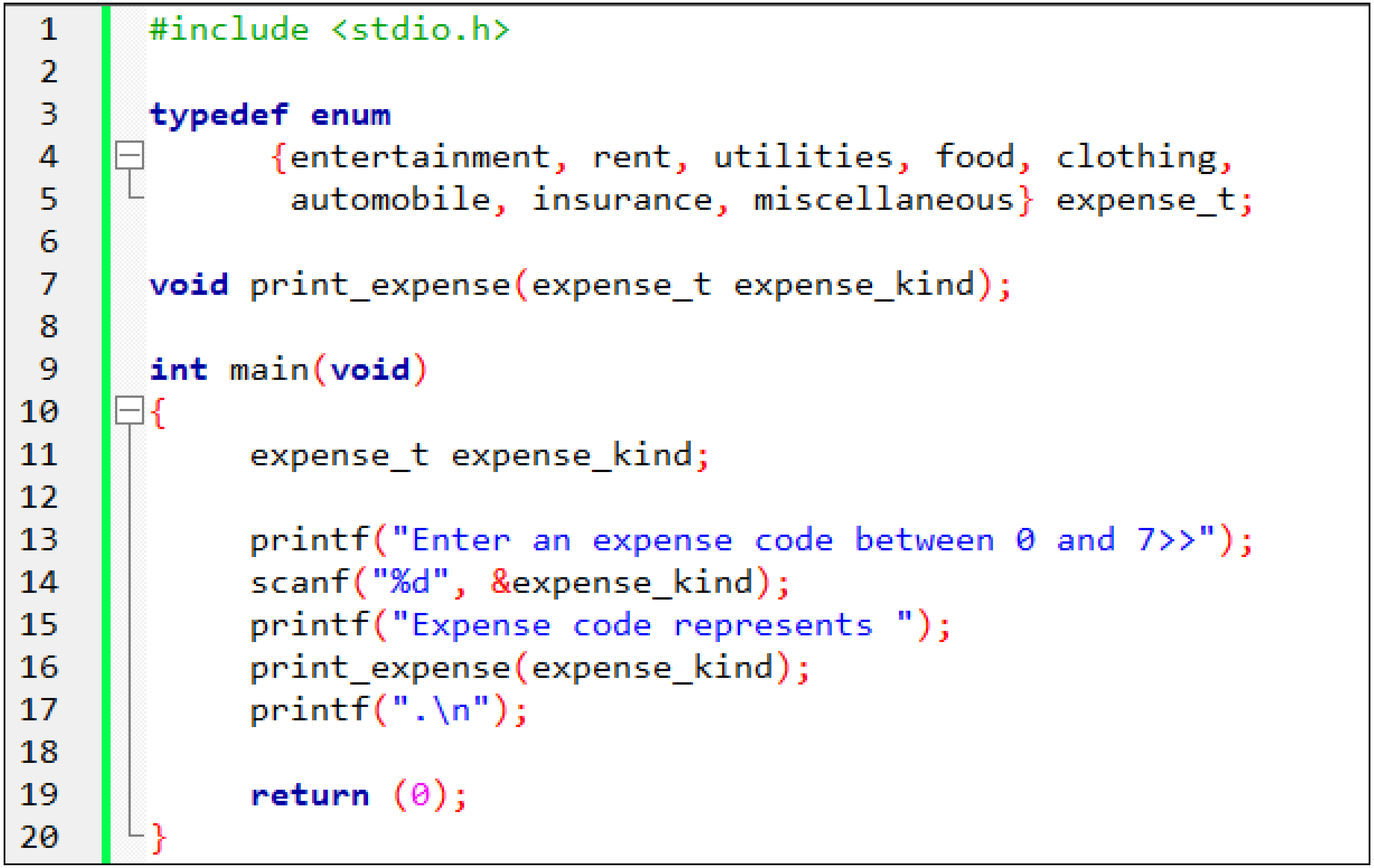
- The C programming language provides a keyword called **typedef**, which you can use to give a type a new name.

- typedef unsigned char BYTE;

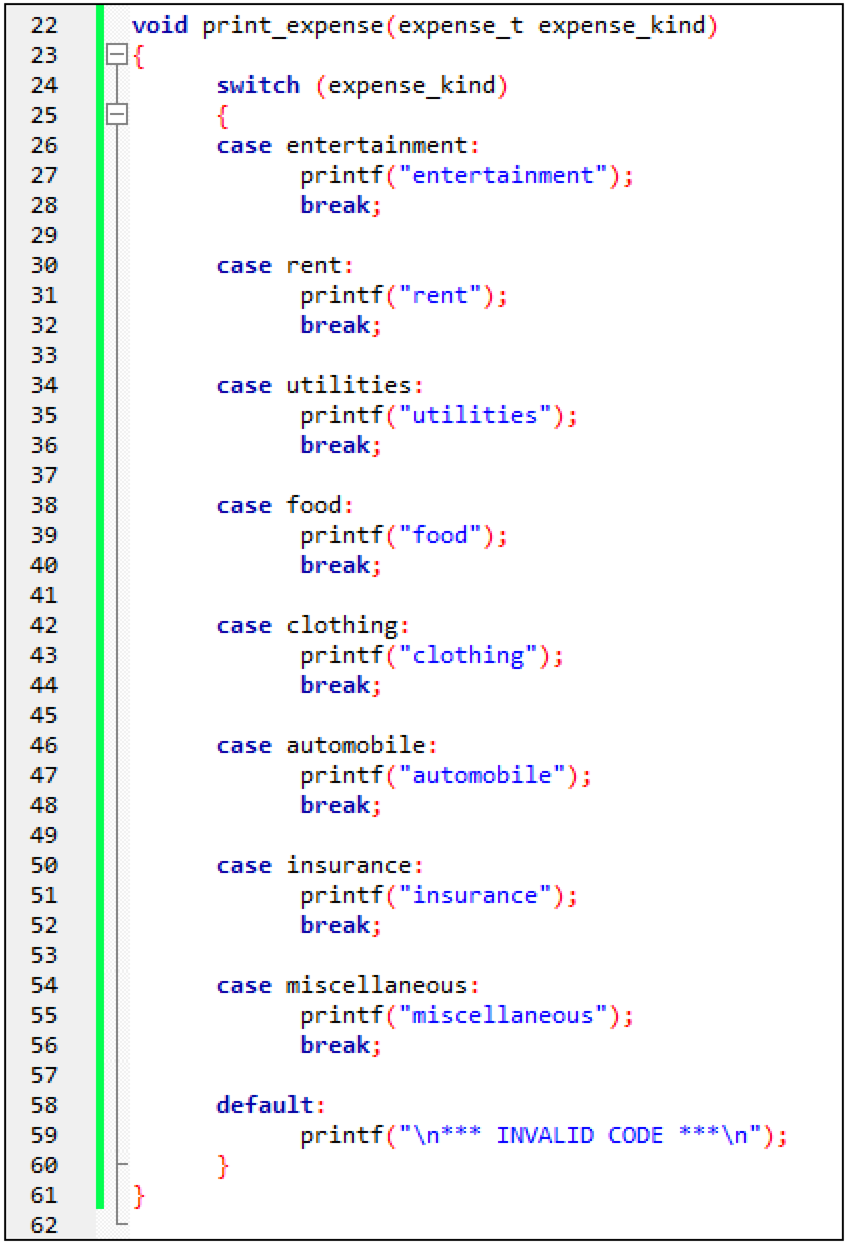
- After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char, for example.**.
  - BYTE b1, b2;

typedef vs. #define

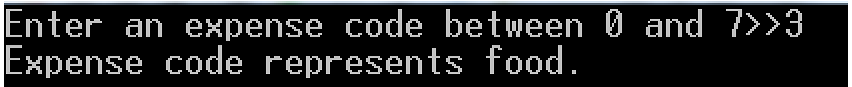
- **#define** is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences:
  - **typedef** is limited to giving symbolic names to types only whereas **#define** can be used to define alias for values as well, you can define 1 as ONE etc.
  - **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the pre-processor.



```c
#include <stdio.h>

typedef enum
      {entertainment, rent, utilities, food, clothing,
       automobile, insurance, miscellaneous} expense_t;

void print_expense(expense_t expense_kind);

int main(void)
{
     expense_t expense_kind;

     printf("Enter an expense code between 0 and 7>>");
     scanf("%d", &expense_kind);
     printf("Expense code represents ");
     print_expense(expense_kind);
     printf(".\n");

     return (0);
}
```



```c
void print_expense(expense_t expense_kind)
{
        switch (expense_kind)
        {
        case entertainment:
                printf("entertainment");
                break;

        case rent:
                printf("rent");
                break;

        case utilities:
                printf("utilities");
                break;

        case food:
                printf("food");
                break;

        case clothing:
                printf("clothing");
                break;

        case automobile:
                printf("automobile");
                break;

        case insurance:
                printf("insurance");
                break;

        case miscellaneous:
                printf("miscellaneous");
                break;

        default:
                printf("\n*** INVALID CODE ***\n");
        }
}
```



```
Enter an expense code between 0 and 7>>3
Expense code represents food.
```

Enum Arithmetic

```
typedef enum
            { Monday, Tuesday, Wednesday, Thursday,
           Friday, Saturday, Sunday } day_t;
```

- Sunday < Monday
- Wednesday != Friday
- Tuesday >= Sunday

Enumerations are actually constant integer values, by default starts from 0 and increments by one.

Enum Arithmetic

Enumerations are actually constant integer values, by default starts from 0 and increments by 1.

You can define the starting enumeration value:

```
enum more_fruit {banana = -17, apple, blueberry, mango};
```

This defines banana to be -17, and the remaining values are incremented by 1: apple is -16, blueberry is -15, and mango is -14.

Unless specified otherwise, an enumeration value is equal to one more than the previous value (and the first value defaults to 0).

```
enum more_fruit {banana, apple = 20, blueberry, mango};

enum yet_more_fruit {kumquat, raspberry, peach,   plum = peach + 2};
```

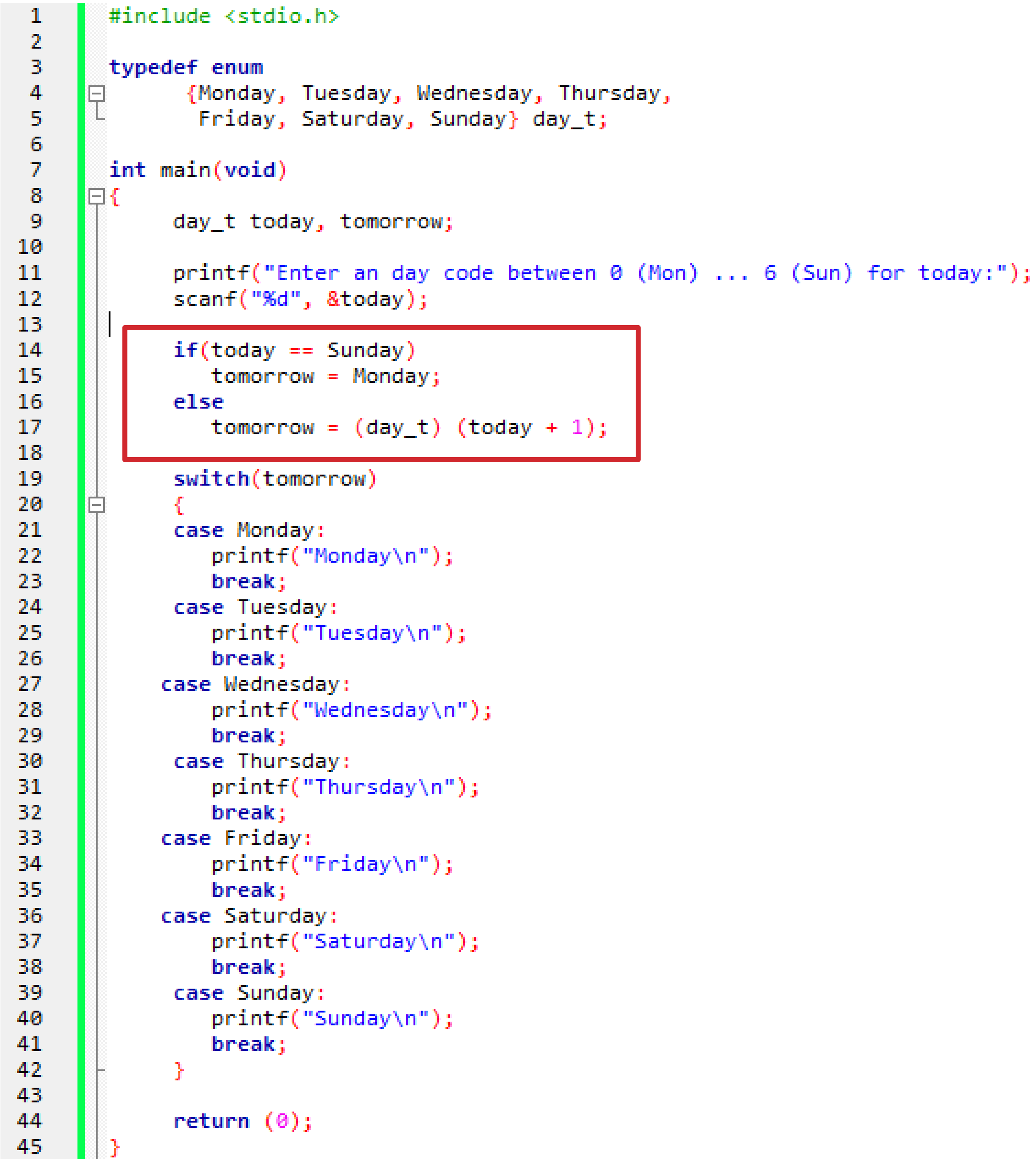
Enum Arithmetic

- enum fruit {banana, apple, blueberry, mango};
- enum fruit my_fruit;

- Enum variables are actually integers, so you can assign integer values to enum variables, including values from other enumerations.
- Furthermore, any variable that can be assigned an int value can be assigned a value from an enumeration.

- However, you cannot change the values in an enumeration once it has been defined; they are constant values. For example, this won’t work:

- enum fruit {banana, apple, blueberry, mango};
- banana = 15; /* You can’t do this! */

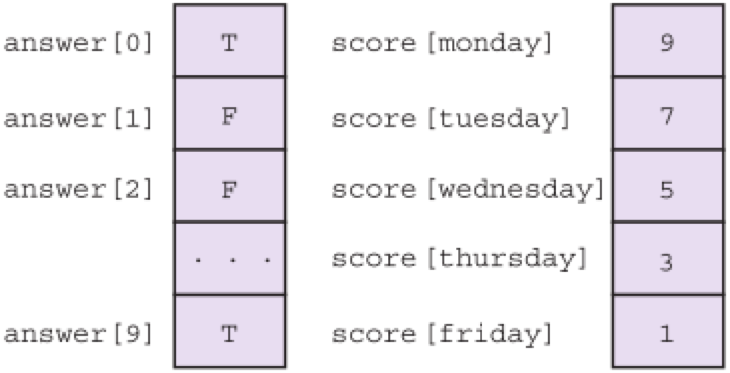


```
#include <stdio.h>

typedef enum
      {Monday, Tuesday, Wednesday, Thursday,
       Friday, Saturday, Sunday} day_t;

int main(void)
{
     day_t today, tomorrow;

     printf("Enter an day code between 0 (Mon) ... 6 (Sun) for today:");
     scanf("%d", &today);

     if(today == Sunday)
        tomorrow = Monday;
     else
        tomorrow = (day_t) (today + 1);

     switch(tomorrow)
     {
     case Monday:
        printf("Monday\n");
        break;
     case Tuesday:
        printf("Tuesday\n");
        break;
    case Wednesday:
        printf("Wednesday\n");
        break;
     case Thursday:
        printf("Thursday\n");
        break;
    case Friday:
        printf("Friday\n");
        break;
    case Saturday:
        printf("Saturday\n");
        break;
     case Sunday:
        printf("Sunday\n");
        break;
     }

     return (0);
}
```

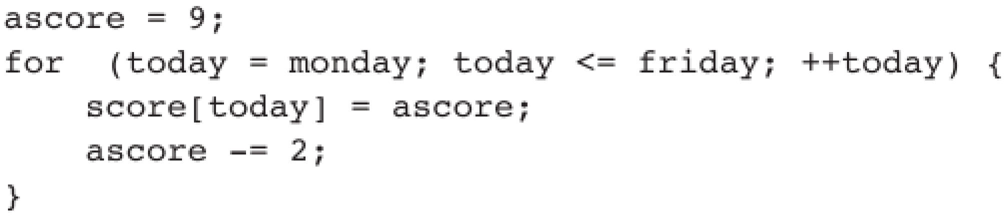
Another enum Example

typedef enum
{ Monday, Tuesday, Wednesday,
Thursday, Friday} weekday_t;

char answer [10]
int score [5]



answer[0]	T	score[monday]	9
answer[1]	F	score[tuesday]	7
answer[2]	F	score[wednesday]	5
	. . .	score[thursday]	3
answer[9]	T	score[friday]	1



```
ascore = 9;
for  (today = monday; today <= friday; ++today) {
     score[today] = ascore;
     ascore -= 2;
}
```

STRUCTURES

User-Defined Structure Types

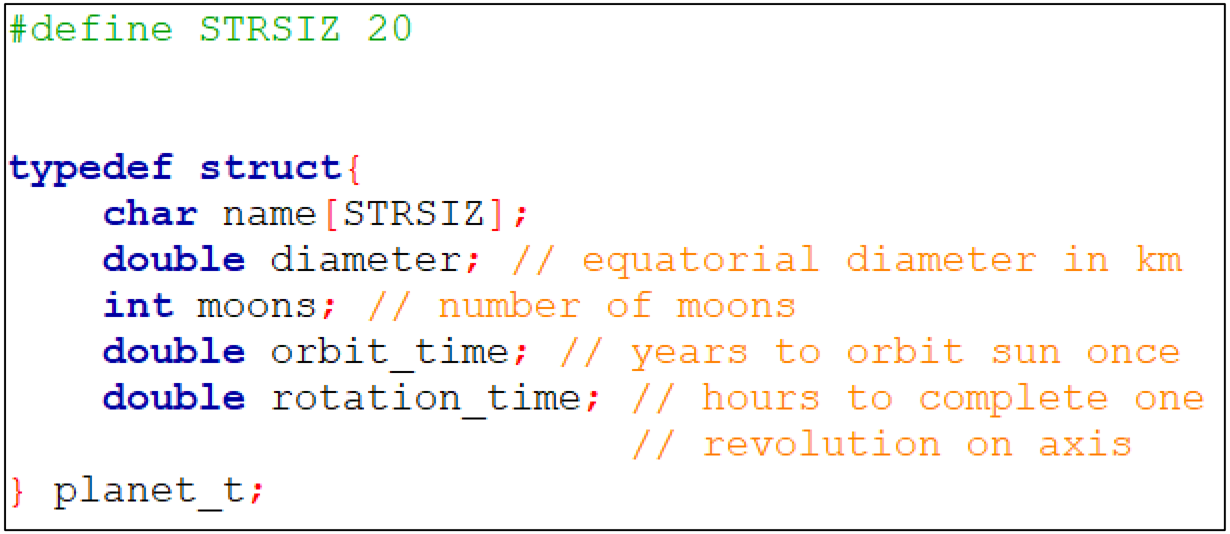
- record
  - a collection of information about one data object in a database
- structure type
  - a data type for a record composed of multiple components
- hierarchical structure
  - a structure containing components that are structures, e.g. array, struct

User-Defined Structure Types

- Assume that you want to create a template which describes the format of a planet. A planet has some properties which we call components, e.g.

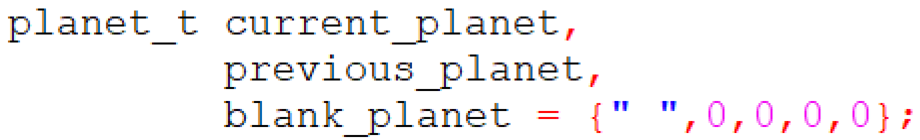
- Name: Jupiter
- Diameter: 142.800km
- Moons: 16
- Orbit time: 11.9 years
- Rotation time: 9.925 hours

User-Defined Structure Types



```
#define STRSIZ 20


typedef struct{
    char name[STRSIZ];
    double diameter; // equatorial diameter in km
    int moons; // number of moons
    double orbit_time; // years to orbit sun once
    double rotation_time; // hours to complete one
                          // revolution on axis
} planet_t;
```

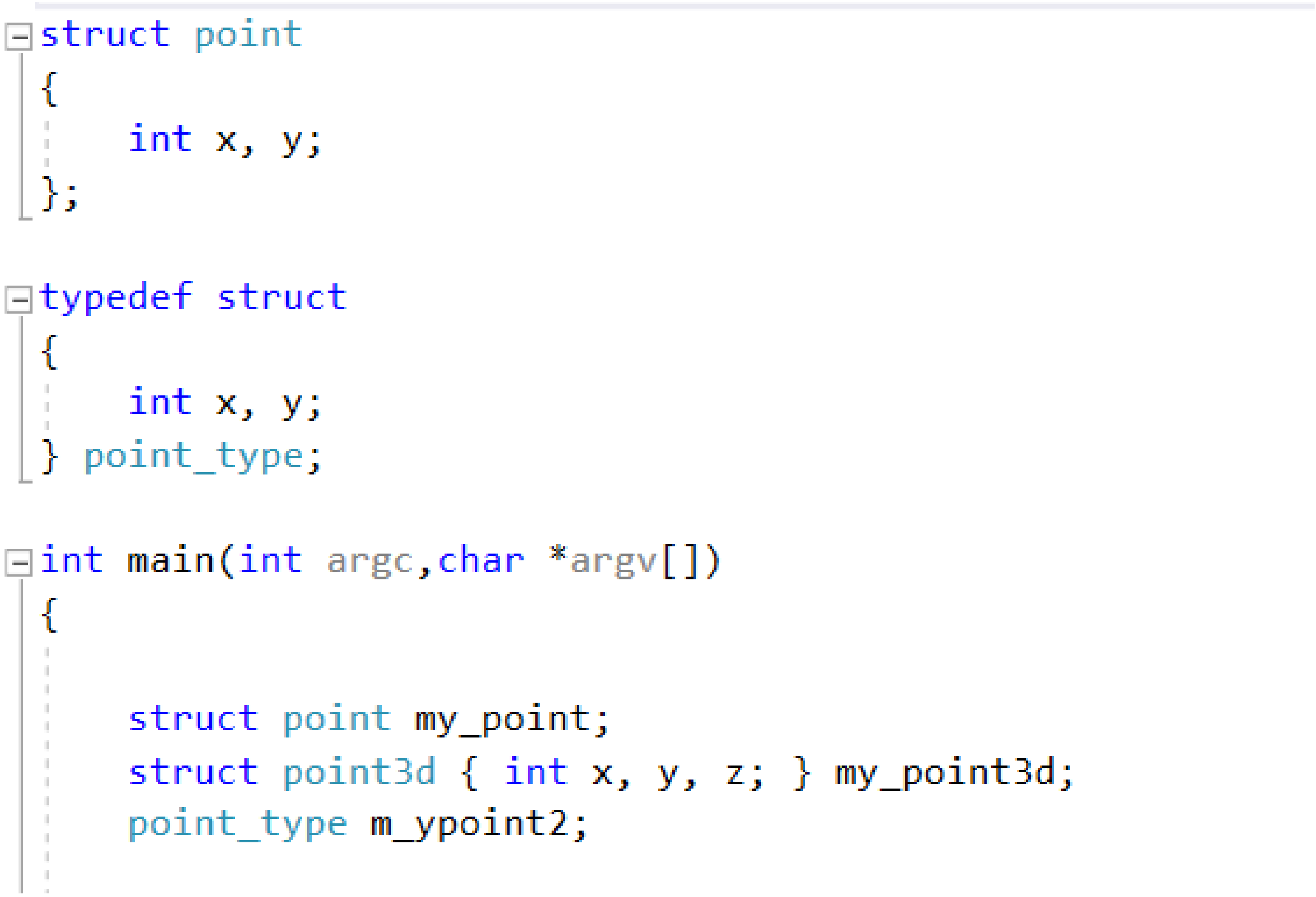
- This typedef definition itself allocates no memory. To allocate, declare a variable of this struct type:



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

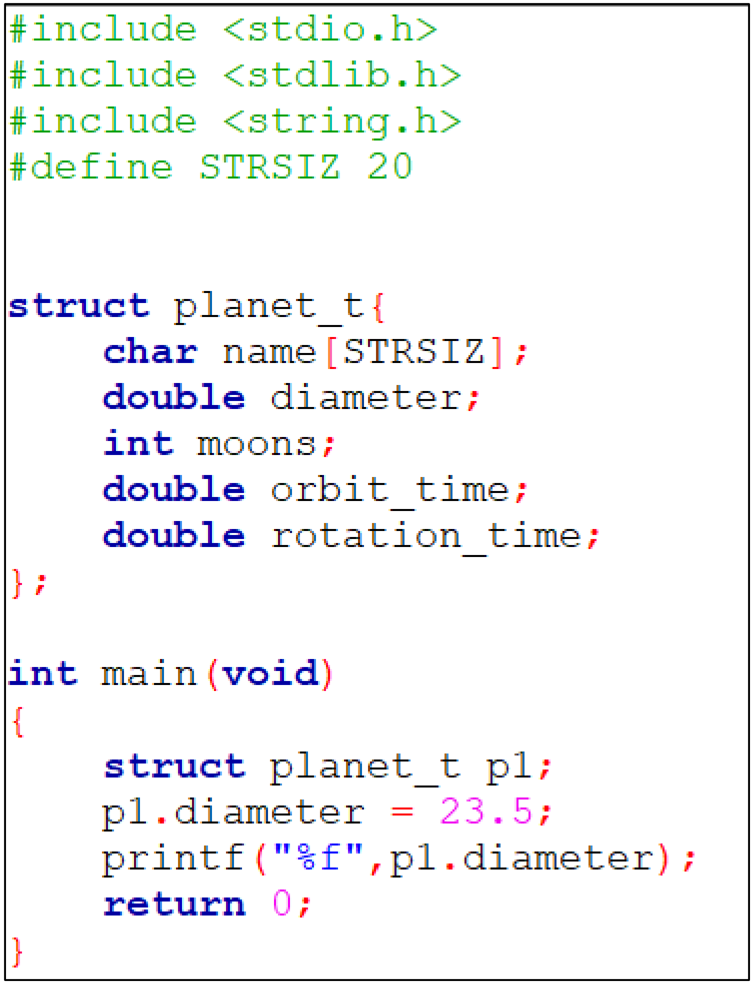
If there are fewer initializers in the list than members in the structure, the rest are automatically initialized to 0 or NULL.

Alternative Ways



```
struct point
{
    int x, y;
};

typedef struct
{
    int x, y;
} point_type;

int main(int argc,char *argv[])
{

    struct point my_point;
    struct point3d { int x, y, z; } my_point3d;
    point_type m_ypoint2;

```

Alternative Convention



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define STRSIZ 20


struct planet_t{
    char name[STRSIZ];
    double diameter;
    int moons;
    double orbit_time;
    double rotation_time;
};

int main(void)
{
    struct planet_t pl;
    pl.diameter = 23.5;
    printf("%f",pl.diameter);
    return 0;
}
```

typedef merely creates a new name for an existing type therefore easy to use

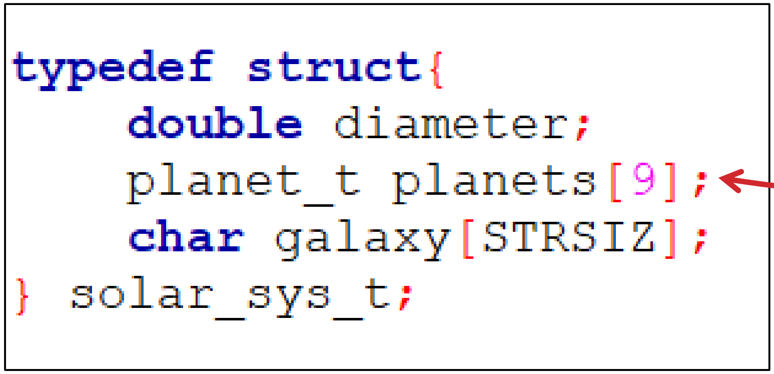
User-Defined Structure Types

Quick Check: Create a complex number structure.

```
typedef struct {
        double real_pt,
               imag_pt;
} complex_t;
```

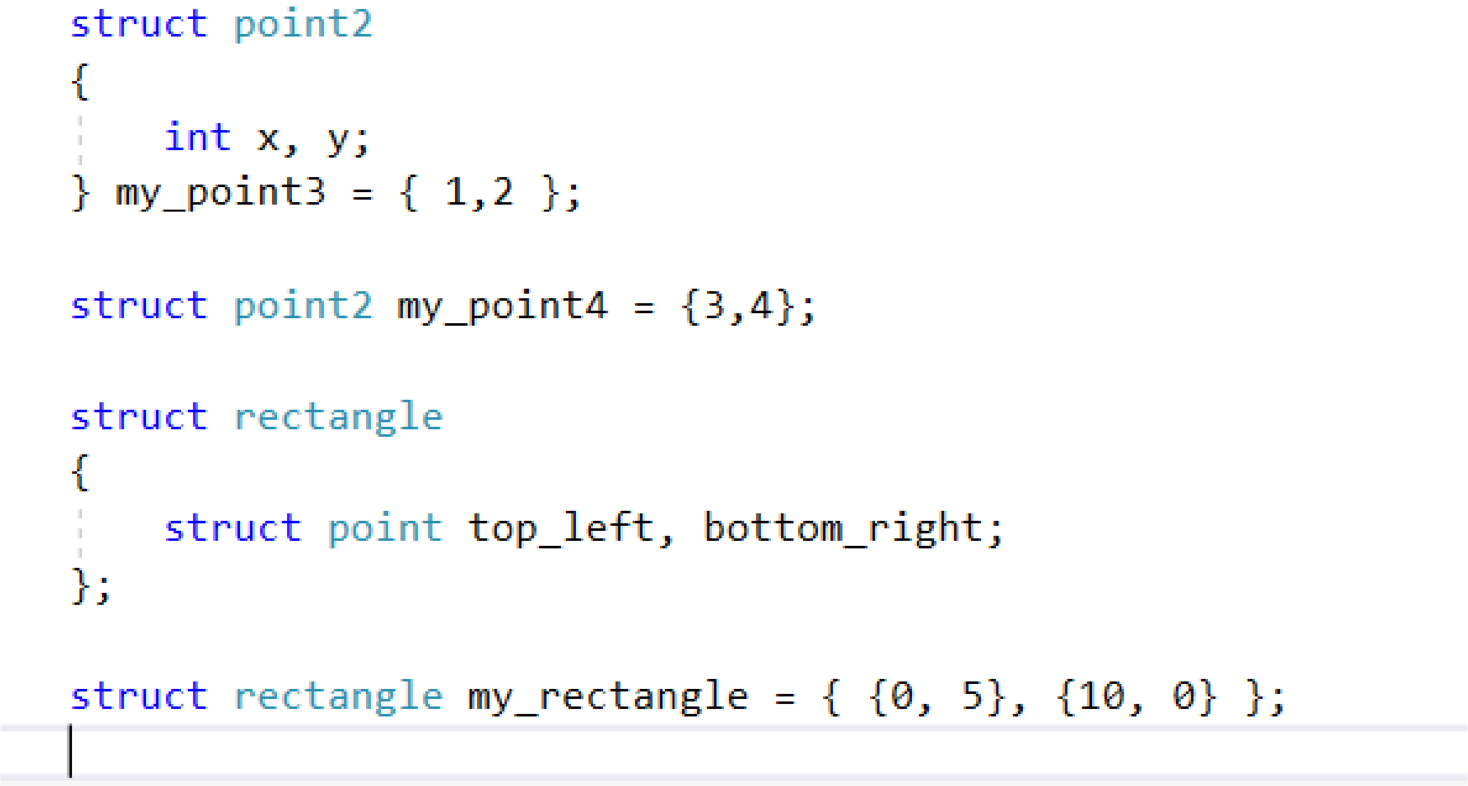
Hierarchical Structure

A structure containing components that are structures, e.g. array, struct.



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

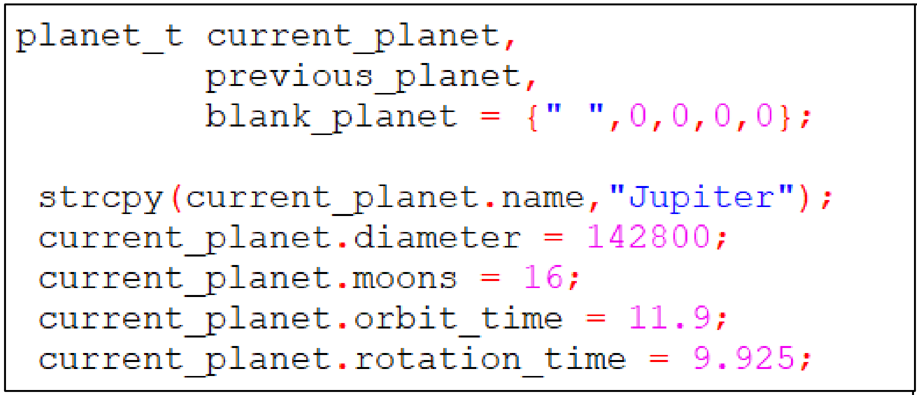
Initializing Structure Members



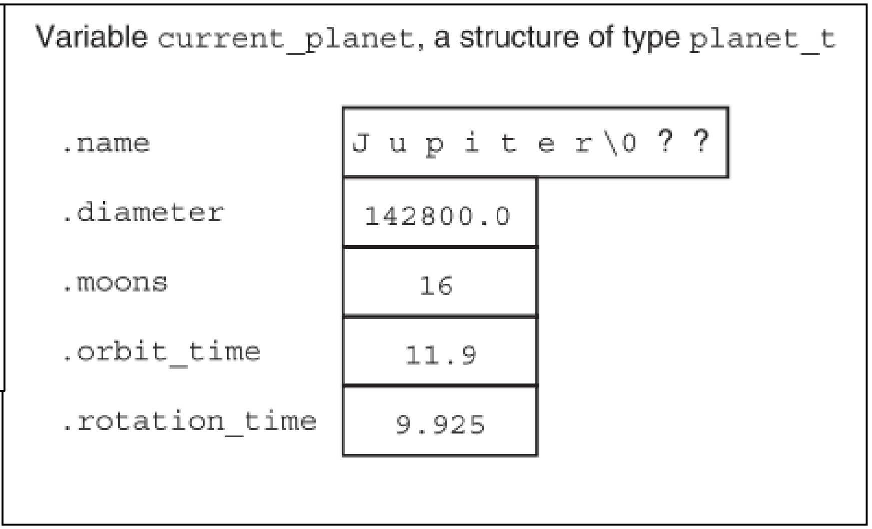
```
struct point2
{
    int x, y;
} my_point3 = { 1,2 };

struct point2 my_point4 = {3,4};

struct rectangle
{
    struct point top_left, bottom_right;
};

struct rectangle my_rectangle = { {0, 5}, {10, 0} };
```

Manipulate Individual Components of a Structured Data Object

- direct component selection operator
  - a period placed between a structure type variable and a component name to create a reference to the component



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

 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

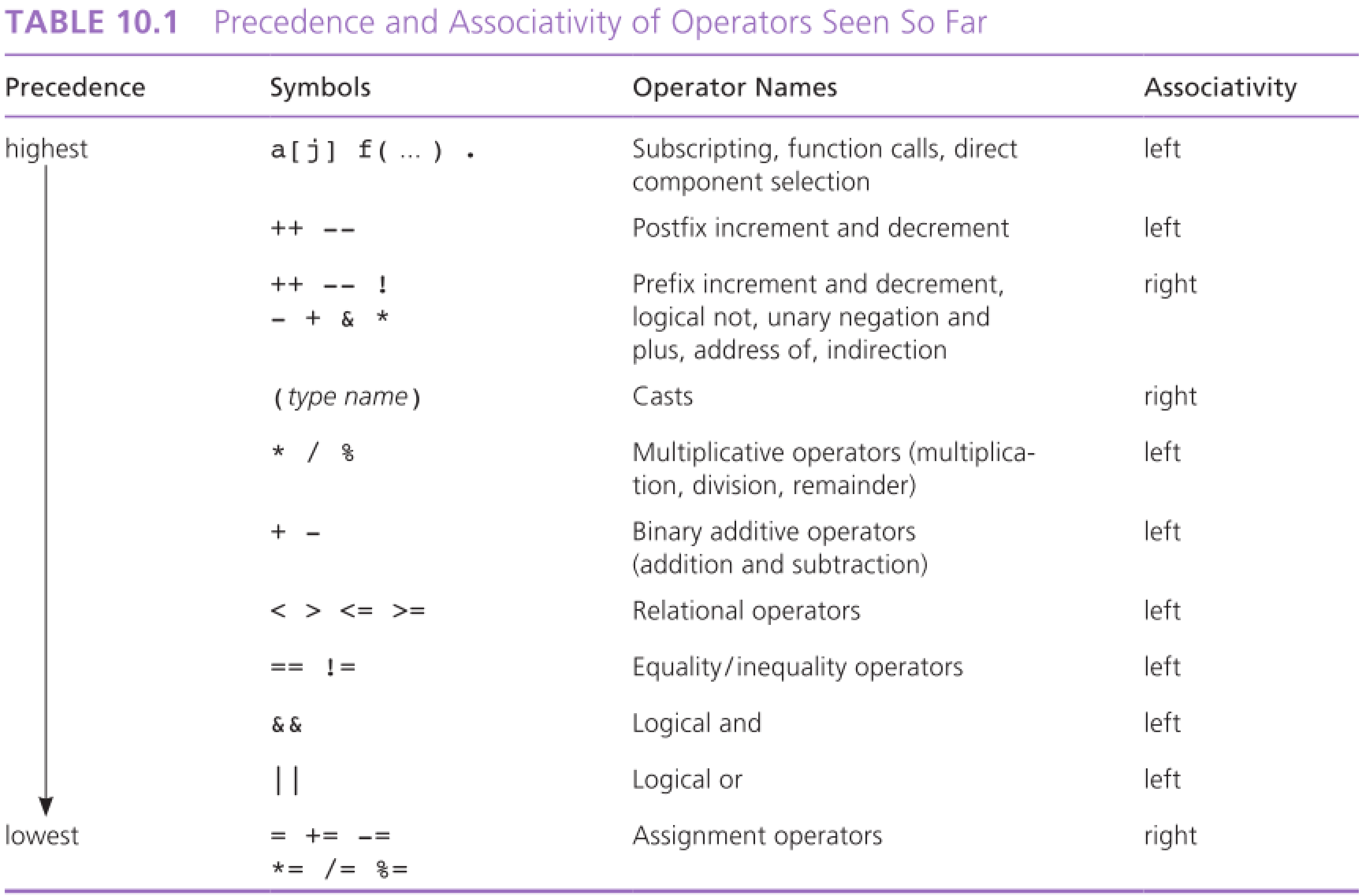


TABLE 10.1 Precedence and Associativity of Operators Seen So Far

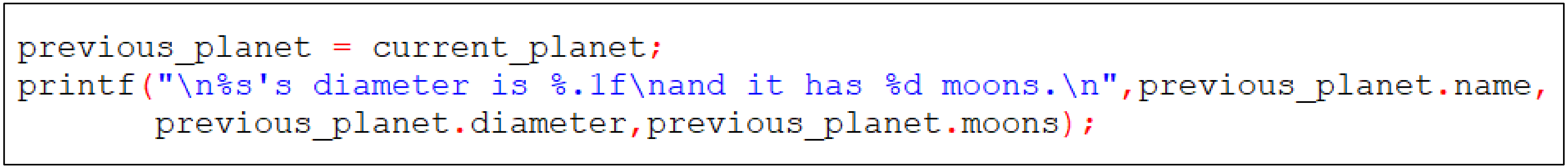
Precedence	Symbols	Operator Names	Associativity
highest	`a[j] f(...) .`	Subscripting, function calls, direct component selection	left
	`++ --`	Postfix increment and decrement	left

	`++ -- !`
	`- + & *`
		Prefix increment and decrement, logical not, unary negation and plus, address of, indirection	right

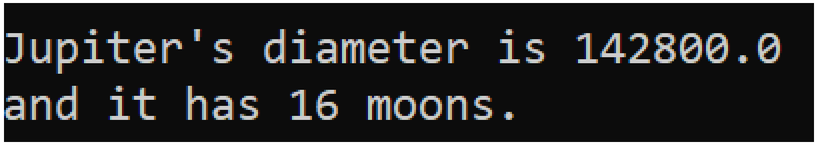
	`(`*type name*`)`	Casts	right
	`* / %`	Multiplicative operators (multiplication, division, remainder)	left
	`+ -`	Binary additive operators (addition and subtraction)	left
	`< > <= >=`	Relational operators	left
	`== !=`	Equality/inequality operators	left
	`&&`	Logical and	left
	`||`	Logical or	left

lowest	`= += -=`
	`*= /= %=`
		Assignment operators	right

Assignment Operator



```c
previous_planet = current_planet;
printf("\n%s's diameter is %.1f\nand it has %d moons.\n",previous_planet.name,
       previous_planet.diameter,previous_planet.moons);
```



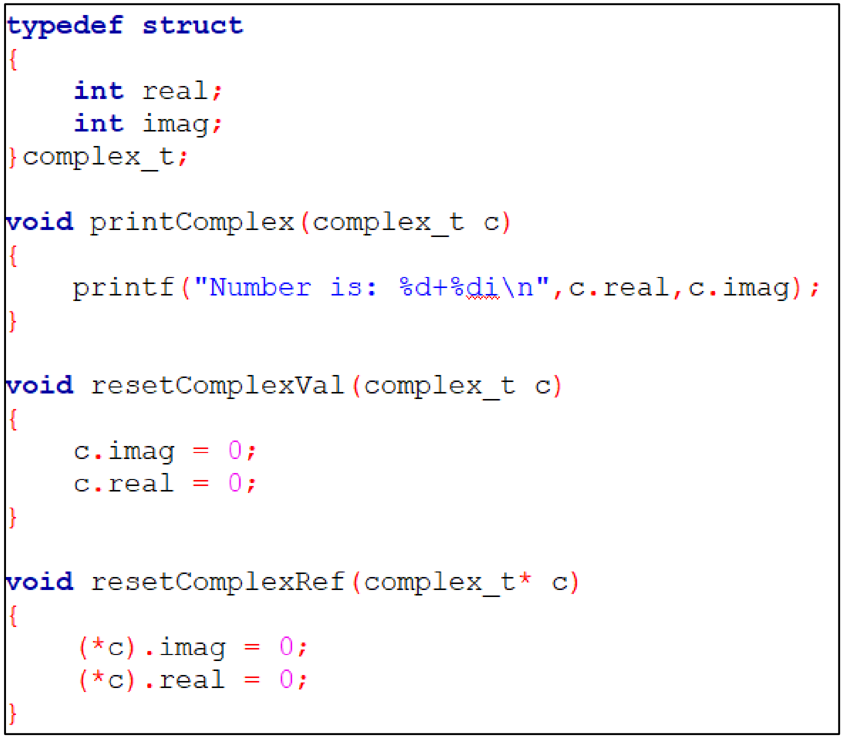
```
Jupiter's diameter is 142800.0
and it has 16 moons.
```

What if structure has pointer variables ?

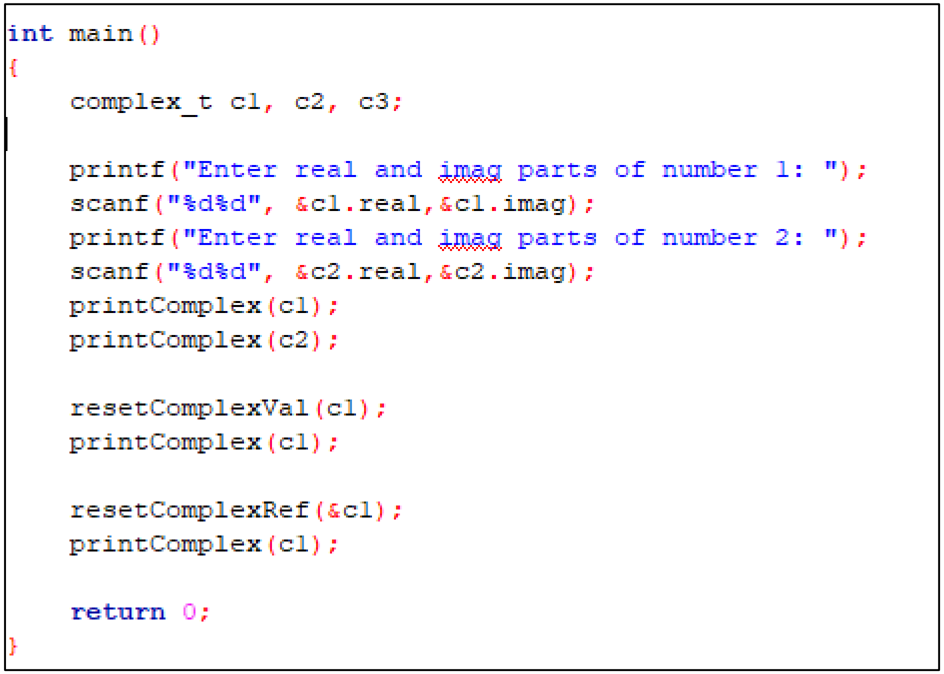
Structure Data Type as Input and Output Parameters

- 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 in the same way that we would pass output arguments of the standard types char, int, and double.

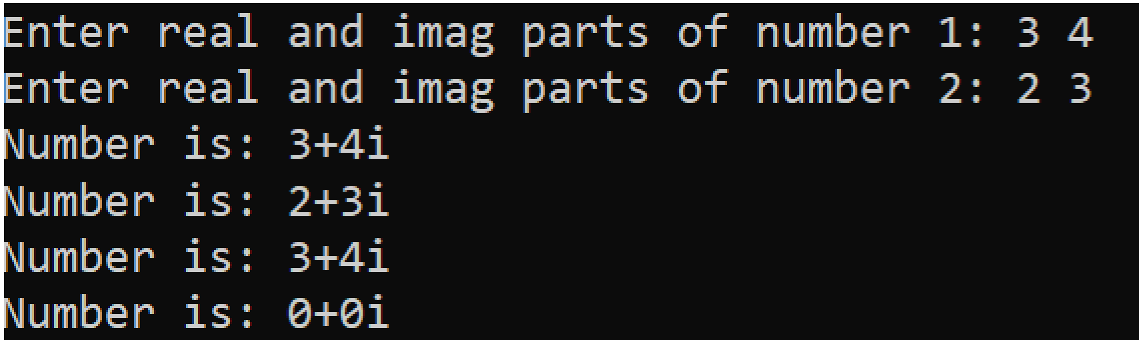
Pass by Value - Pass by Reference



```
typedef struct
{
    int real;
    int imag;
}complex_t;

void printComplex(complex_t c)
{
    printf("Number is: %d+%di\n",c.real,c.imag);
}

void resetComplexVal(complex_t c)
{
    c.imag = 0;
    c.real = 0;
}

void resetComplexRef(complex_t* c)
{
    (*c).imag = 0;
    (*c).real = 0;
}
```

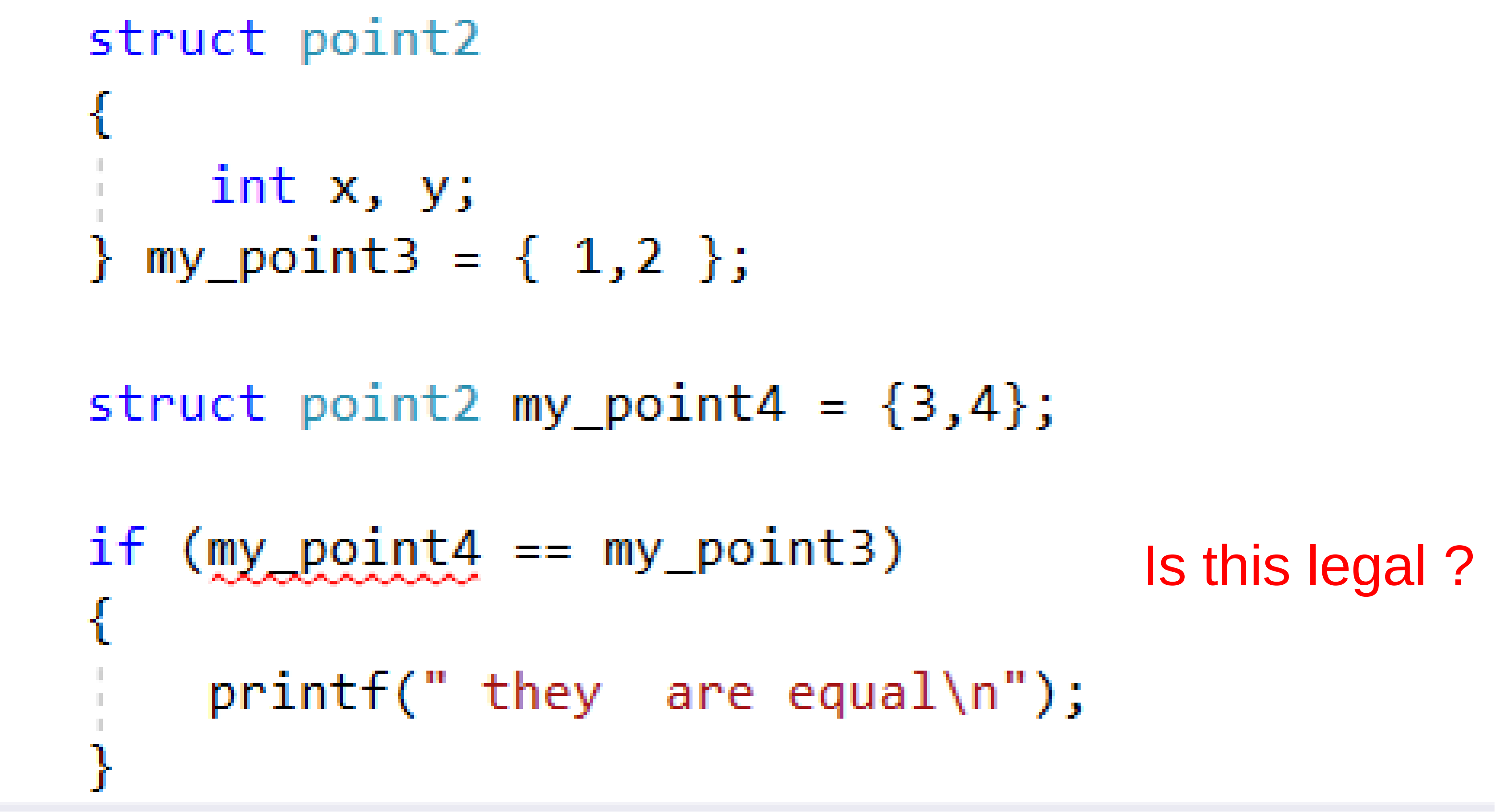


```
int main()
{
    complex_t c1, c2, c3;

    printf("Enter real and imag parts of number 1: ");
    scanf("%d%d", &c1.real,&c1.imag);
    printf("Enter real and imag parts of number 2: ");
    scanf("%d%d", &c2.real,&c2.imag);
    printComplex(c1);
    printComplex(c2);

    resetComplexVal(c1);
    printComplex(c1);

    resetComplexRef(&c1);
    printComplex(c1);

    return 0;
}
```



```
Enter real and imag parts of number 1: 3 4
Enter real and imag parts of number 2: 2 3
Number is: 3+4i
Number is: 2+3i
Number is: 3+4i
Number is: 0+0i
```

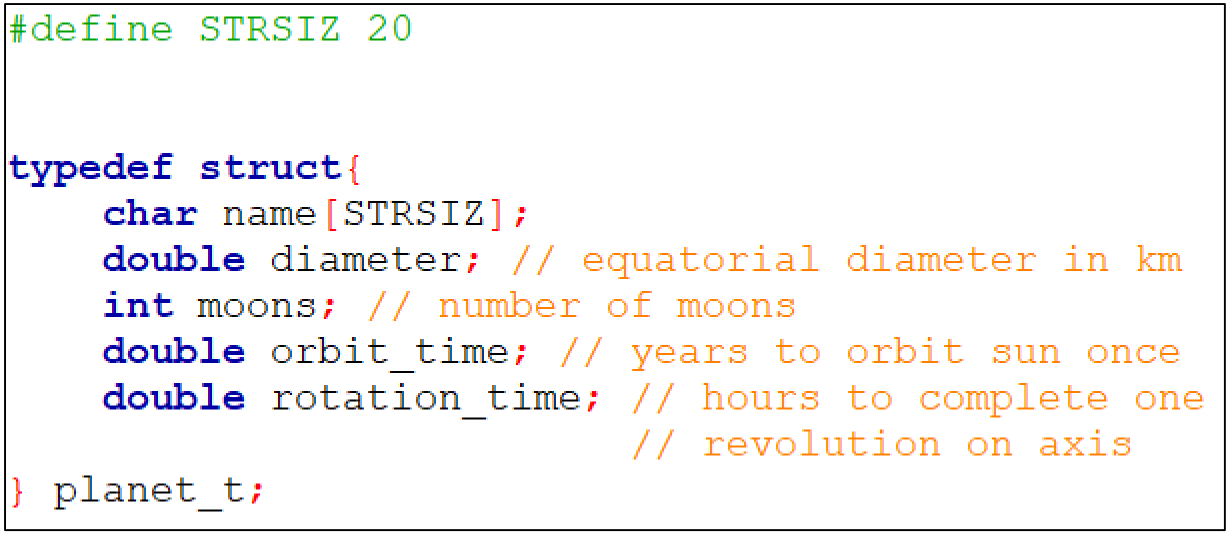
Equality Check



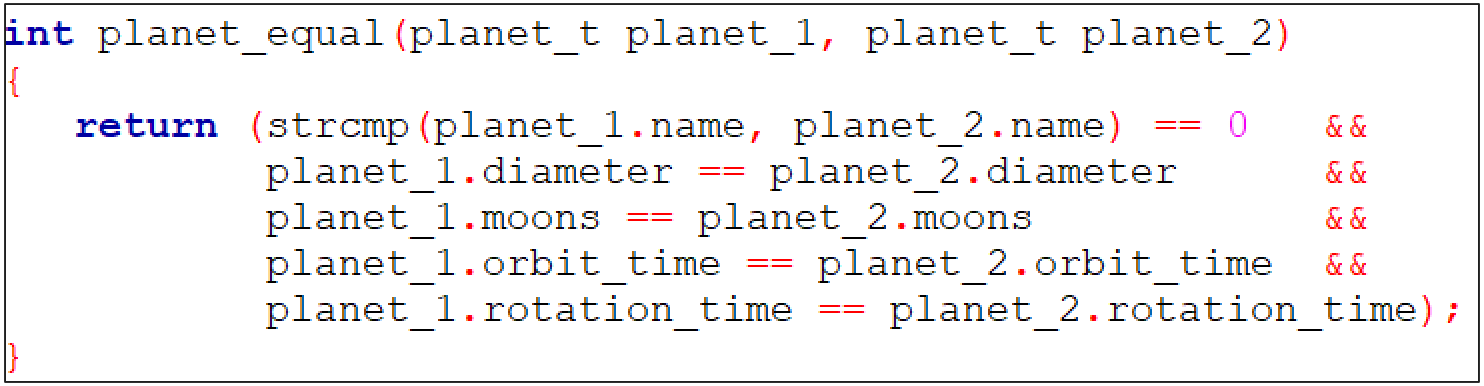
```
struct point2
{
    int x, y;
} my_point3 = { 1,2 };

struct point2 my_point4 = {3,4};

if (my_point4 == my_point3)
{
    printf(" they  are equal\n");
}
```

Is this legal ?

Equality Check

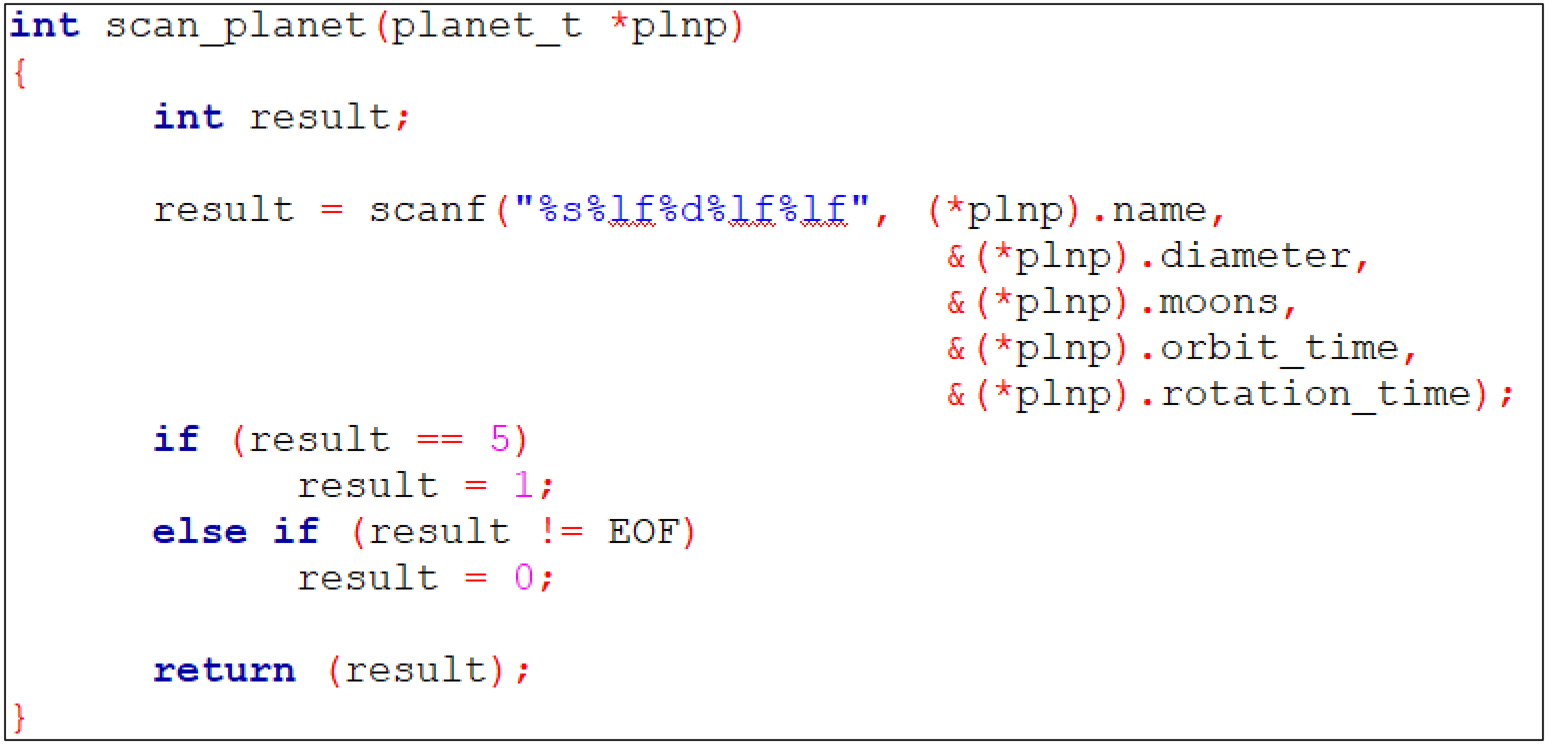


```
#define STRSIZ 20


typedef struct{
    char name[STRSIZ];
    double diameter; // equatorial diameter in km
    int moons; // number of moons
    double orbit_time; // years to orbit sun once
    double rotation_time; // hours to complete one
                          // revolution on axis
} planet_t;
```



```
int planet_equal(planet_t planet_1, planet_t planet_2)
{
   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);
}
```

Scan Function



```
int scan_planet(planet_t *plnp)
{
      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);
}
```

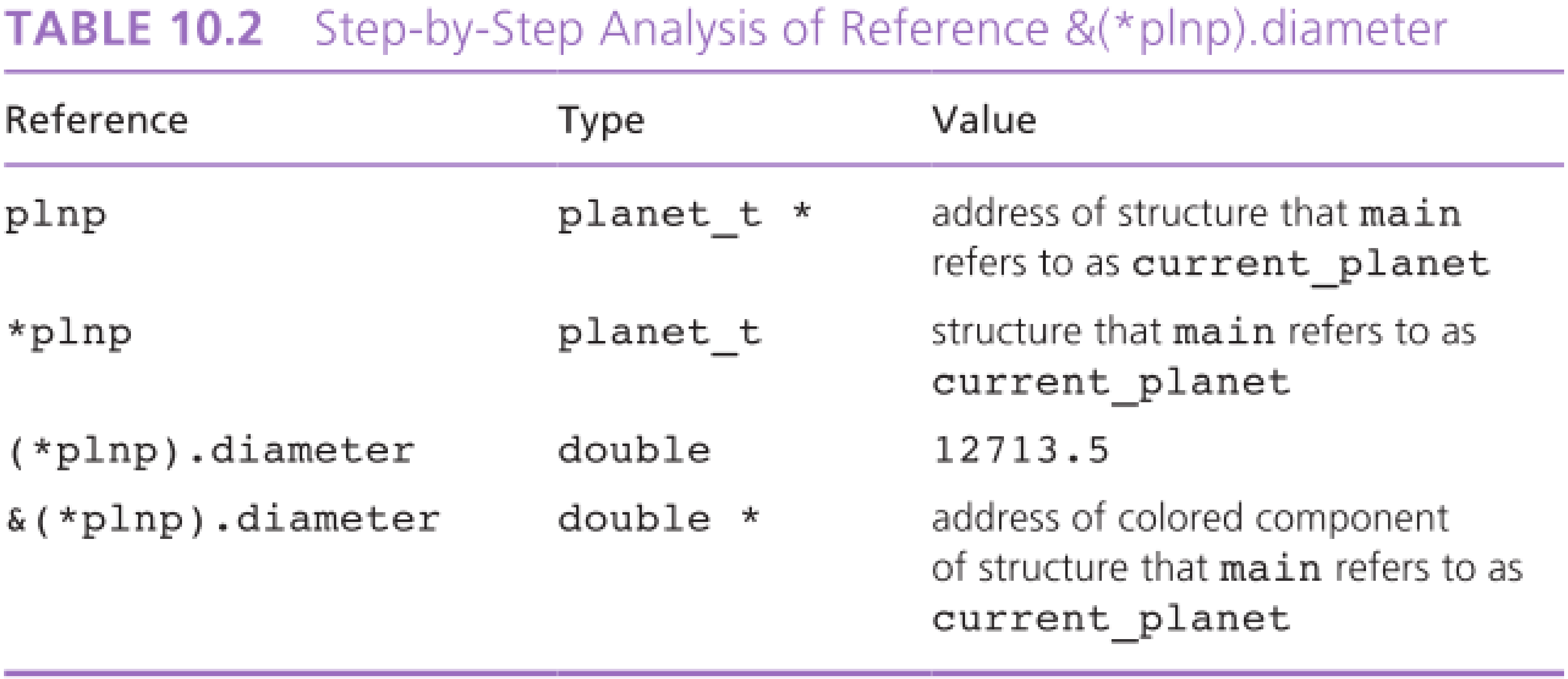
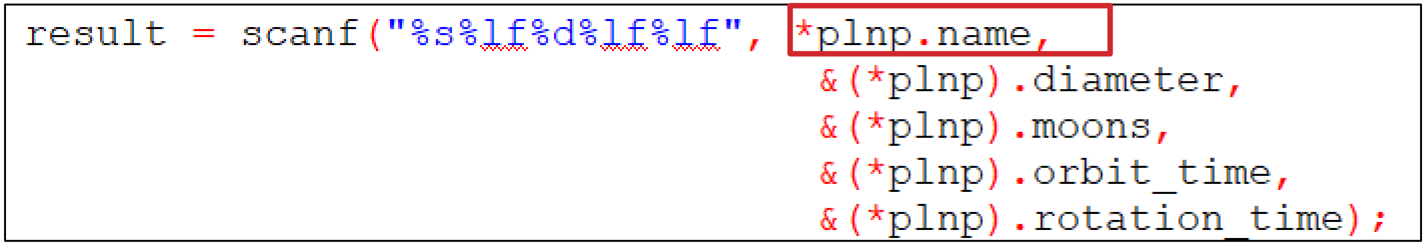


TABLE 10.2 Step-by-Step Analysis of Reference &(*plnp).diameter

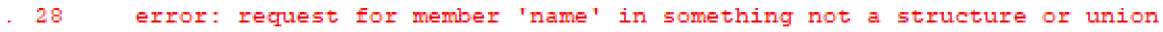
Reference	Type	Value
`plnp`	`planet_t *`	address of structure that `main` refers to as `current_planet`
`*plnp`	`planet_t`	structure that `main` refers to as `current_planet`
`(*plnp).diameter`	`double`	`12713.5`
`&(*plnp).diameter`	`double *`	address of colored component of structure that `main` refers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

Precedence

- Writing *plnp.name instead of (*plnp).name



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



```
. 28     error: request for member 'name' in something not a structure or union
```

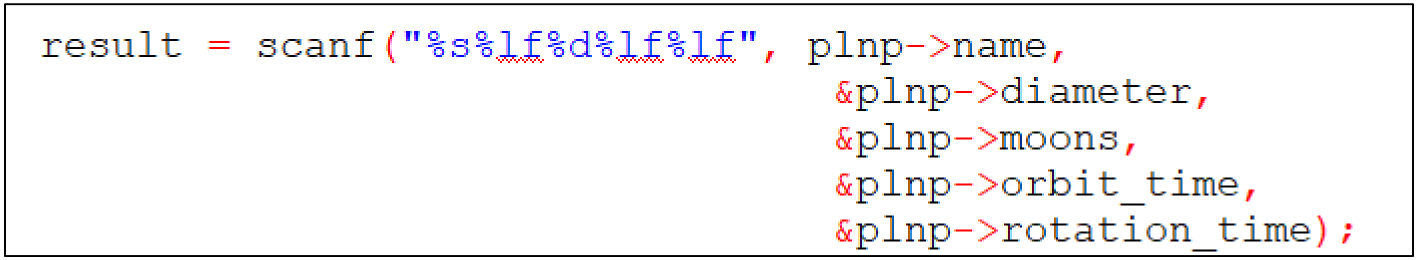
. (direct component selection dot) comes before

*(indirection) and &(address of) operators in precedence

Put parantheses!!

Structure Data Type as Input and Output Parameters

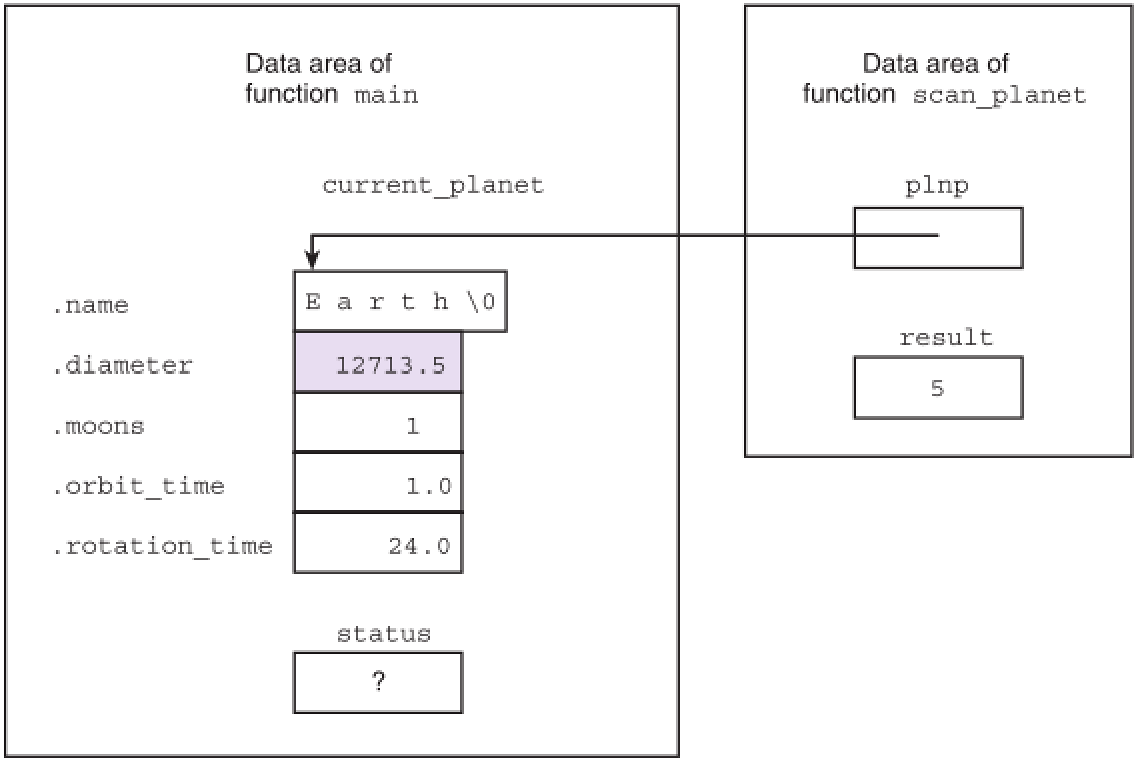
- indirect component selection operator
  - the character sequence **->** placed between a pointer variable and a component name creates a reference that follows the pointer to a structure and selects the component



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

FIGURE 10.5

Data Areas of main and scan_planet During Execution of `status = scan_planet (&current_planet);`



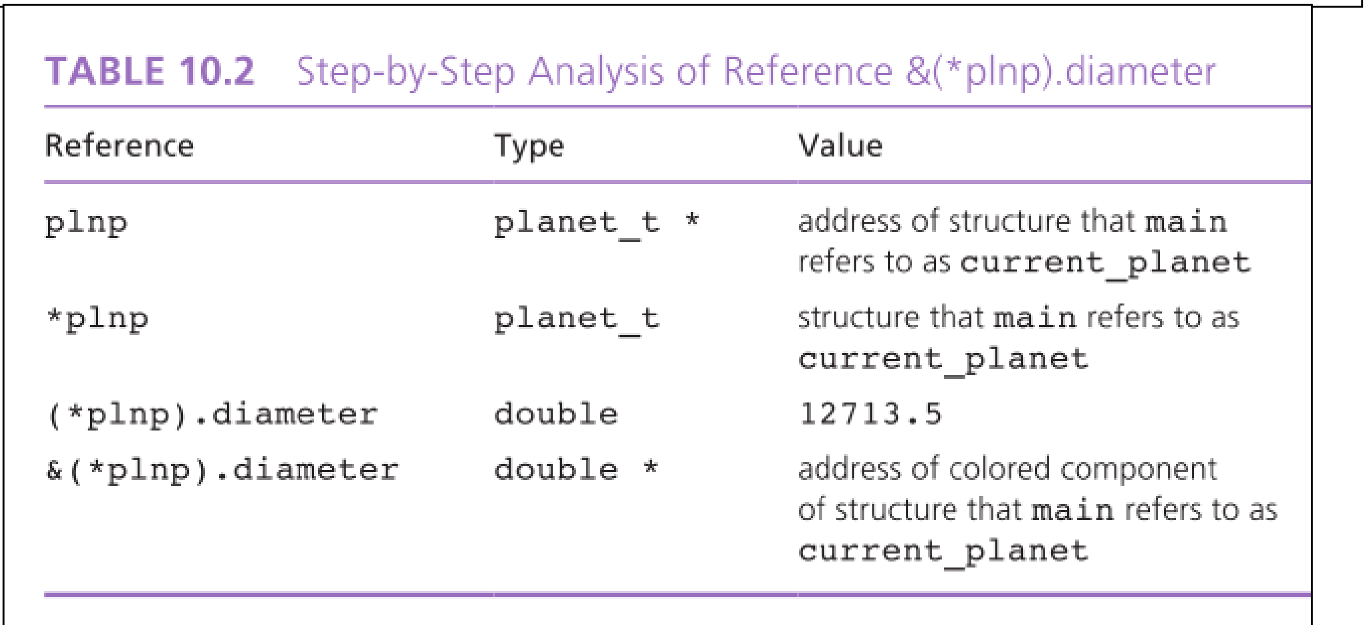


TABLE 10.2 Step-by-Step Analysis of Reference &(*plnp).diameter

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`

Functions Whose Result Values are Structured

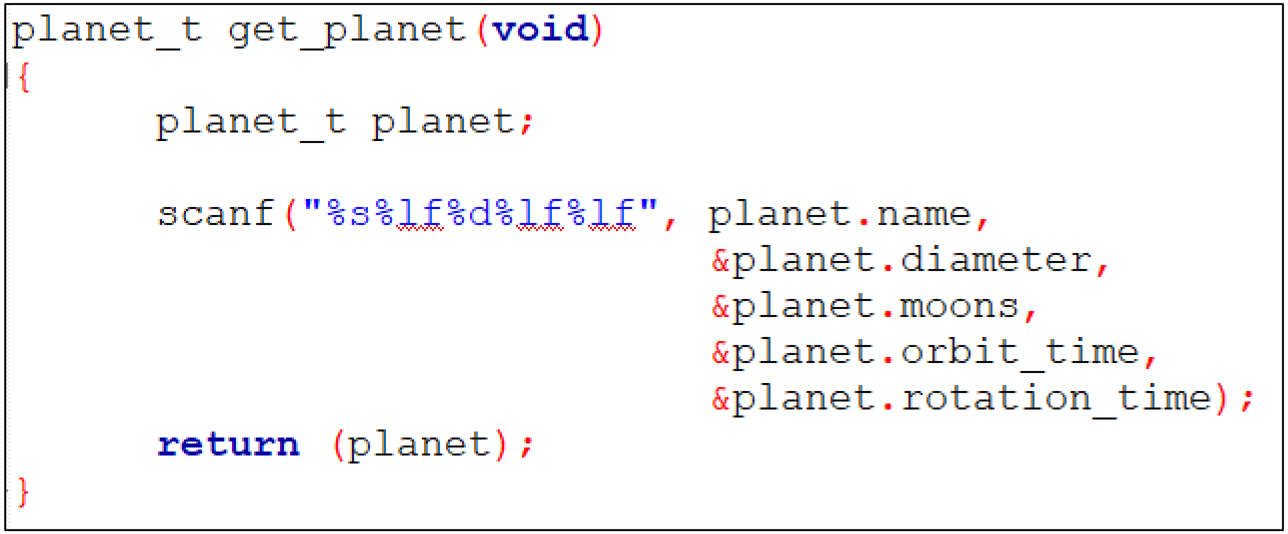
- A function that computes a structured result can be modeled on a function computing a simple result.

- A local variable of the structure type can be allocated, fill with the desired data, and returned as the function result.

Functions Whose Result Values are Structured

- The function does not return the *address* of the structure as it would with an array result.

- Rather, it returns the *values* of all components.



```
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);
}
```

current_planet = get_planet()

has the same effect as:

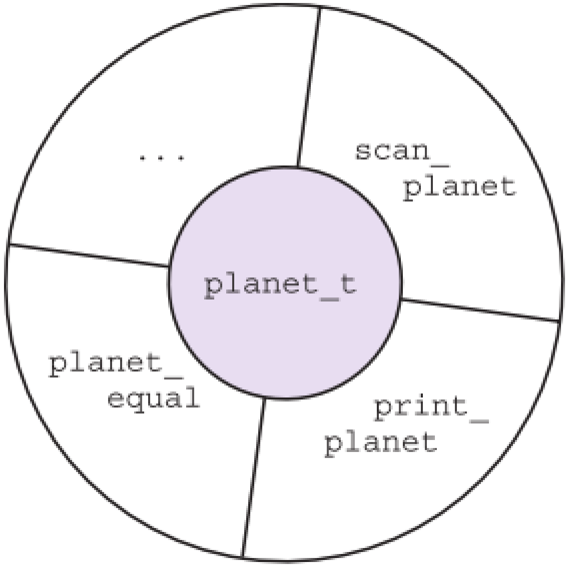
scan_planet(&current_planet)

Problem Solving with Structure Types

- abstract data type (ADT)
  - a data type combined with a set of basic operations

FIGURE 10.9

Data Type planet_t and Basic Operations



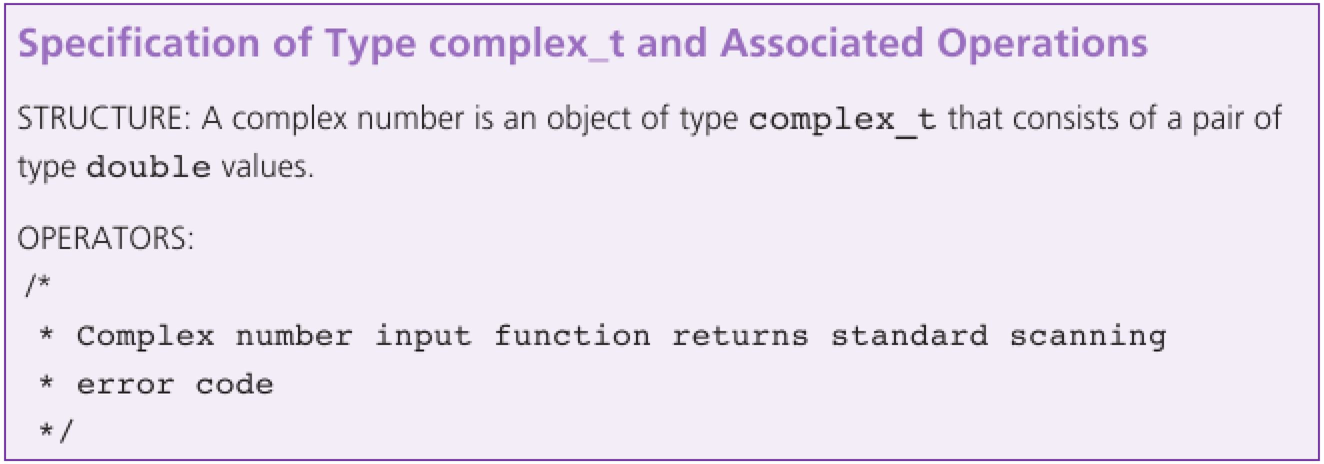
HOA 1: A User-Defined Type for Complex Numbers

- Create a structure called *complex_t* with 2 double components for real and imaginary parts and following functions:

- int scan_complex(complex_t *c);
- void print_complex(complex_t c);
- complex_t abs_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);

A USER-DEFİNED TYPE FOR COMPLEX NUMBERS

Case Study

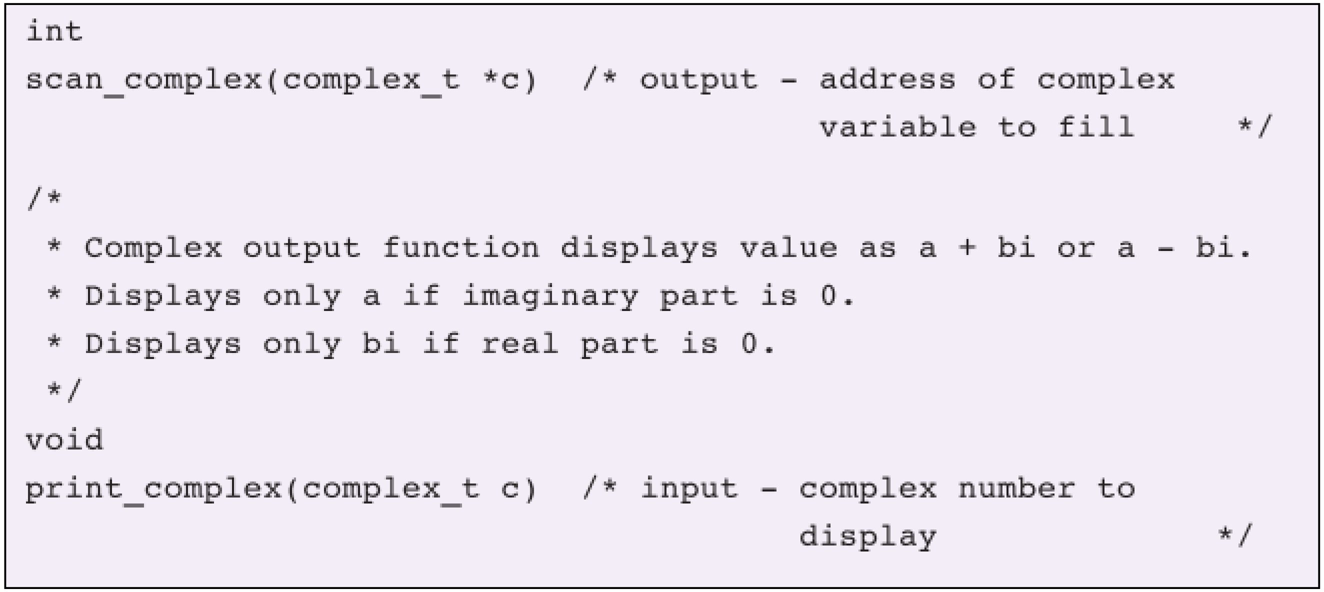


Specification of Type complex_t and Associated Operations

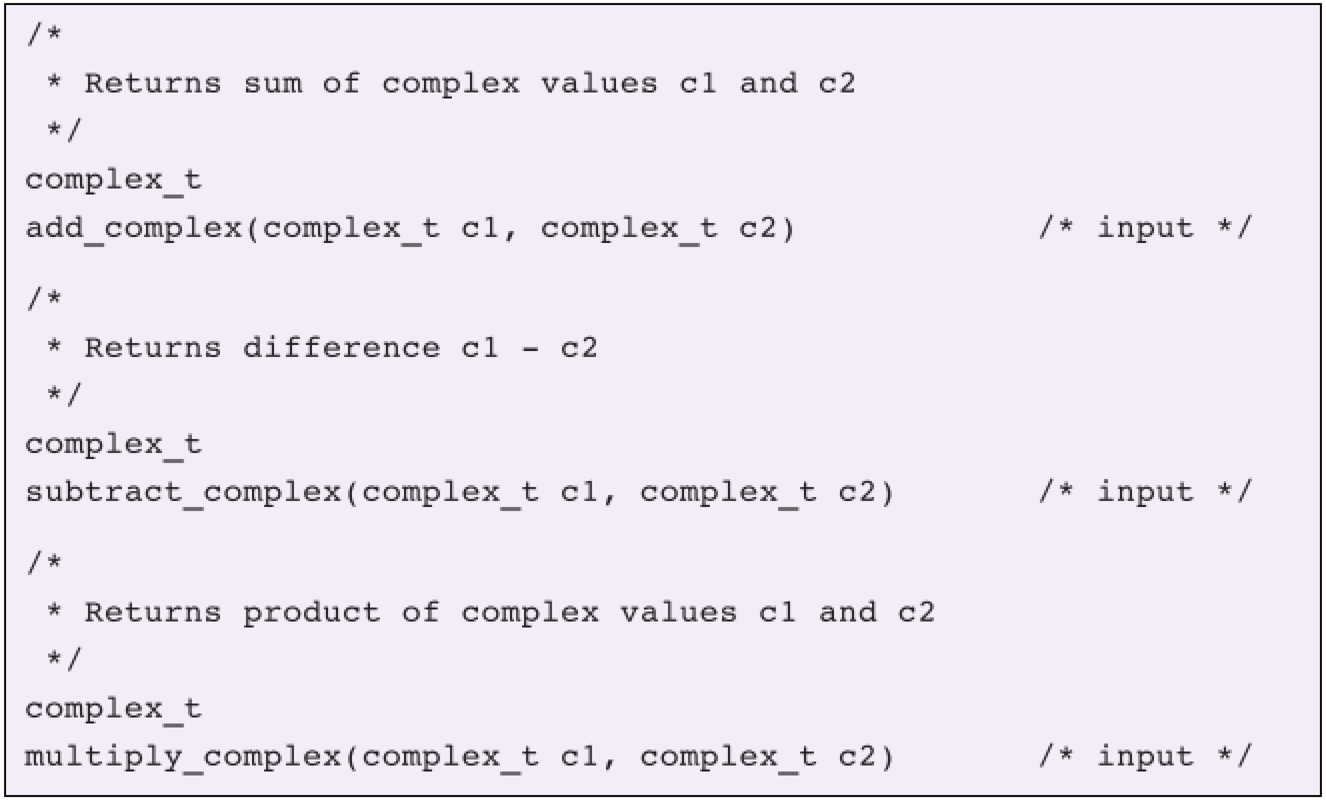
STRUCTURE: A complex number is an object of type `complex_t` that consists of a pair of type `double` values.

OPERATORS:

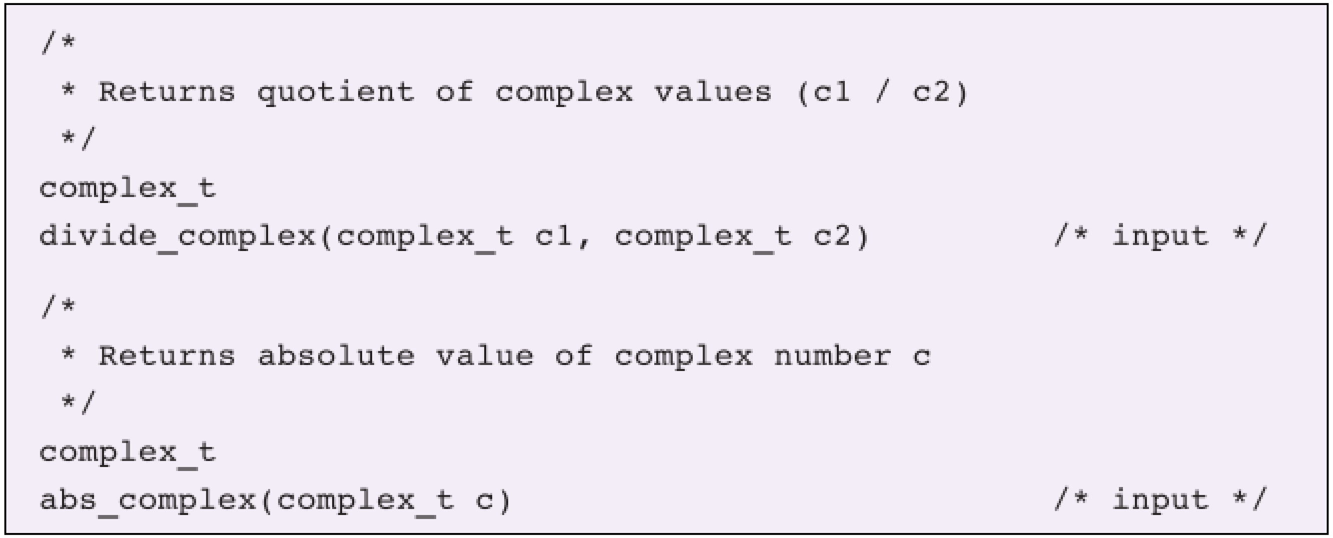
```
/*
 * Complex number input function returns standard scanning
 * error code
 */
```



```
int
scan_complex(complex_t *c)  /* output - address of complex
                                        variable to fill     */

/*
 * Complex output function displays value as a + bi or a - bi.
 * Displays only a if imaginary part is 0.
 * Displays only bi if real part is 0.
 */
void
print_complex(complex_t c)  /* input - complex number to
                                       display               */
```



```
/*
 * Returns sum of complex values c1 and c2
 */
complex_t
add_complex(complex_t c1, complex_t c2)            /* input */

/*
 * Returns difference c1 - c2
 */
complex_t
subtract_complex(complex_t c1, complex_t c2)       /* input */

/*
 * Returns product of complex values c1 and c2
 */
complex_t
multiply_complex(complex_t c1, complex_t c2)       /* input */
```



```
/*
 * Returns quotient of complex values (c1 / c2)
 */
complex_t
divide_complex(complex_t c1, complex_t c2)        /* input */

/*
 * Returns absolute value of complex number c
 */
complex_t
abs_complex(complex_t c)                          /* input */
```

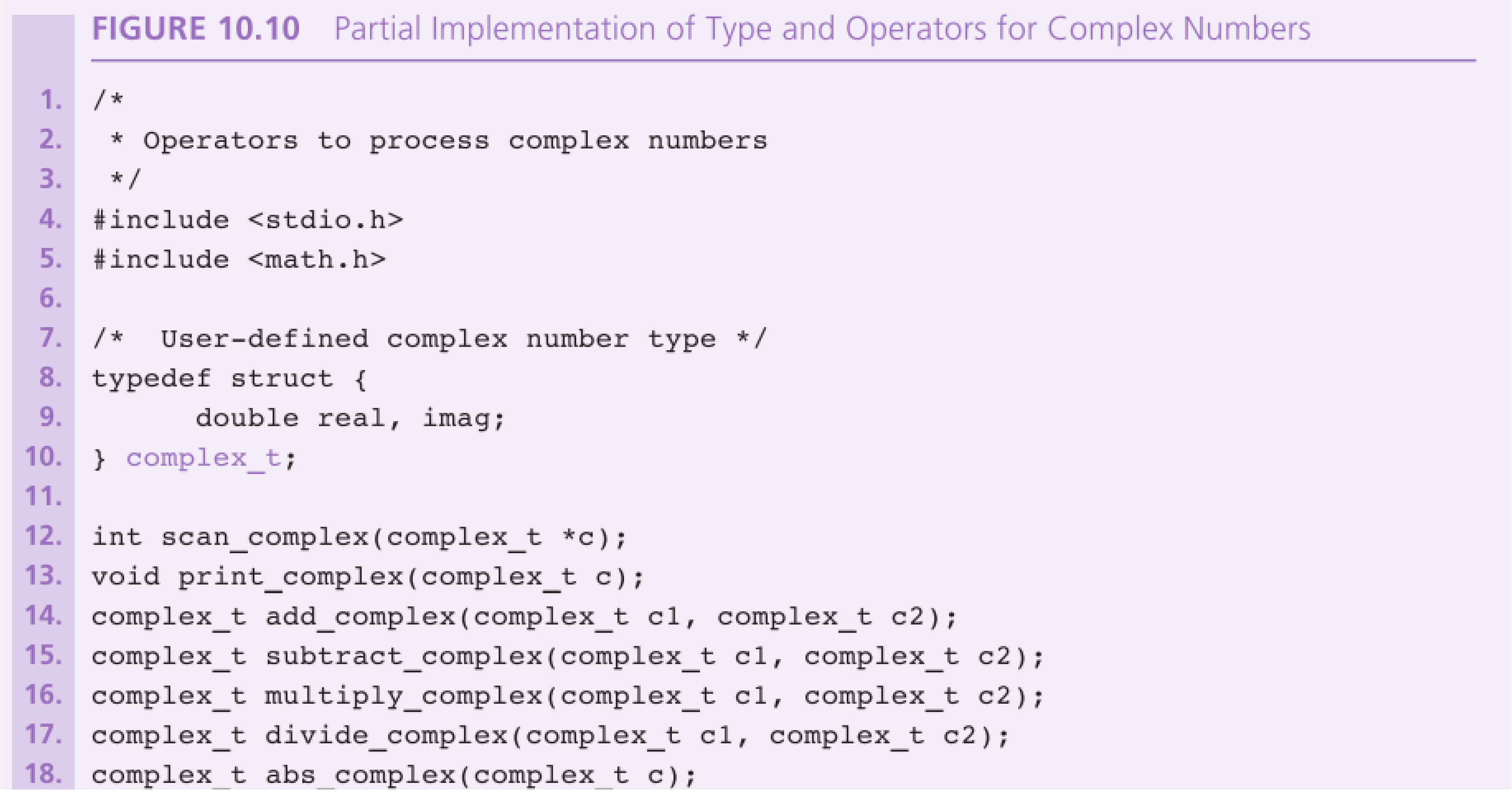
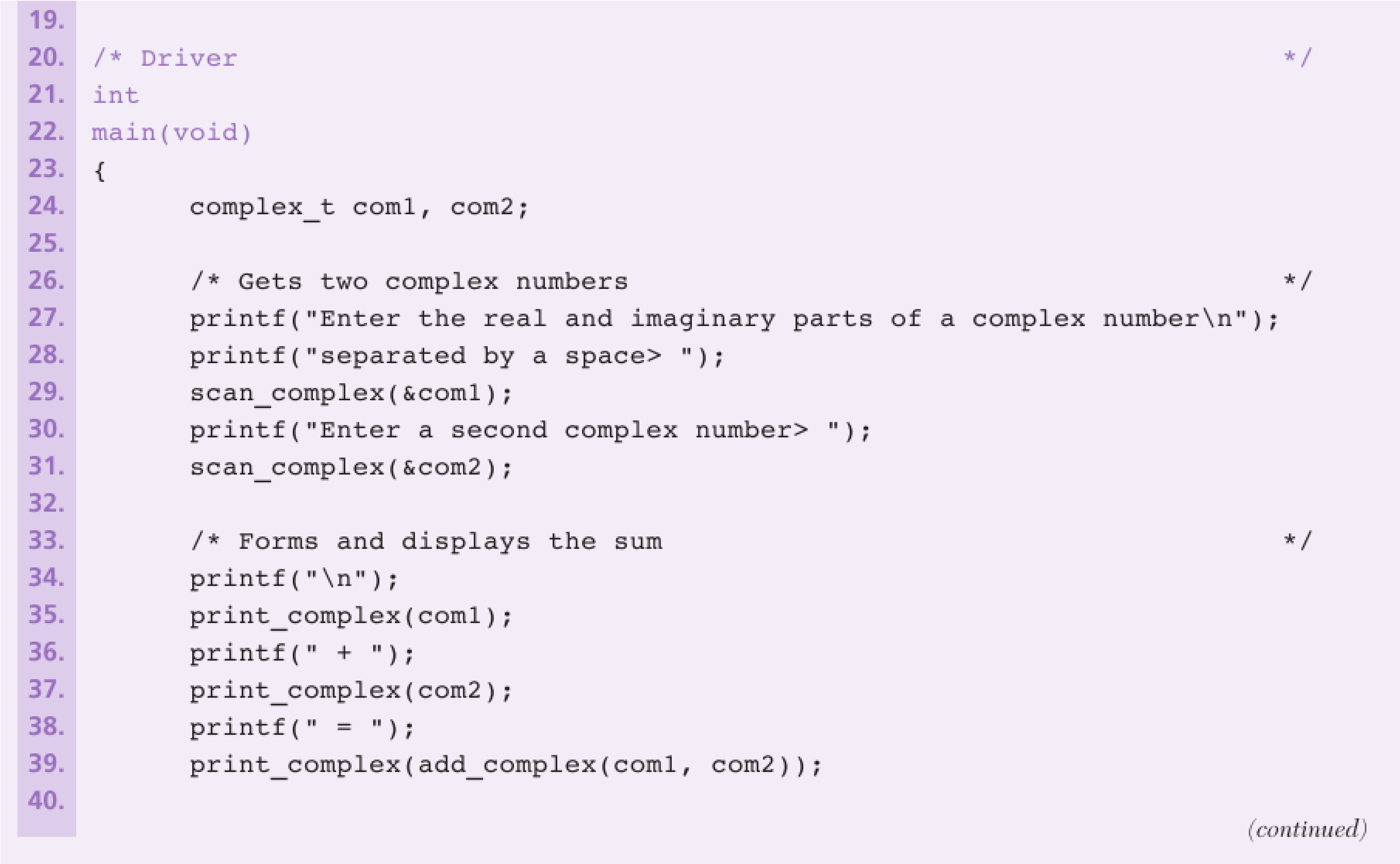


FIGURE 10.10 Partial Implementation of Type and Operators for Complex Numbers

```
/*
 * 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);
```



```

/* Driver                                                                  */
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));

```

(continued)

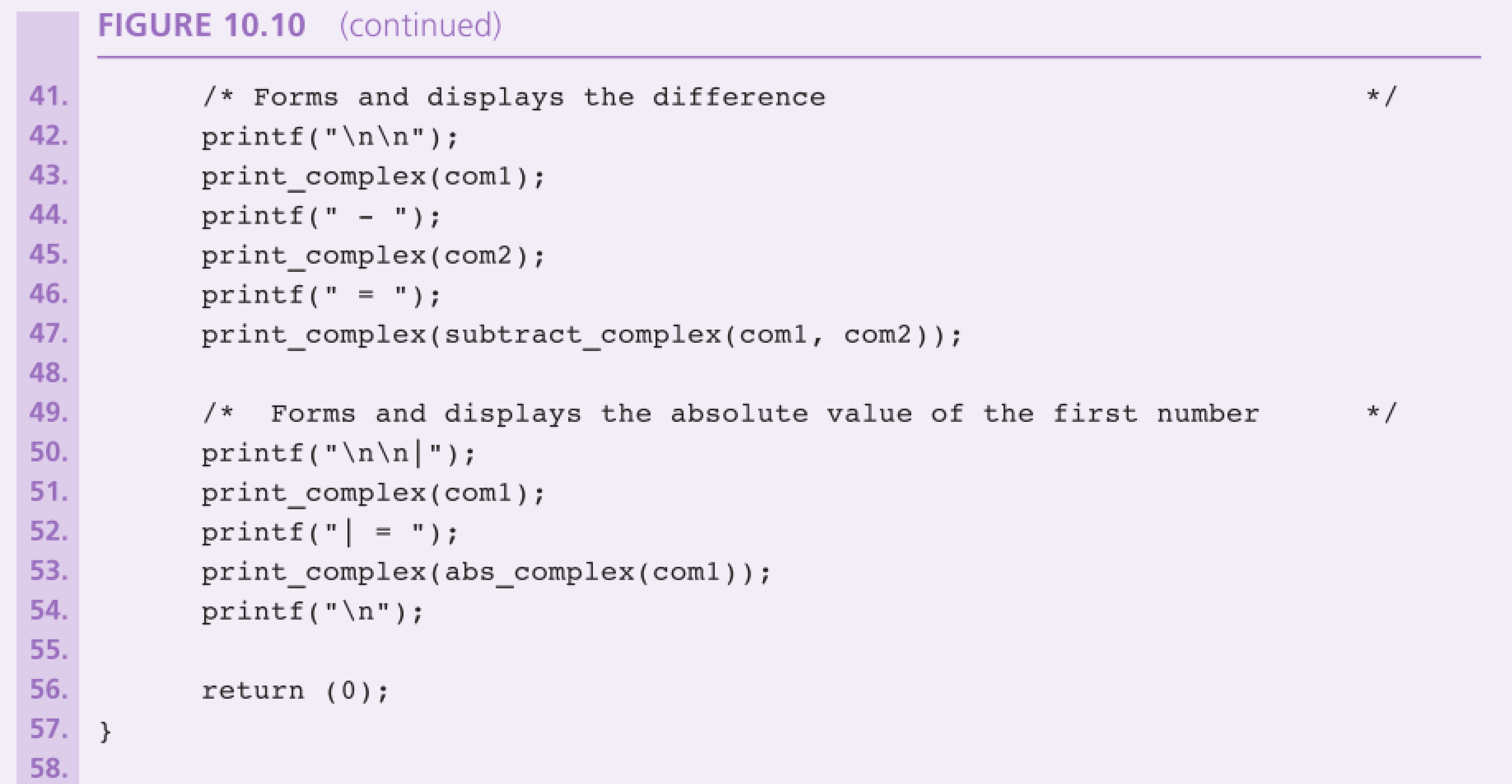
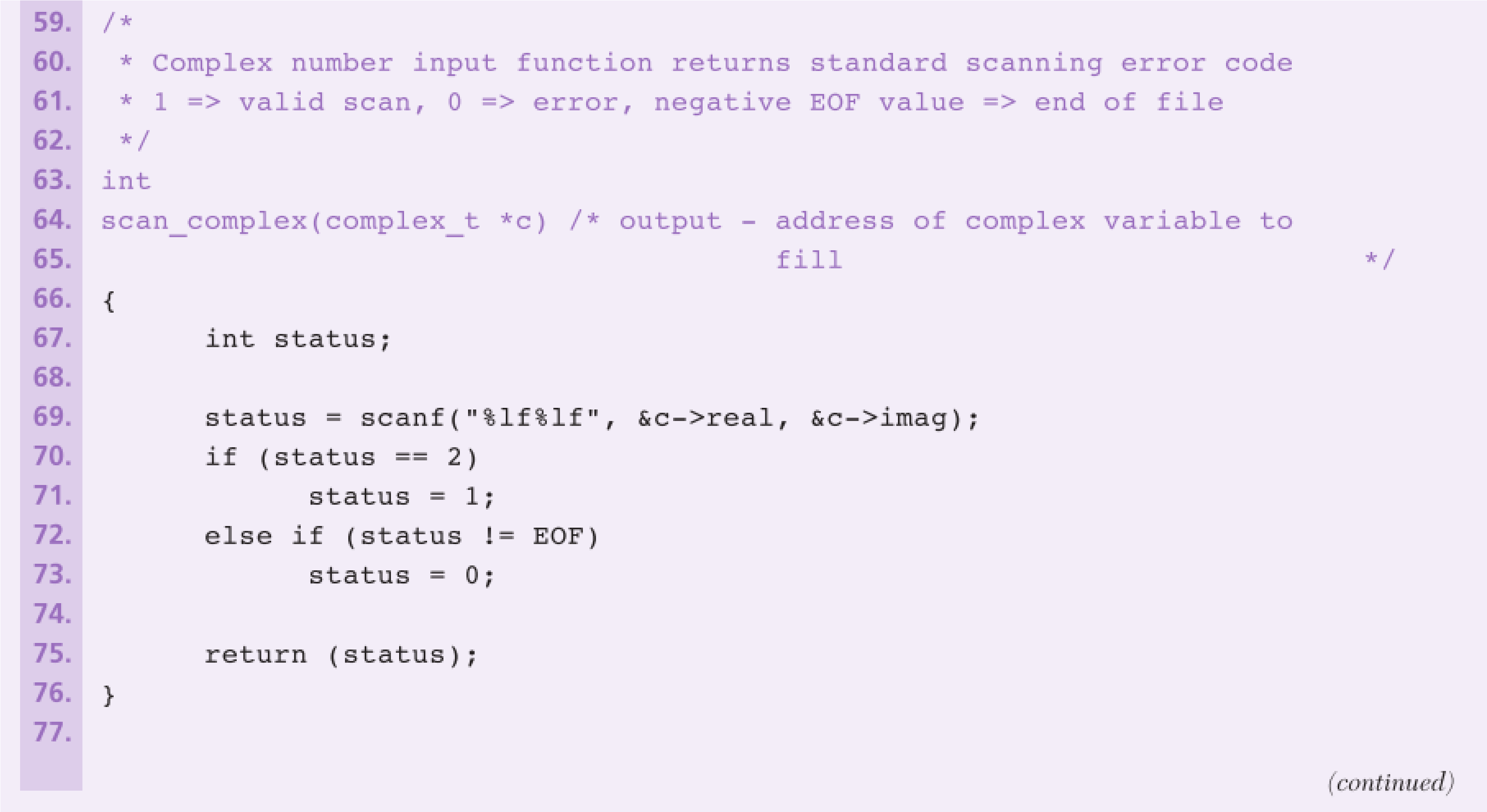


FIGURE 10.10 (continued)

```
      /* 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);
}
```

(continued)

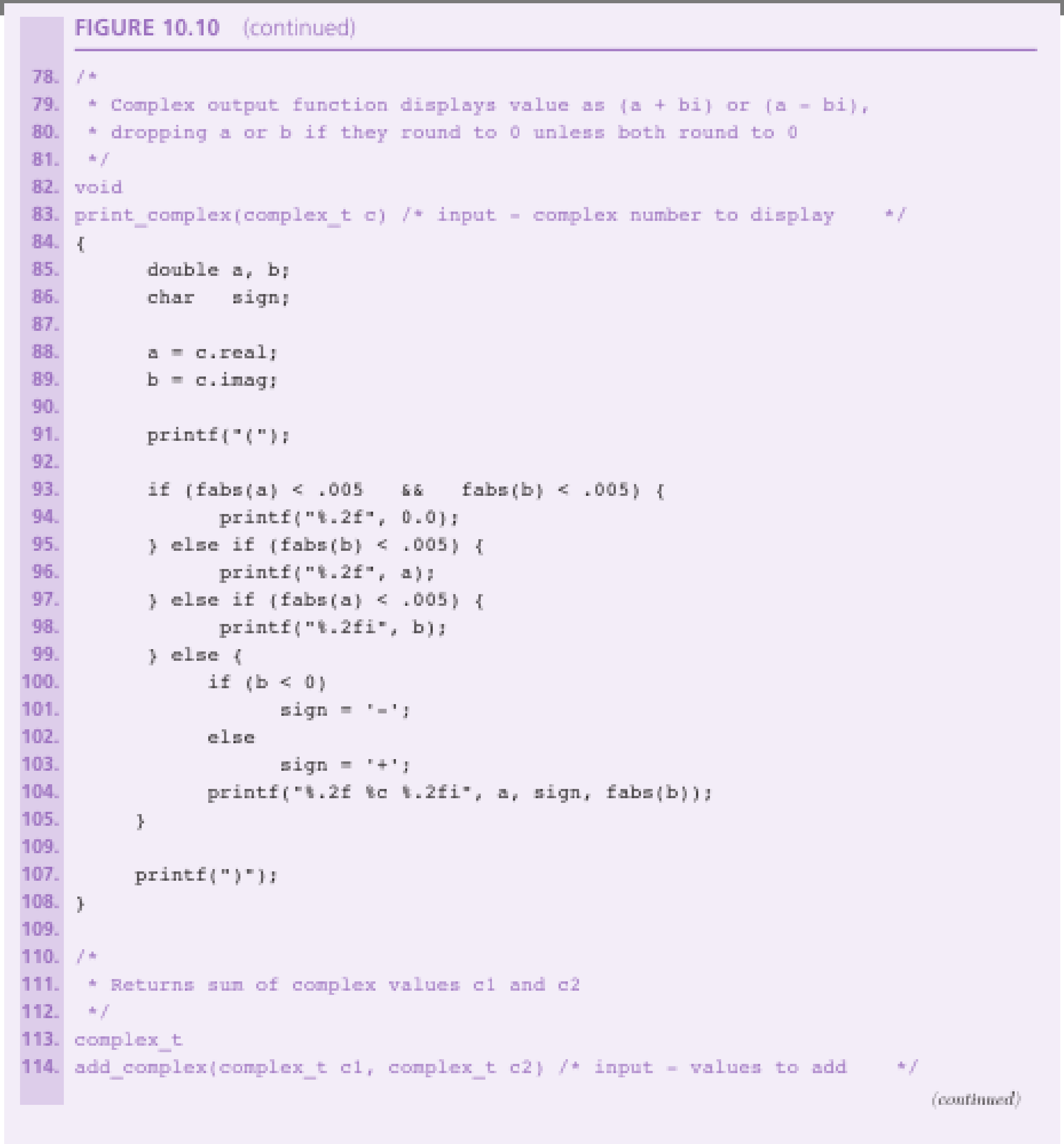


FIGURE 10.10 (continued)

```c
/*
 * 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));
     }

     printf(")");
}

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

(continued)

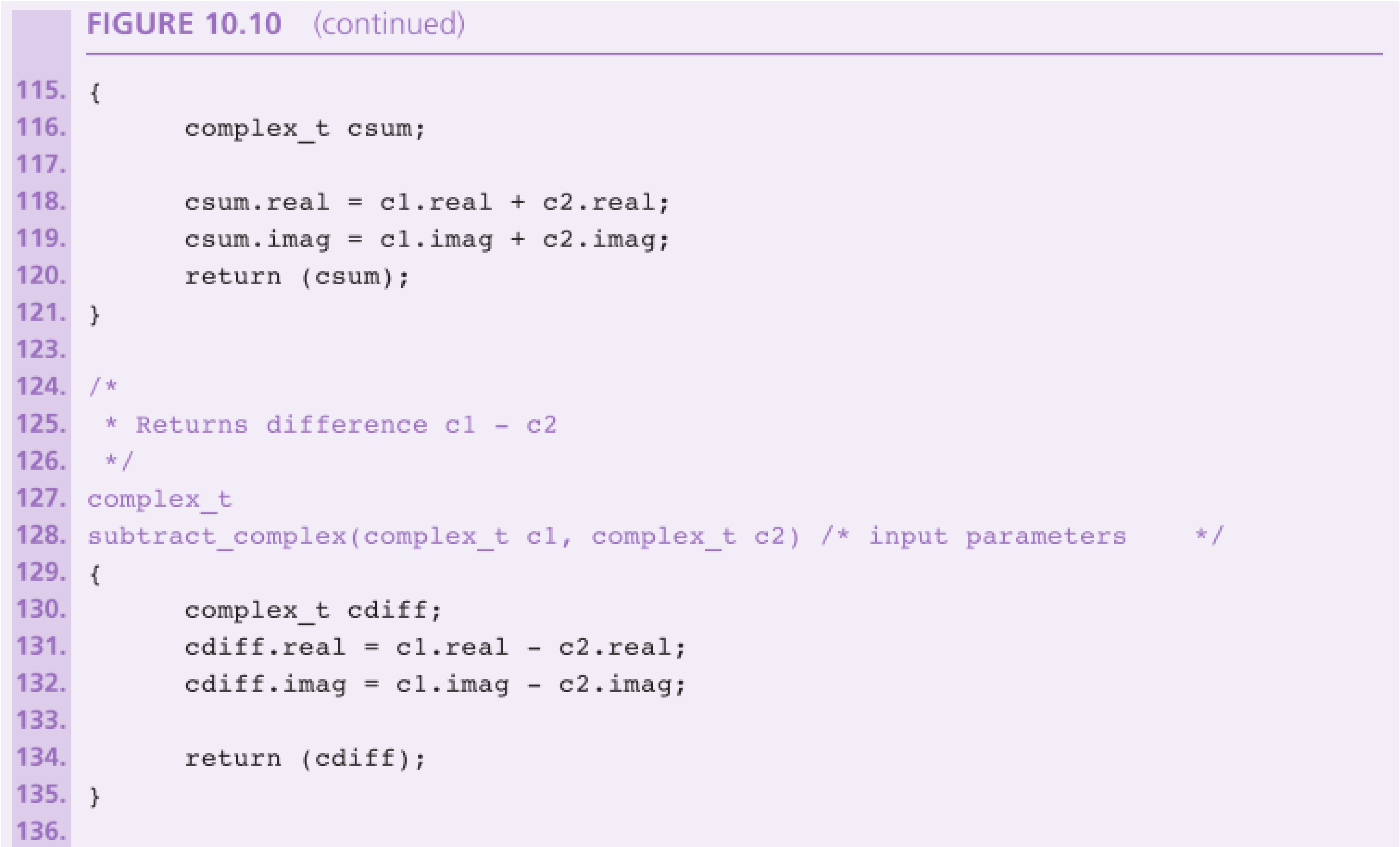
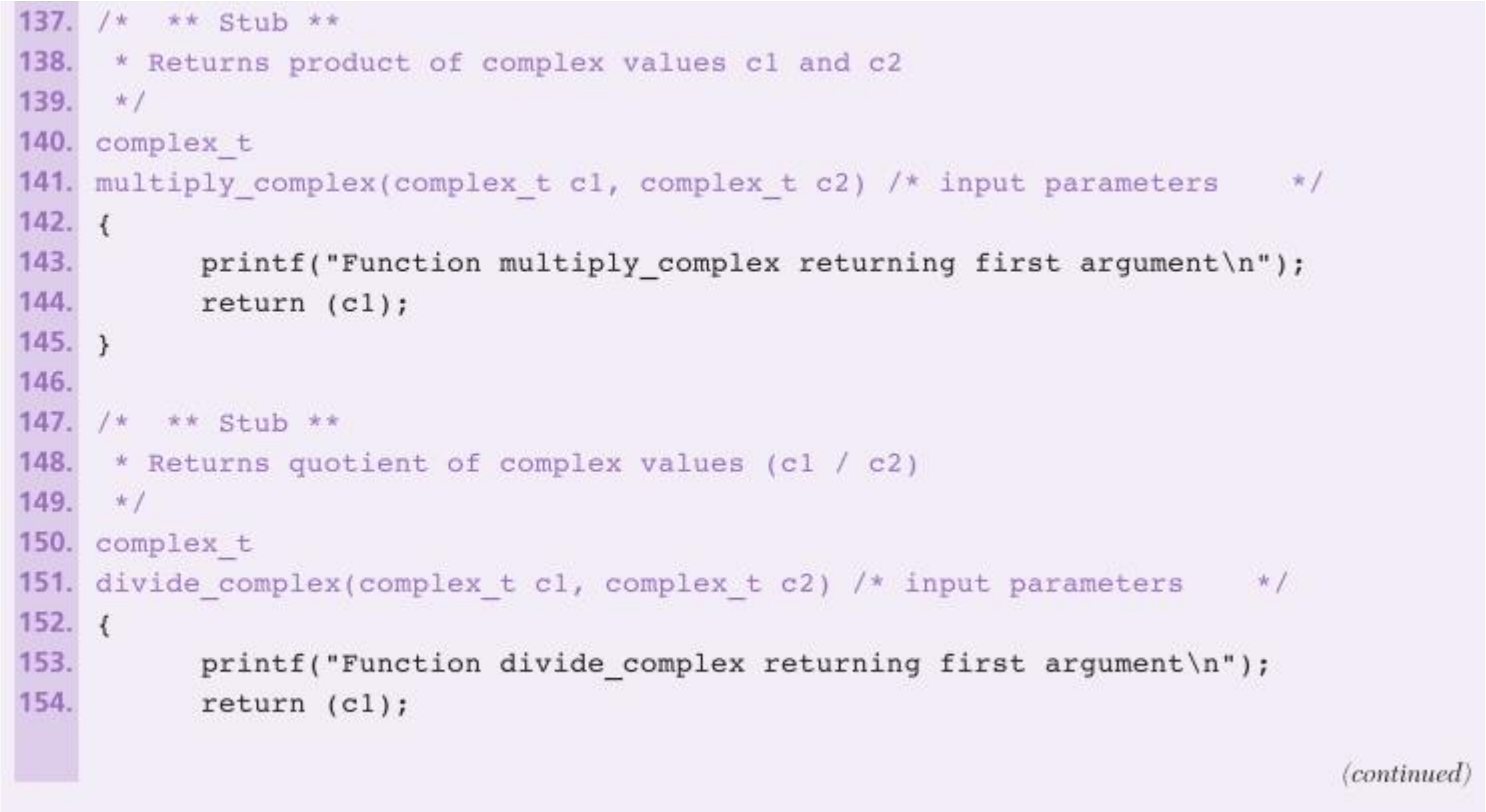


FIGURE 10.10 (continued)

```
{
      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)

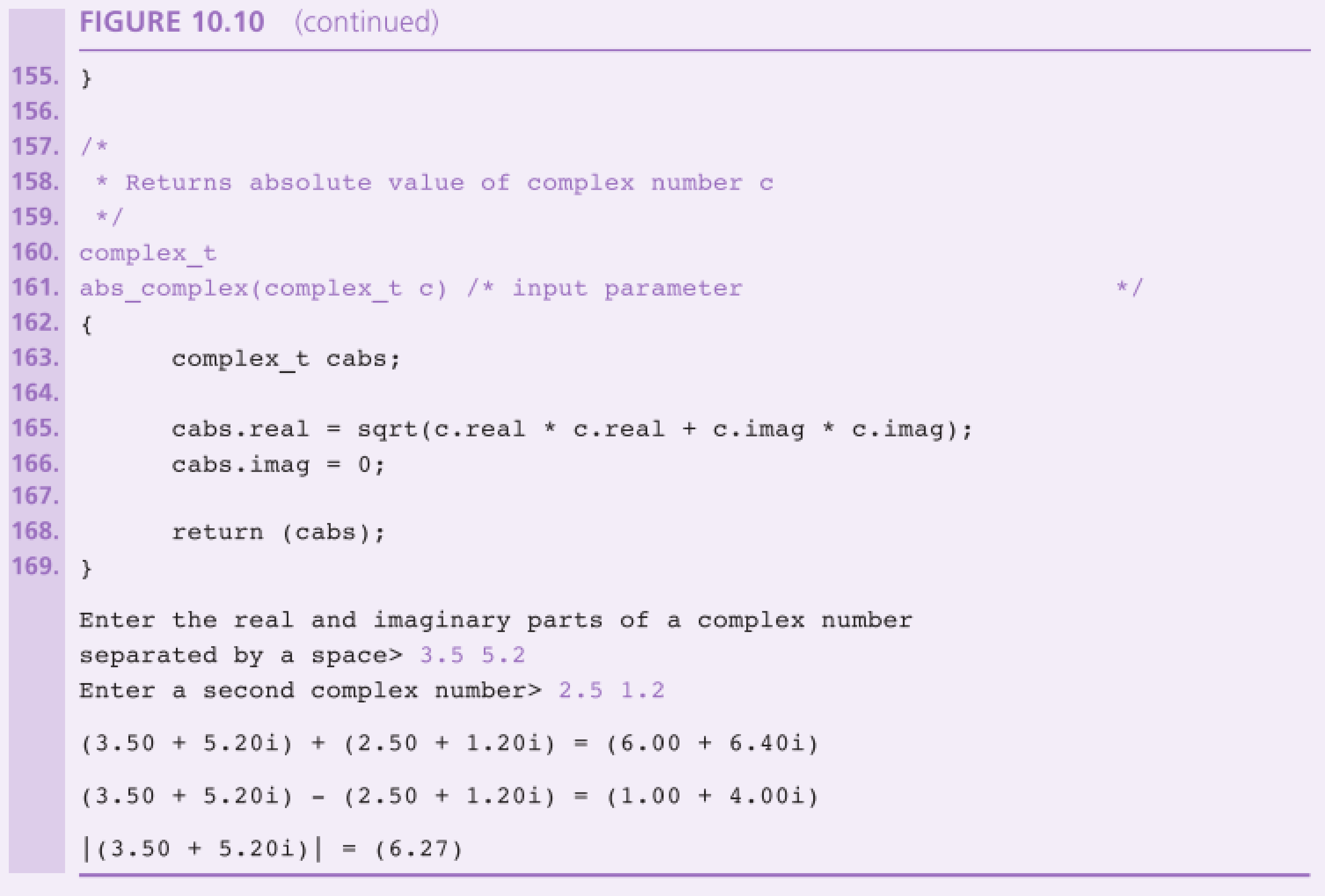


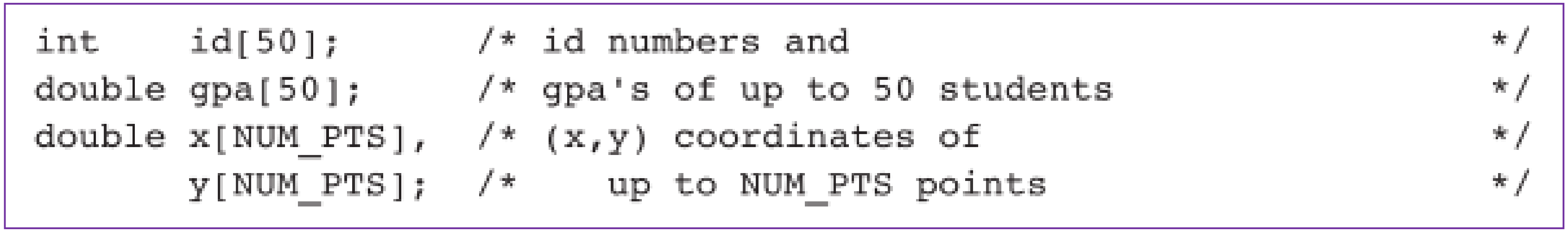
FIGURE 10.10 (continued)

```
}

/*
 * 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);
}
```

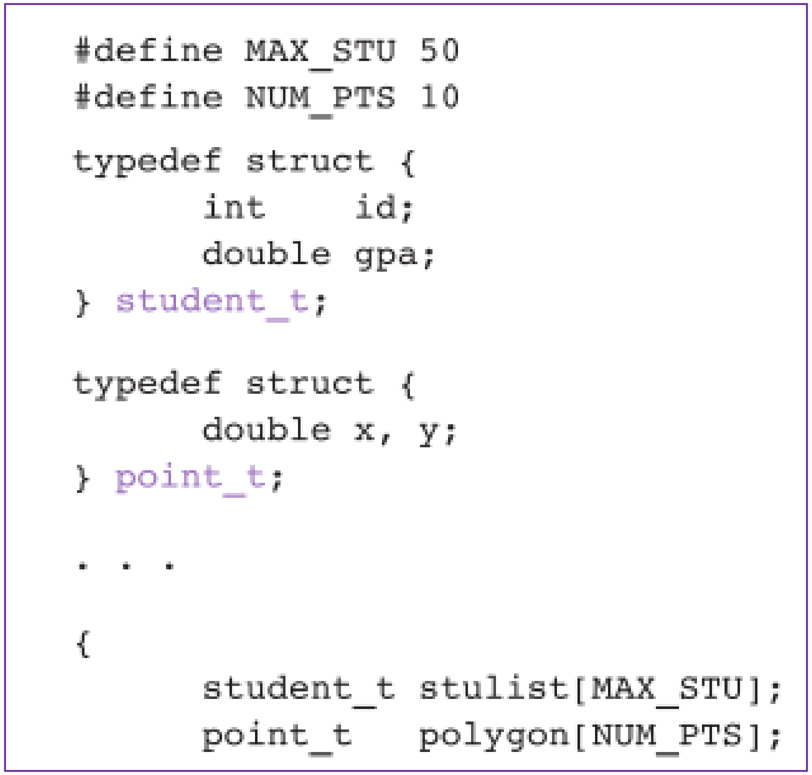
```
Enter the real and imaginary parts of a complex number
separated by a space> 3.5 5.2
Enter a second complex number> 2.5 1.2

(3.50 + 5.20i) + (2.50 + 1.20i) = (6.00 + 6.40i)

(3.50 + 5.20i) - (2.50 + 1.20i) = (1.00 + 4.00i)

|(3.50 + 5.20i)| = (6.27)
```

Parallel Arrays and Arrays of Structures

- A natural organization of parallel arrays with data that contain items of different types is to group the data into a structure whose type we define.



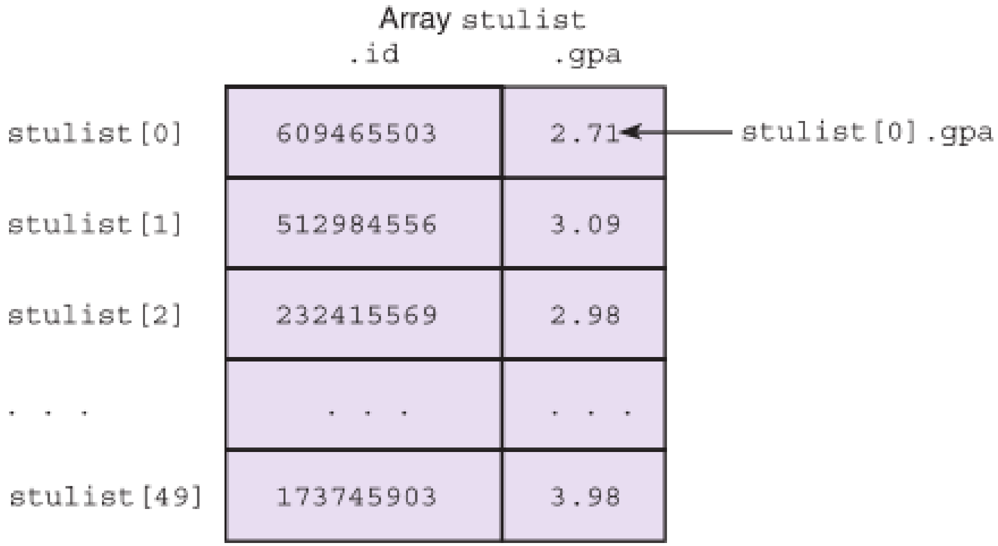
```
int    id[50];       /* id numbers and                      */
double gpa[50];      /* gpa's of up to 50 students          */
double x[NUM_PTS],   /* (x,y) coordinates of                */
       y[NUM_PTS];   /*    up to NUM_PTS points             */
```



```
#define MAX_STU 50
#define NUM_PTS 10

typedef struct {
      int    id;
      double gpa;
} student_t;

typedef struct {
      double x, y;
} point_t;

. . .

{
      student_t stulist[MAX_STU];
      point_t   polygon[NUM_PTS];
```

FIGURE 10.11

An Array of Structures

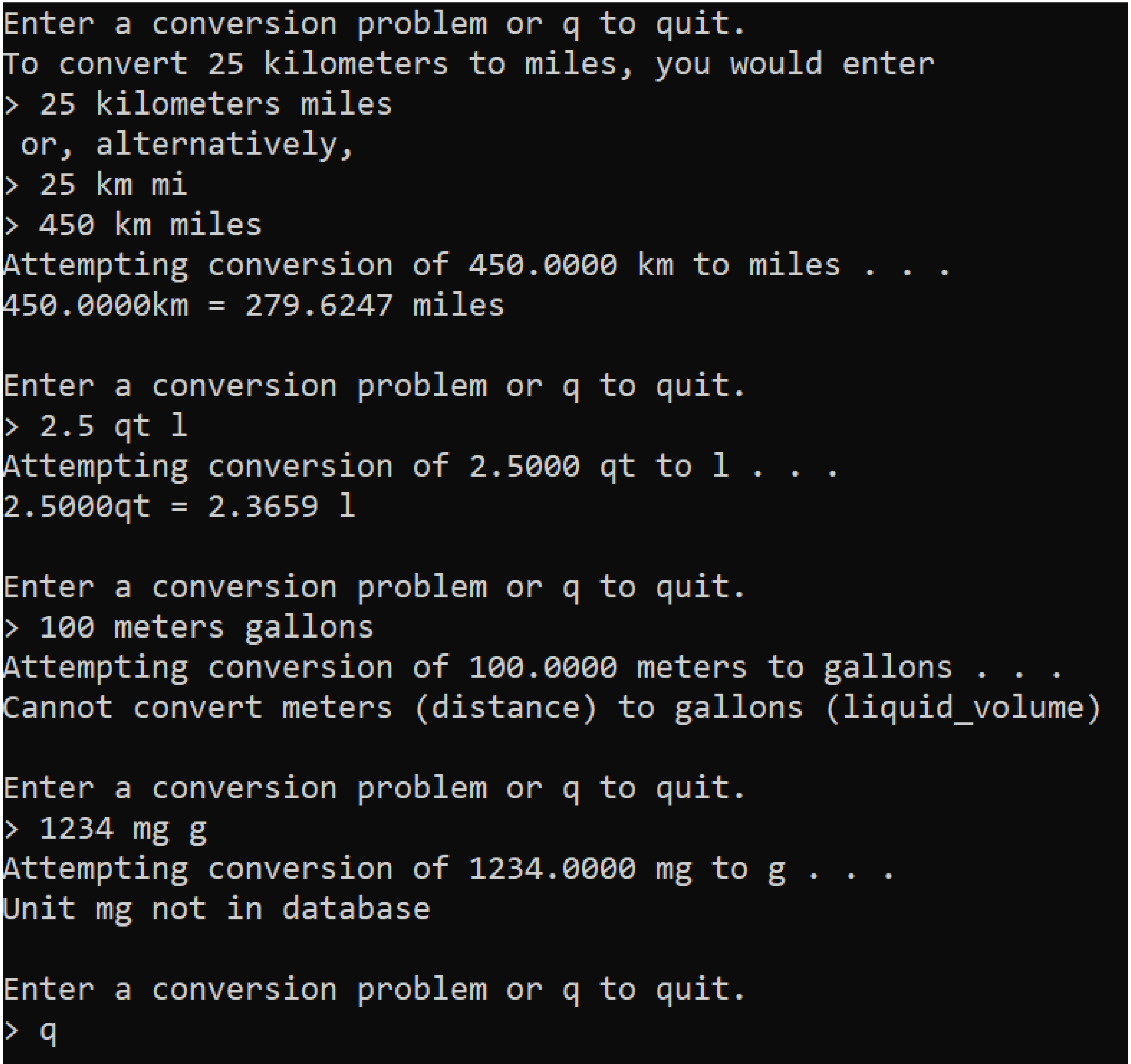


Array stulist

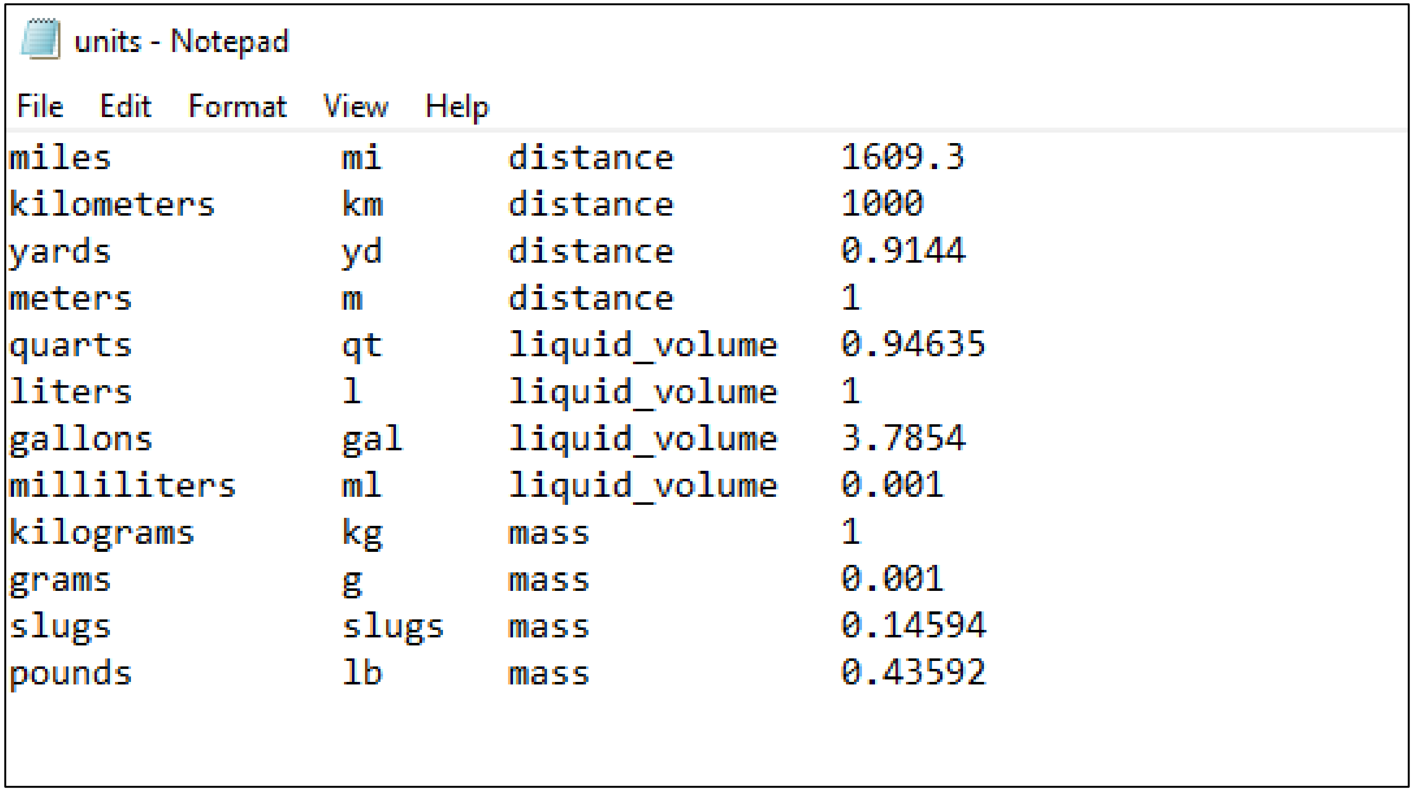
	.id	.gpa
stulist[0]	609465503	2.71
stulist[1]	512984556	3.09
stulist[2]	232415569	2.98
. . .	. . .	. . .
stulist[49]	173745903	3.98

```
for(int  i = 0; i < nrSt; i++)
   scan_student(&stulist[i]);
```

HOA 2: Universal Measurement Conversion



```
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
```



```
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
pounds          lb      mass            0.43592
```

Algorithm

```
1. Load units of measurement database.
2. Get value to convert and old and new unit names.
3. Repeat until data format error encountered.
    4. Search for old units in database.
    5. Search for new units in database.
    6. If conversion is impossible
        7. Issue appropriate error message.
   else
        8. Compute and display conversion.
    9. Get value to convert and old and new unit names.
```

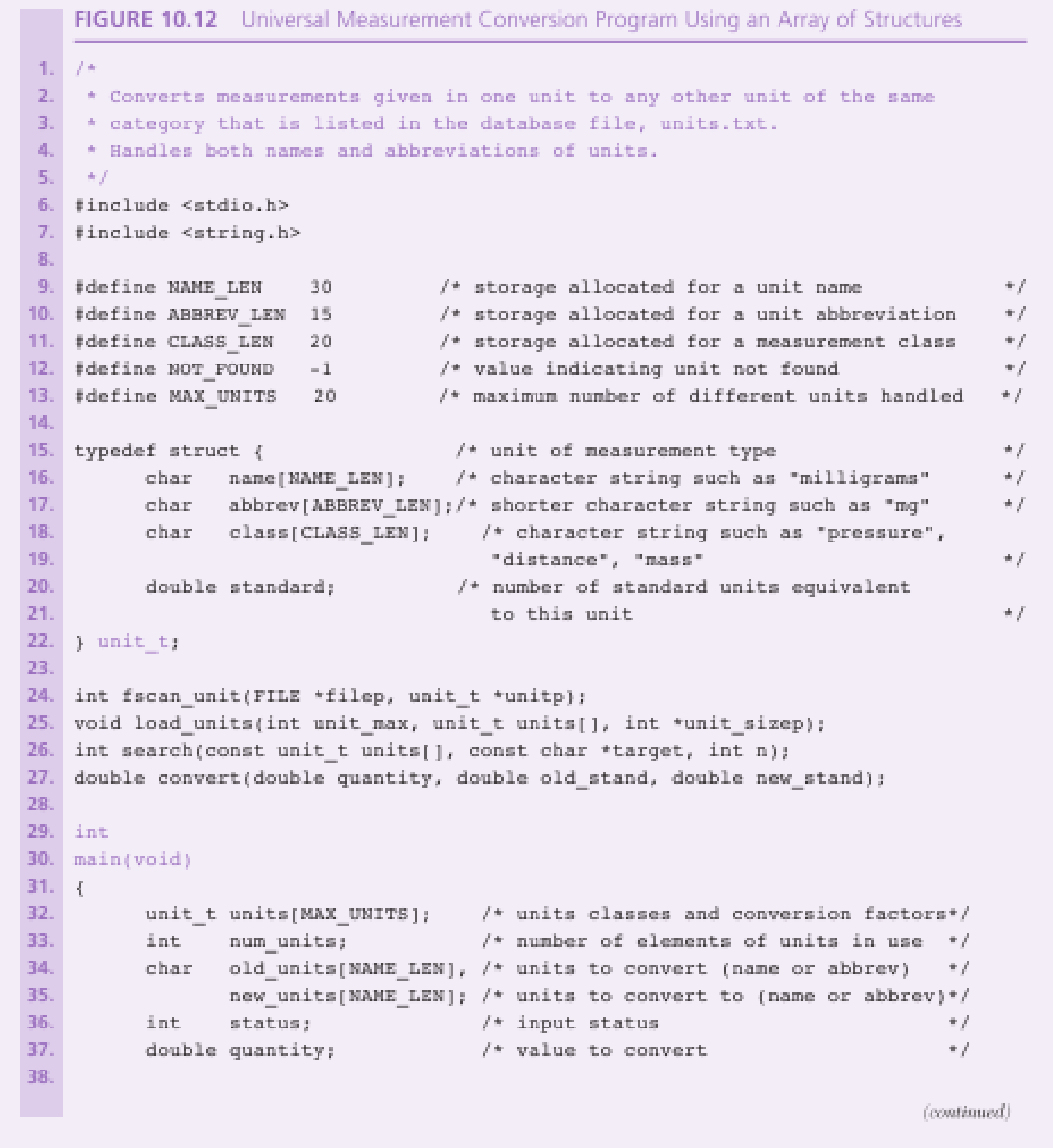


FIGURE 10.12 Universal Measurement Conversion Program Using an Array of Structures

```
/*
 * Converts measurements given in one unit to any other unit of the same
 * category that is listed in the database file, units.txt.
 * 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);

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                    */

```

(continued)

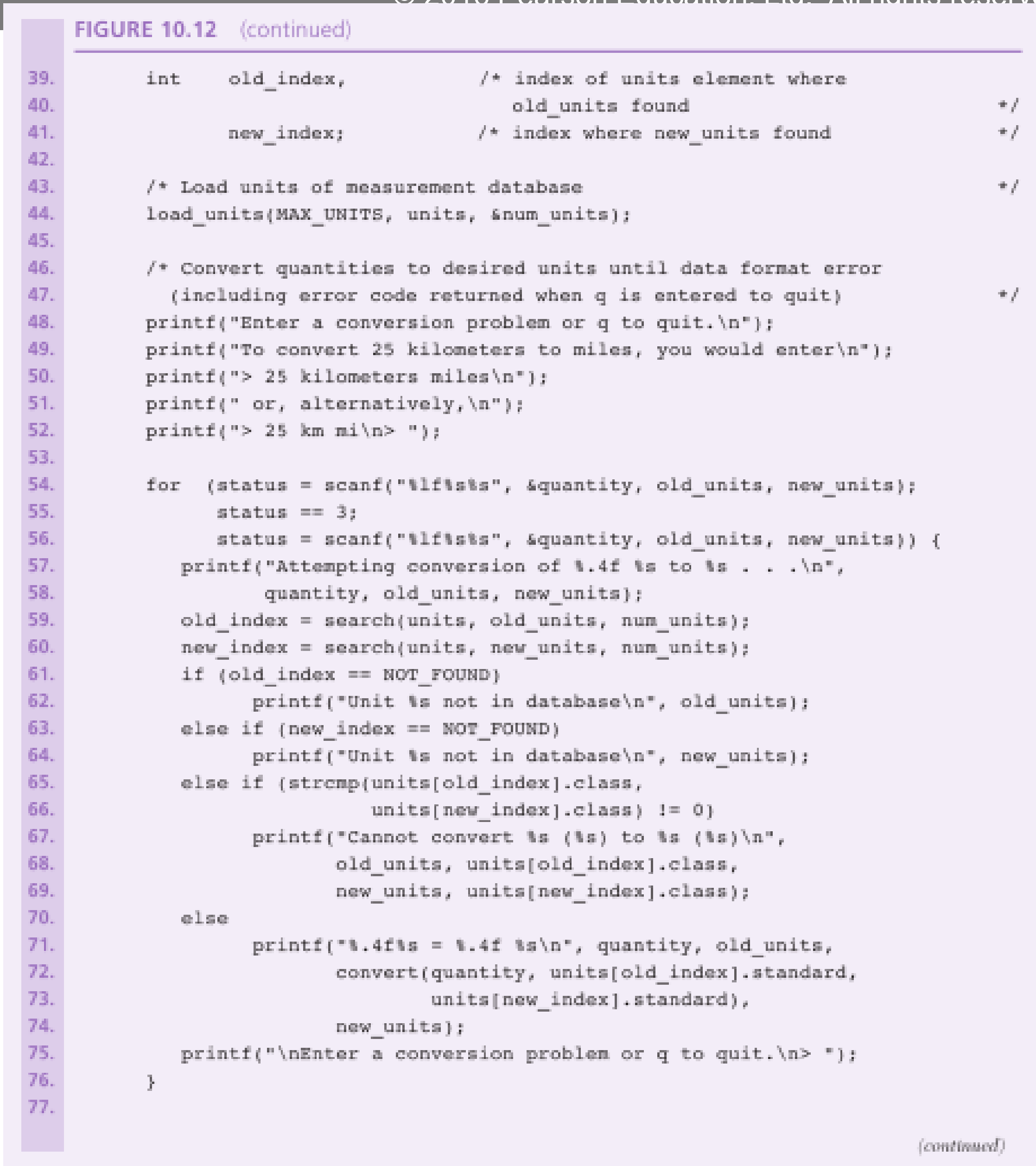


FIGURE 10.12 (continued)

```
    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> ");
    }
```

(continued)

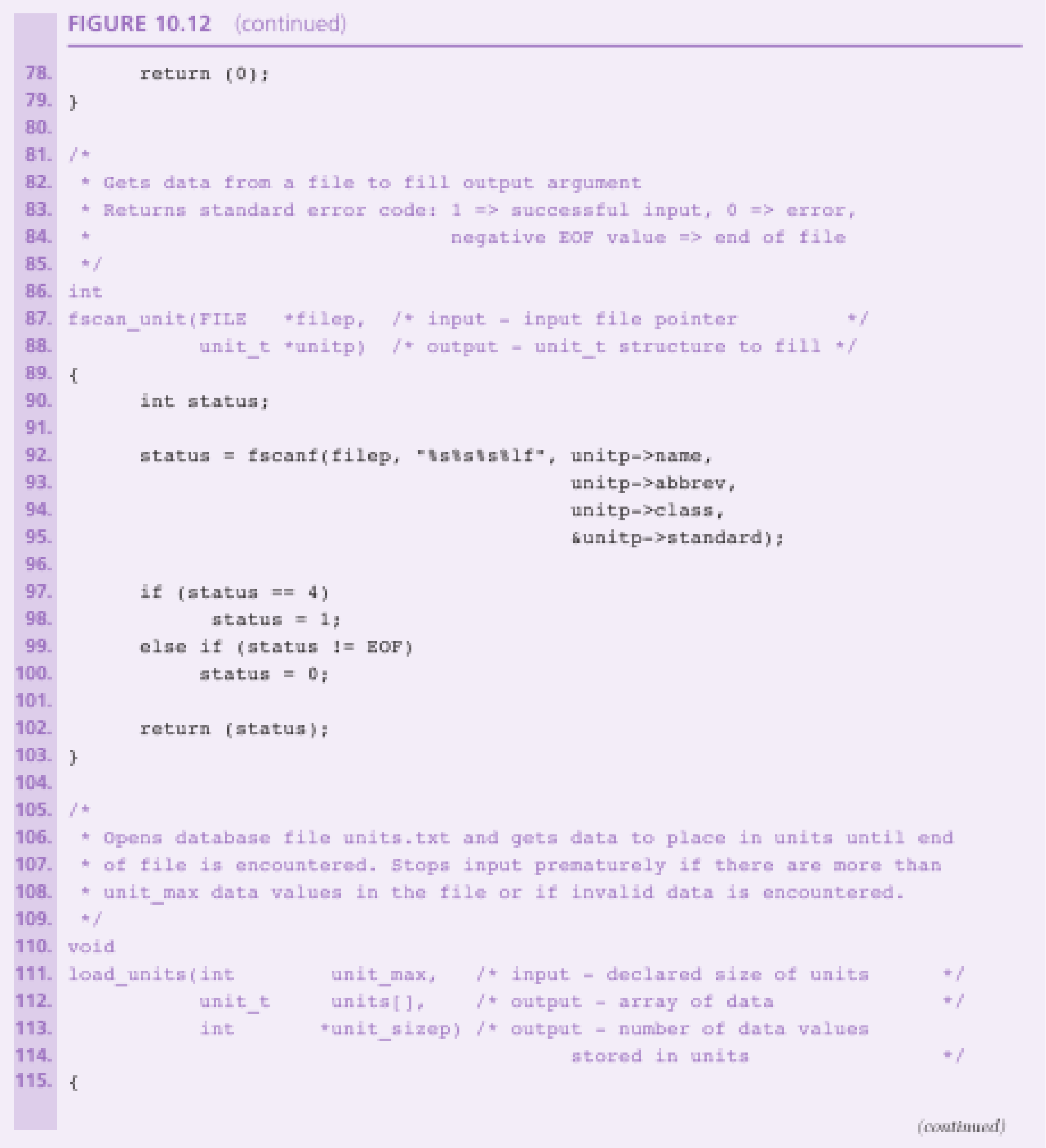


FIGURE 10.12 (continued)

```
      return (0);
}

/*
 * 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.txt 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                */
{
```

(continued)

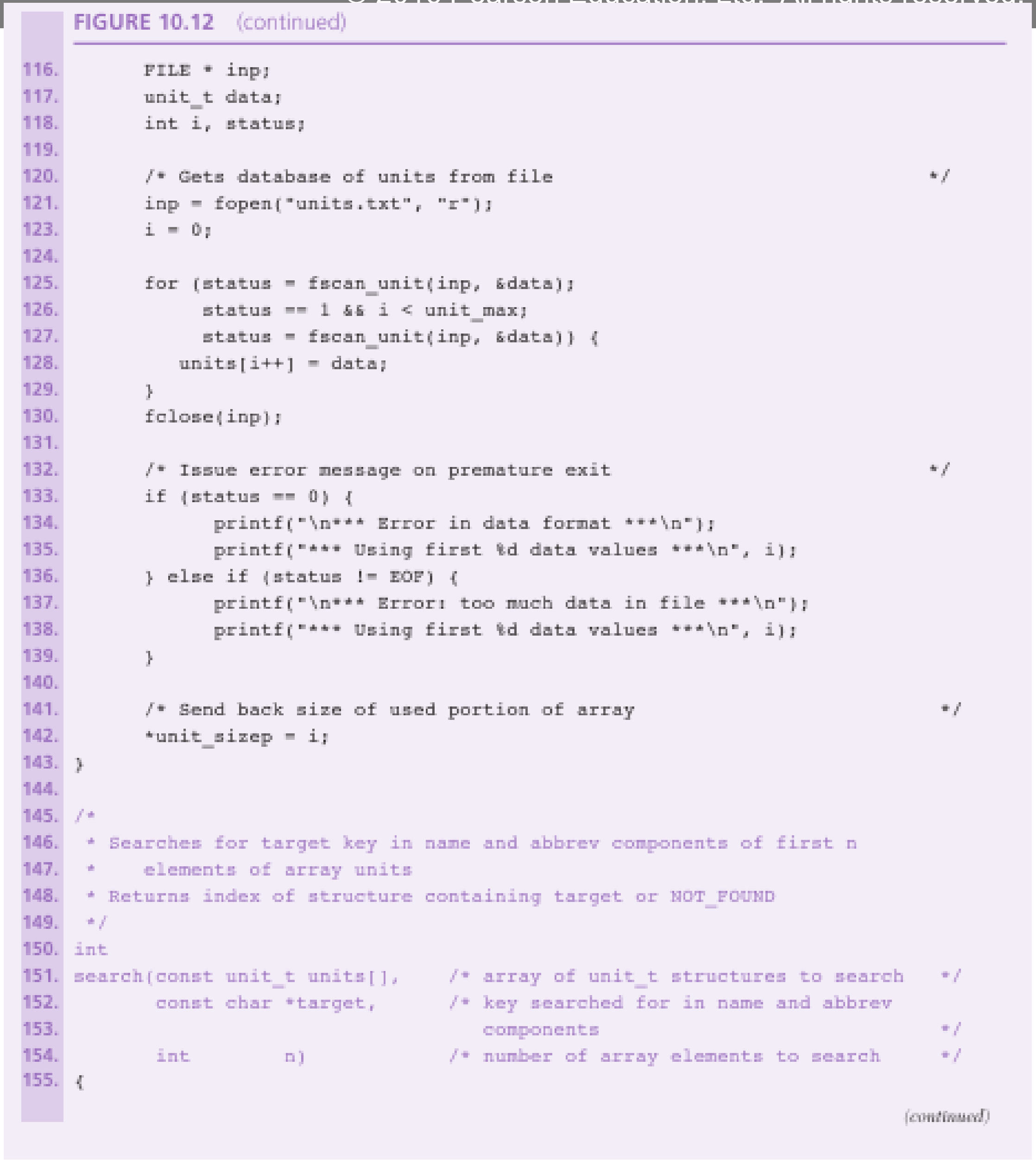


FIGURE 10.12 (continued)

```
      FILE * inp;
      unit_t data;
      int i, status;

      /* Gets database of units from file                                */
      inp = fopen("units.txt", "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     */
{
```

(continued)

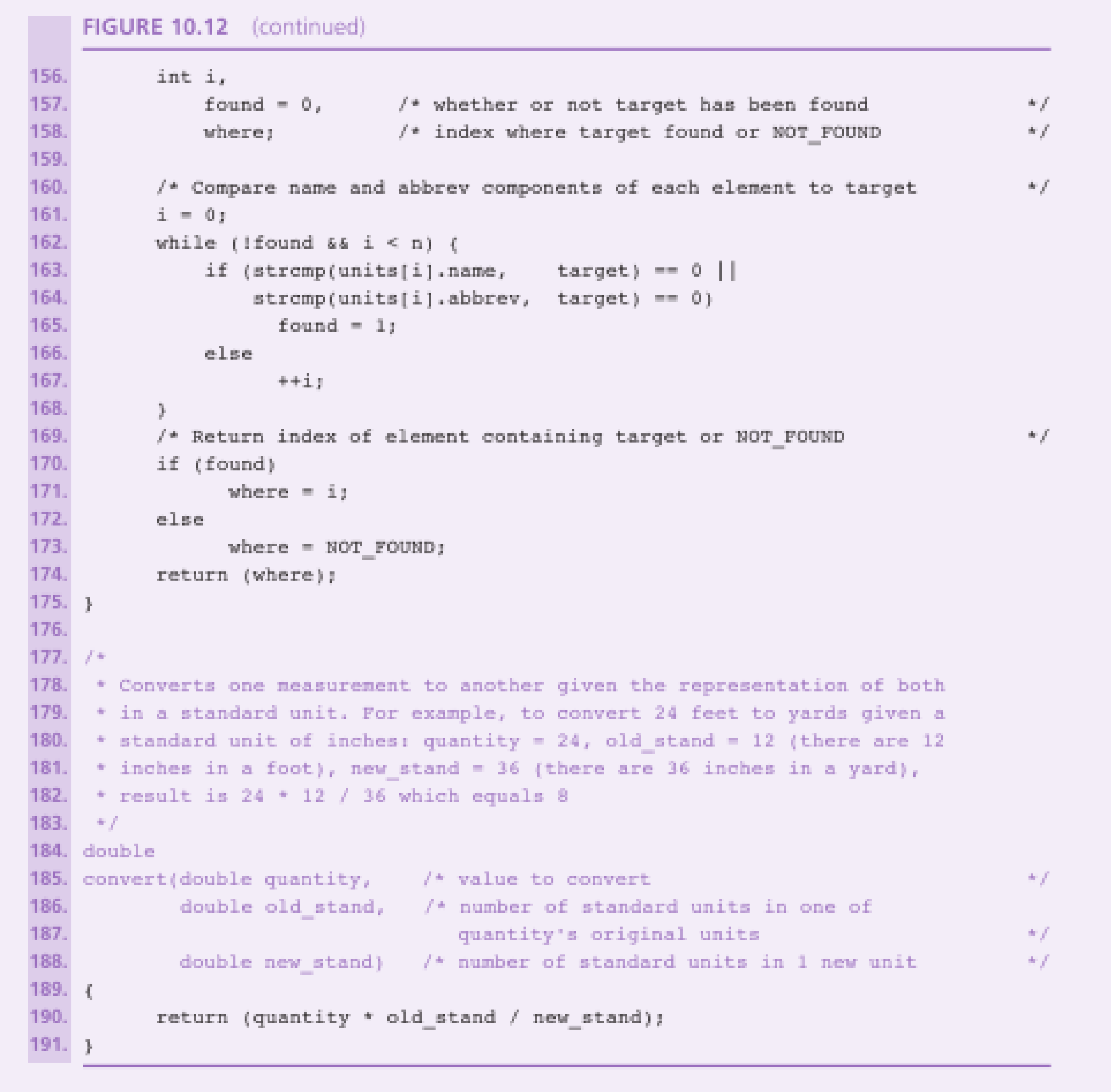


FIGURE 10.12 (continued)

```
      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);
}
```

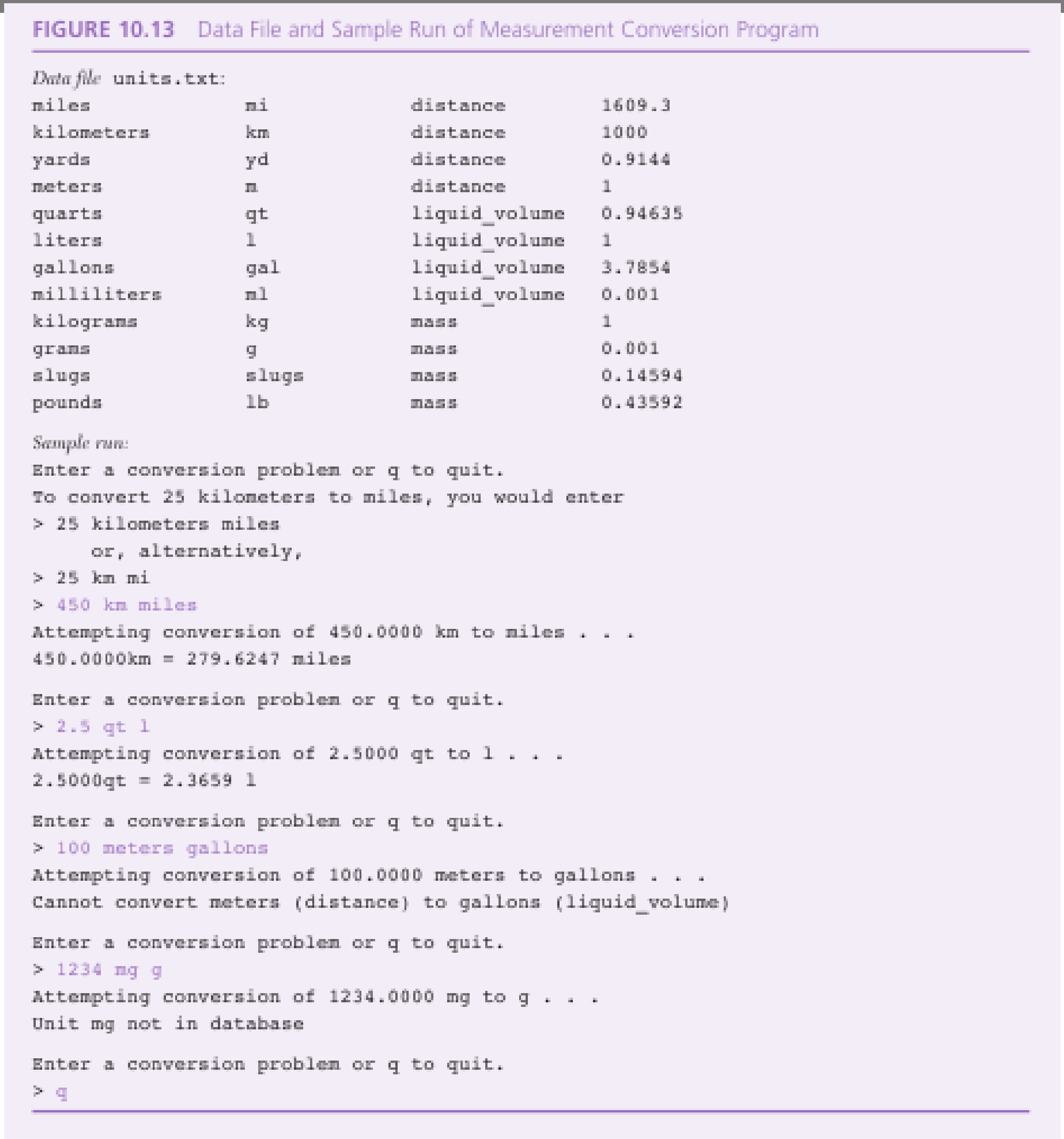


FIGURE 10.13 Data File and Sample Run of Measurement Conversion Program

Data file units.txt:

```
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
pounds             lb            mass            0.43592
```

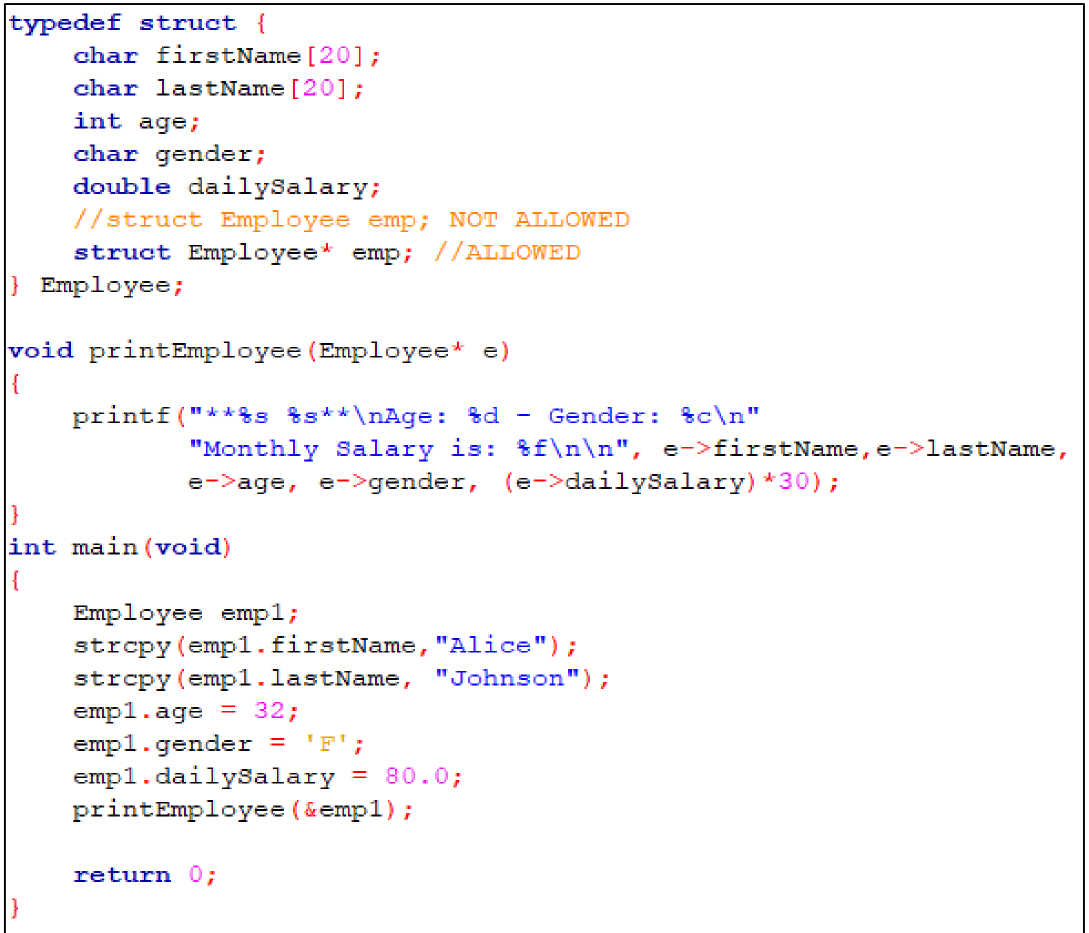
Sample run:

```
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
```

Self-Referential Structures

- A structure containing a member that is a pointer to the same structure type.

Where to use?

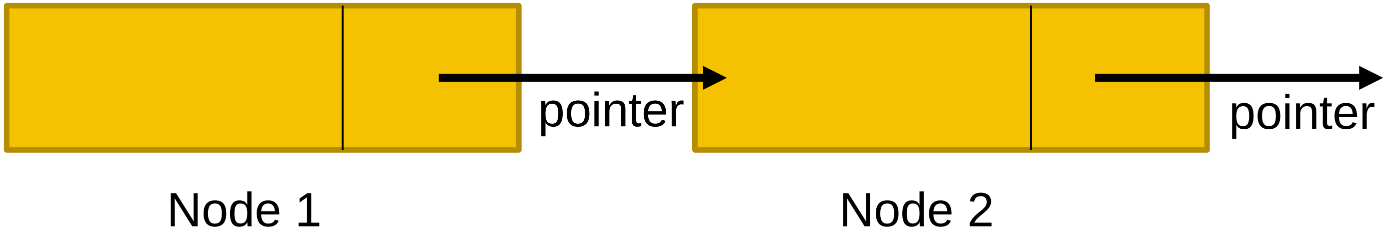


```
typedef struct {
    char firstName[20];
    char lastName[20];
    int age;
    char gender;
    double dailySalary;
    //struct Employee emp; NOT ALLOWED
    struct Employee* emp; //ALLOWED
} Employee;

void printEmployee(Employee* e)
{
    printf("**%s %s**\nAge: %d - Gender: %c\n"
           "Monthly Salary is: %f\n\n", e->firstName,e->lastName,
           e->age, e->gender, (e->dailySalary)*30);
}
int main(void)
{
    Employee emp1;
    strcpy(emp1.firstName,"Alice");
    strcpy(emp1.lastName, "Johnson");
    emp1.age = 32;
    emp1.gender = 'F';
    emp1.dailySalary = 80.0;
    printEmployee(&emp1);

    return 0;
}
```

Self-Referential Structures

- E.g. Linked Lists

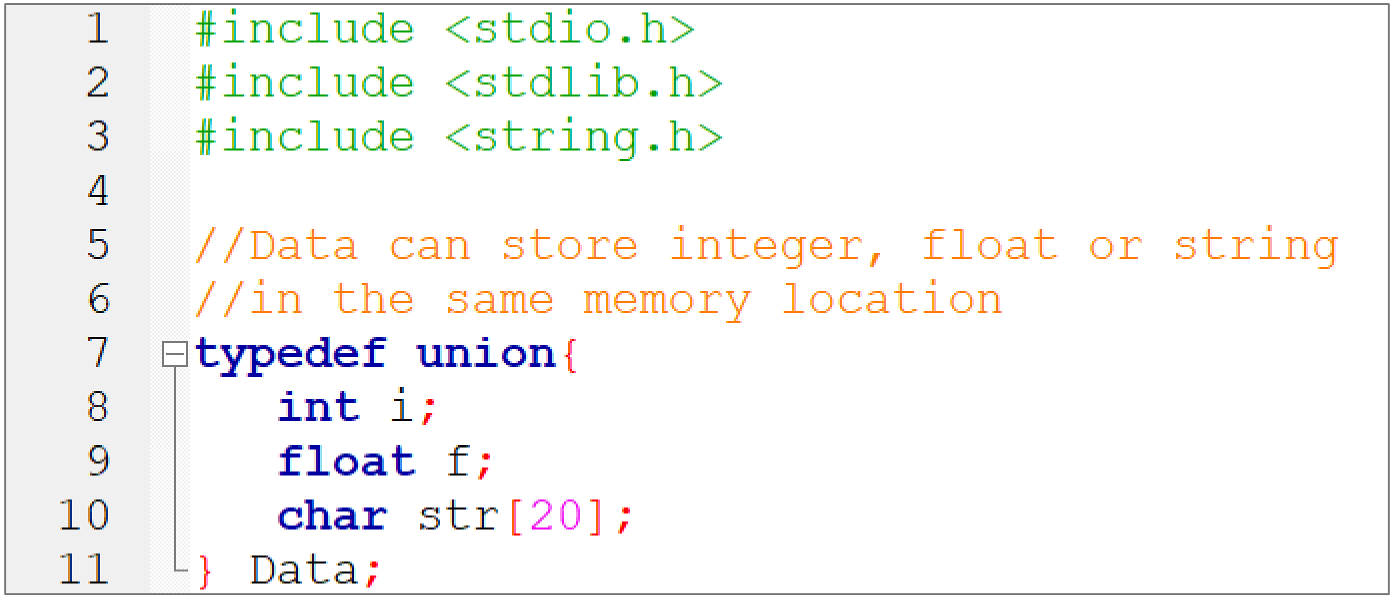
```
struct node_type {
        int data;
        struct node_type *next;
};
```



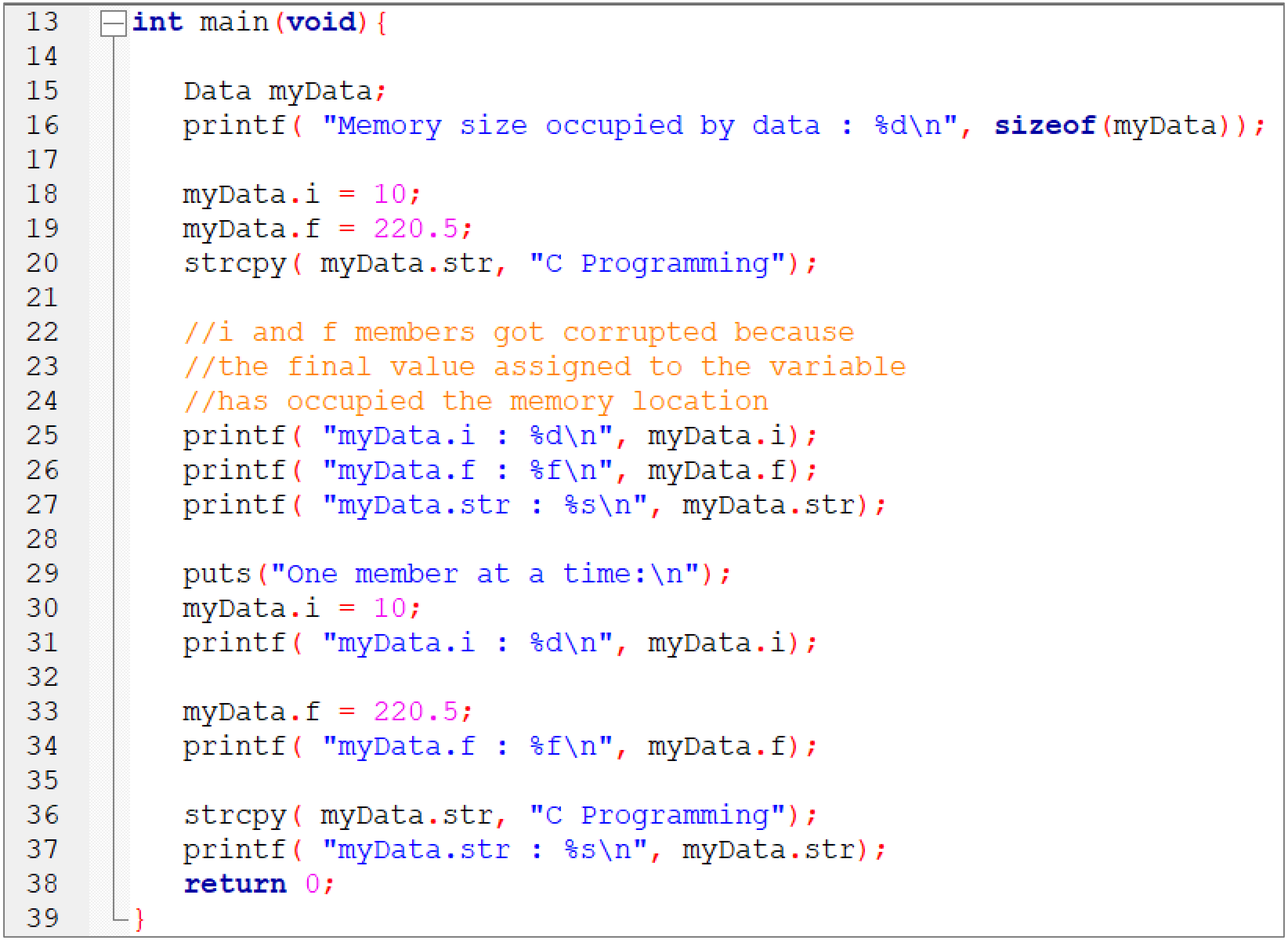
Union Types

- union
  - a data structure that overlays components in memory, allowing one chunk of memory to be interpreted in multiple ways
  - **allows to store different data types in the same memory location**
  - space is reserved at least as large as the largest member
  - may be defined with many members, but only one member can contain a value at any given time

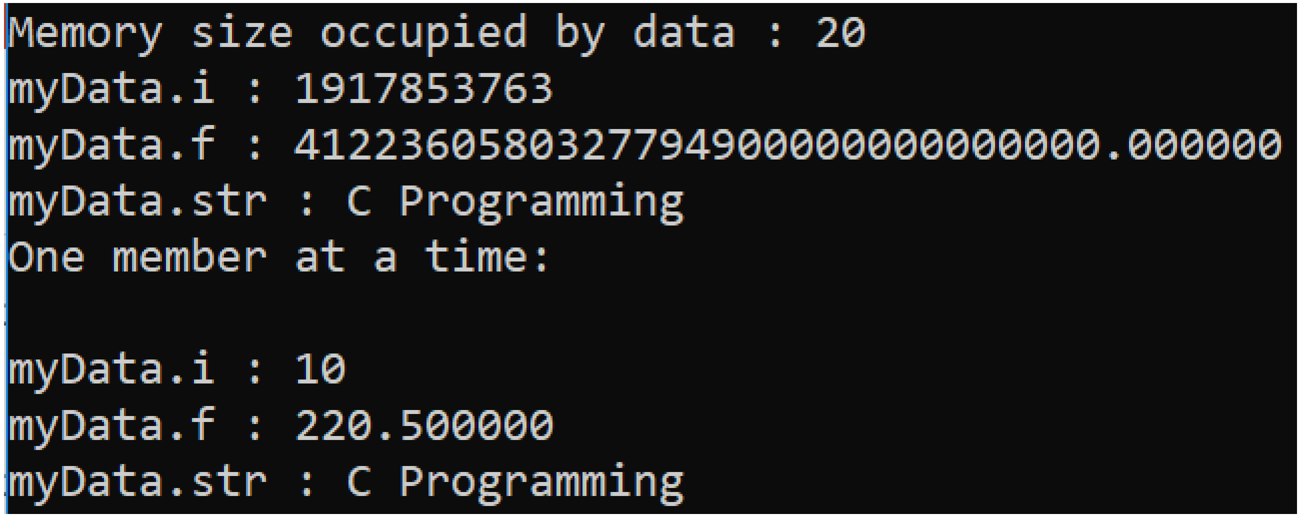
Union Types



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

//Data can store integer, float or string
//in the same memory location
typedef union{
    int i;
    float f;
    char str[20];
} Data;
```



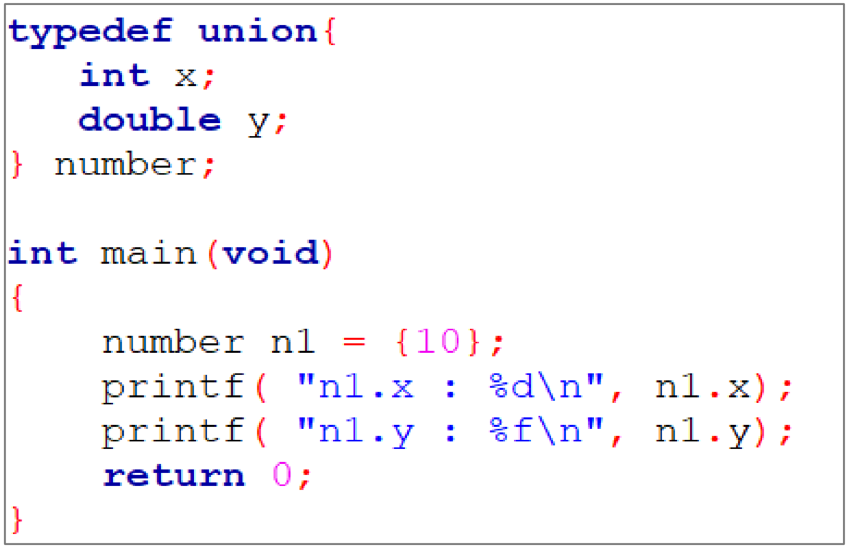
```c
int main(void){

    Data myData;
    printf( "Memory size occupied by data : %d\n", sizeof(myData));

    myData.i = 10;
    myData.f = 220.5;
    strcpy( myData.str, "C Programming");

    //i and f members got corrupted because
    //the final value assigned to the variable
    //has occupied the memory location
    printf( "myData.i : %d\n", myData.i);
    printf( "myData.f : %f\n", myData.f);
    printf( "myData.str : %s\n", myData.str);

    puts("One member at a time:\n");
    myData.i = 10;
    printf( "myData.i : %d\n", myData.i);

    myData.f = 220.5;
    printf( "myData.f : %f\n", myData.f);

    strcpy( myData.str, "C Programming");
    printf( "myData.str : %s\n", myData.str);
    return 0;
}
```



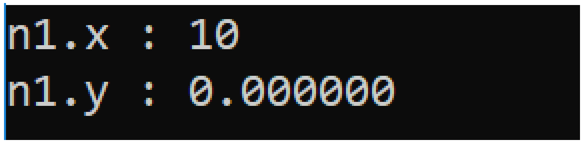
```
Memory size occupied by data : 20
myData.i : 1917853763
myData.f : 4122360580327794900000000000000.000000
myData.str : C Programming
One member at a time:

myData.i : 10
myData.f : 220.500000
myData.str : C Programming
```

Initialization at Declaration Time

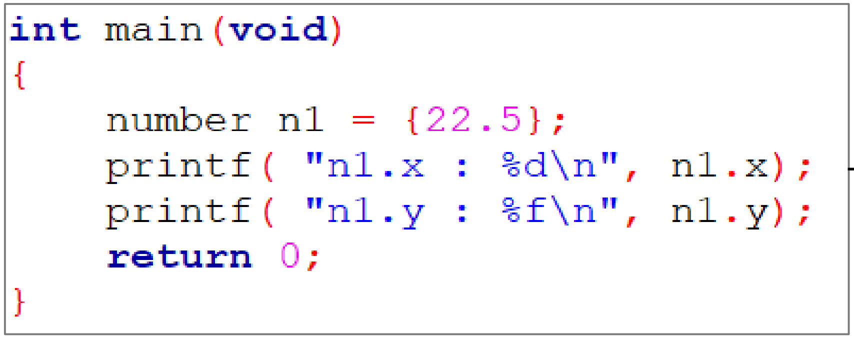
- Initialization with a value of the same type of the first member is allowed.



```
typedef union{
   int x;
   double y;
} number;

int main(void)
{
    number n1 = {10};
    printf( "n1.x : %d\n", n1.x);
    printf( "n1.y : %f\n", n1.y);
    return 0;
}
```



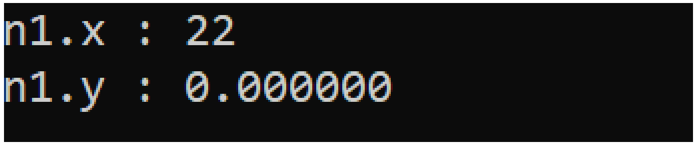
```
n1.x : 10
n1.y : 0.000000
```



```
int main(void)
{
    number n1 = {22.5};
    printf( "n1.x : %d\n", n1.x);
    printf( "n1.y : %f\n", n1.y);
    return 0;
}
```

?

Truncated to match the first member’s data type



```
n1.x : 22
n1.y : 0.000000
```

HOA 3

- Write a program to compute Area and Perimeter of variety of geometric figures
  1. Create a struct *circle_t* with members:
     - area, circumference, radius (:double)
  2. Create a struct *rectangle_t* with members:
     - area, perimeter, width, height(:double)
  3. Create a struct *square_t* with members:
     - area, perimeter, side(:double)
  4. Create a union *figure_data_t* so that the type of a geometric structure can be interpreted in different ways
  5. Create a struct *figure_t* with members:
     - shape (:char), fig (:figure_data_t)

HOA 3

- Write 4 functions:
  - void get_figure_dimensions(figure_t* object);
  - figure_t compute_area(figure_t object);
  - figure_t compute_perim(figure_t object);
  - void print_figure(figure_t object);

Wrap Up

- C permits the user to define a type composed of multiple named components.
- User-defined structure types can be used in most situations where build-in types are value.
- Structured values can be function arguments and function results and can be copied using the assignment operator.

Wrap Up

- Structure types are legitimate in declarations of variables, of structure components, and of arrays.
- Structure types play an important role in data abstraction. You create an abstract data type (ADT) by implementing all of the types necessary operations.
- In a union type, structure components are overlaid in memory.

References

1. Problem Solving & Program Design in C, Jeri R. Hanly & Elliot B. Koffman, Pearson 8. Edition, Global Edition
2. C How to Program, Paul Deitel, Harvey Deitel. Pearson 8th Edition, Global Edition.