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

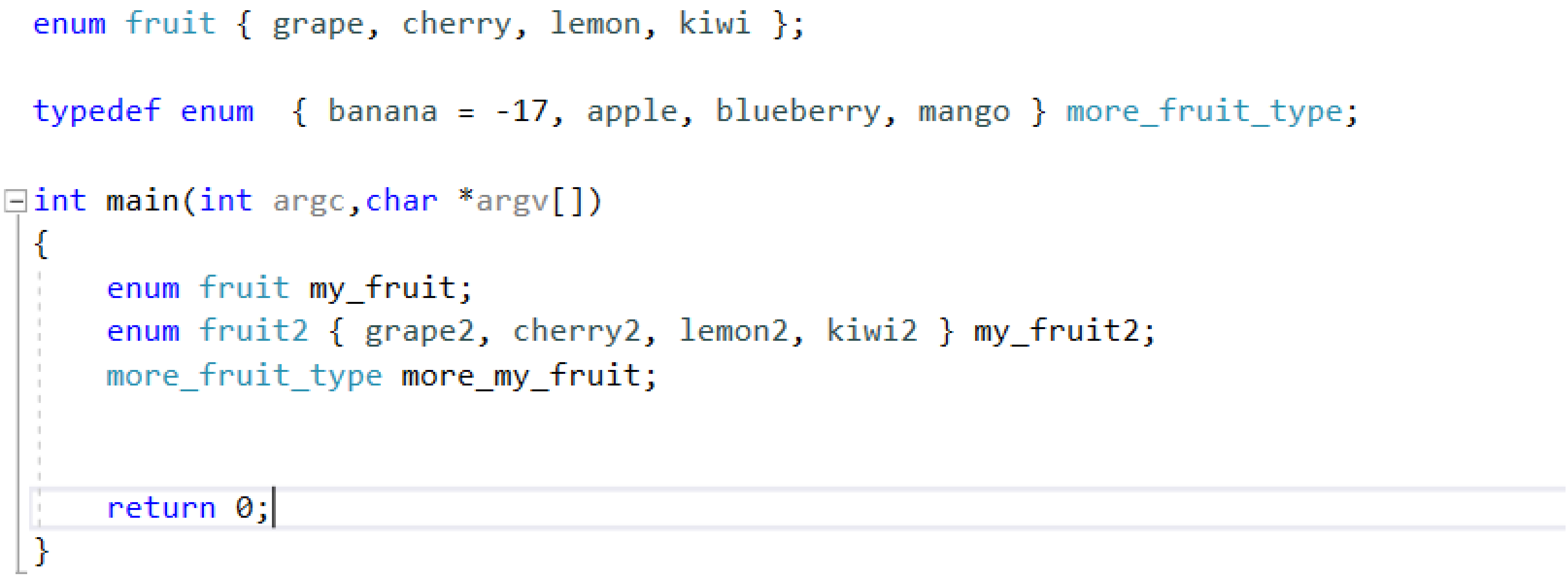
- To see how to create a struct data type for representing complex numbers and how to write functions that perform arithmetic operations on complex numbers
- To understand the relationship between parallel arrays and arrays of structured objects
- To learn about union data types and how they differ form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

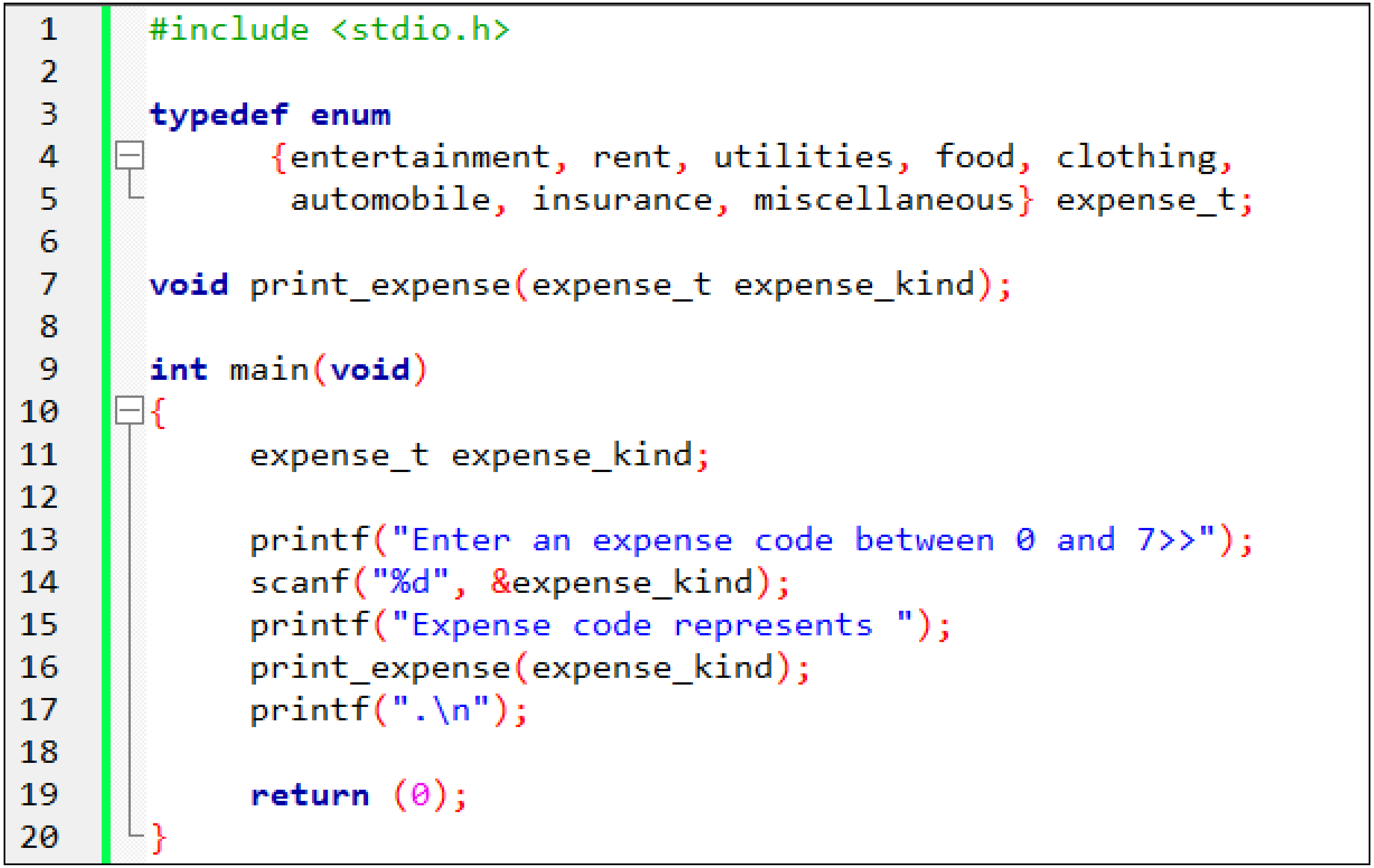
- After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char, for example.**.
  - BYTE b1, b2;

typedef vs. #define

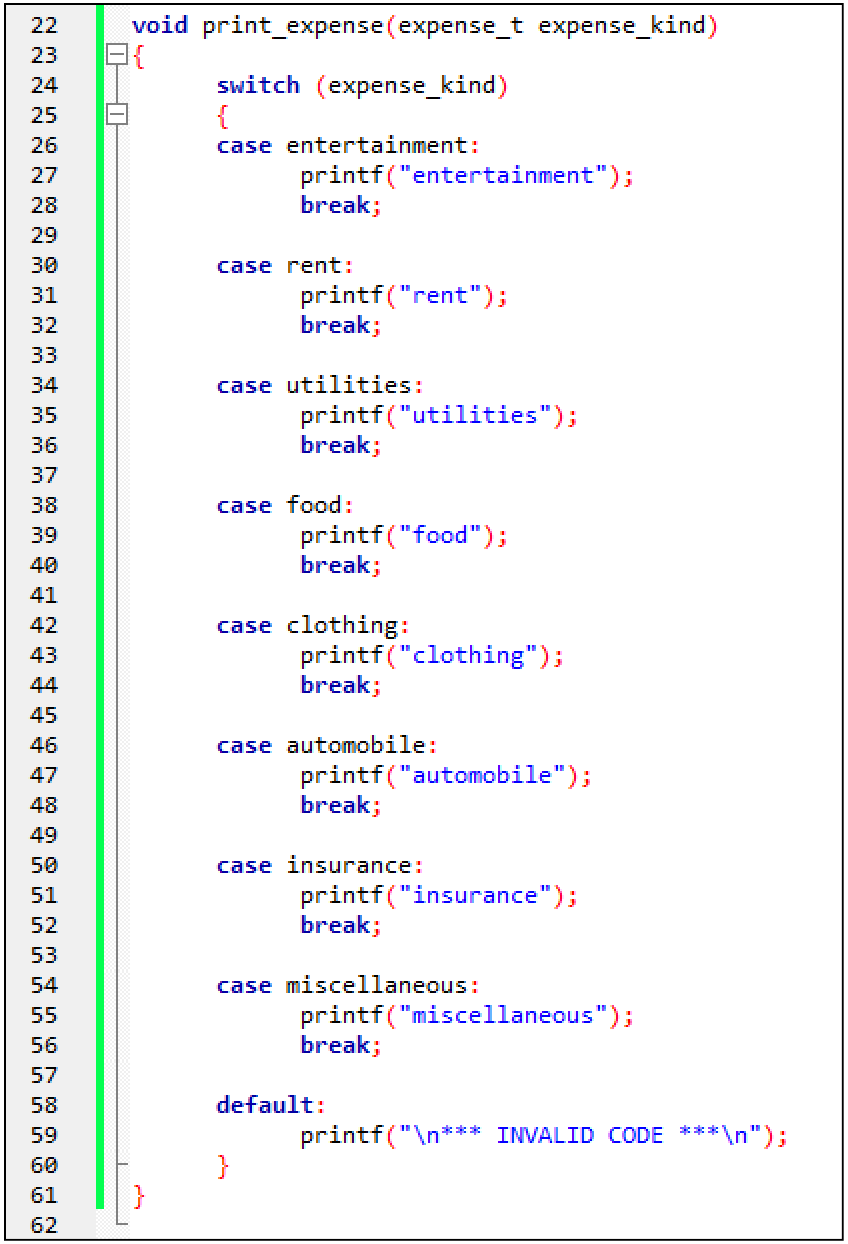
- **#define** is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences:

  - **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.

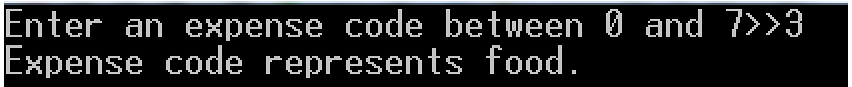
  - **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the pre-processor.



```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};
```

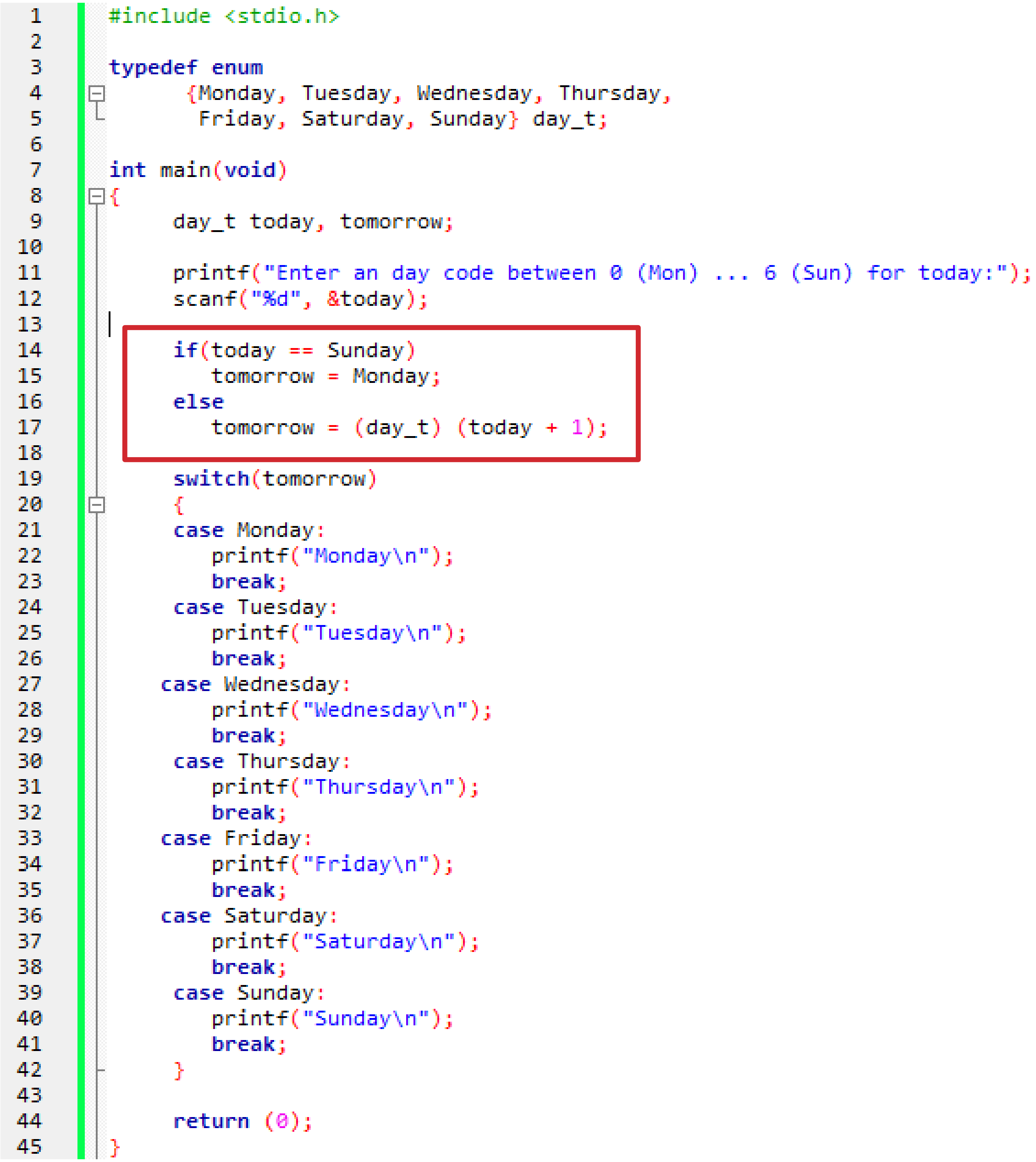
Enum Arithmetic

- 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! */

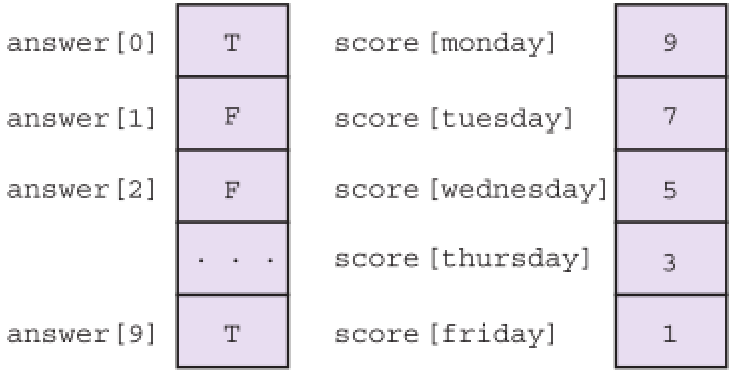


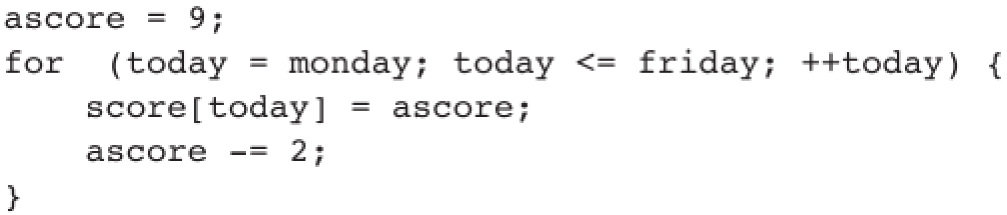
```c
#include <stdio.h>

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]





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

STRUCTURES

User-Defined Structure Types

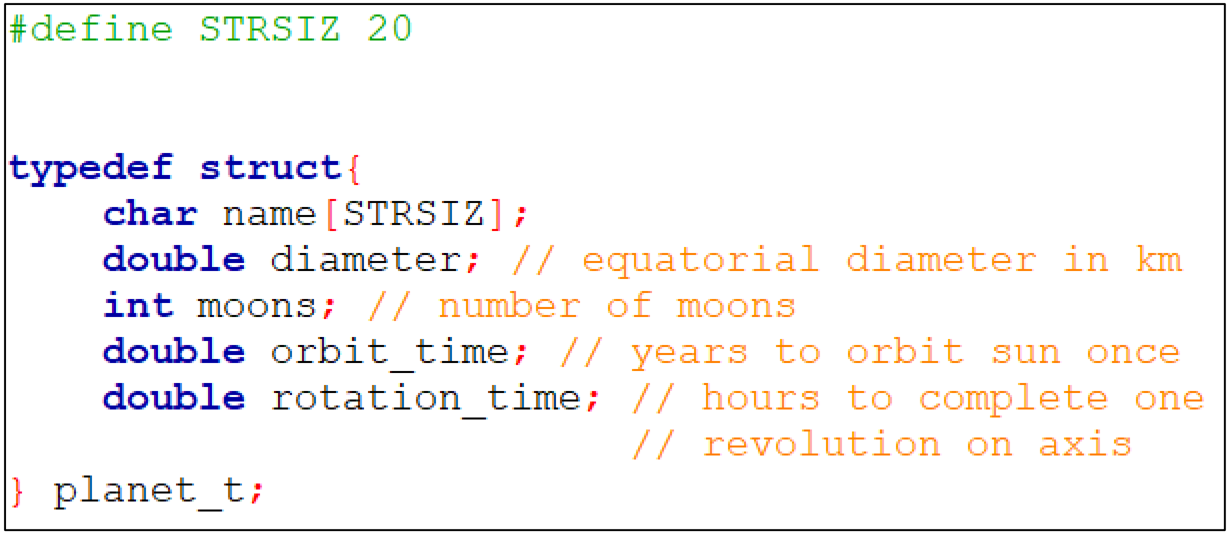
- record
  - a collection of information about one data object in a database
- structure type
  - a data type for a record composed of multiple components
- hierarchical structure
  - a structure containing components that are structures, e.g. array, struct

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.

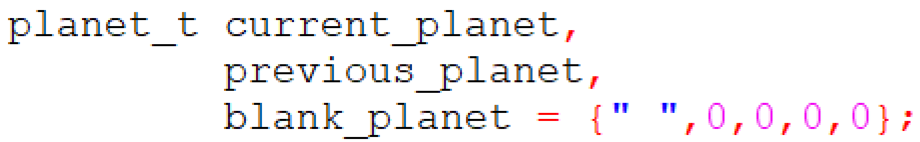
- Name: Jupiter
- Diameter: 142.800km
- Moons: 16
- Orbit time: 11.9 years
- Rotation time: 9.925 hours

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;
```

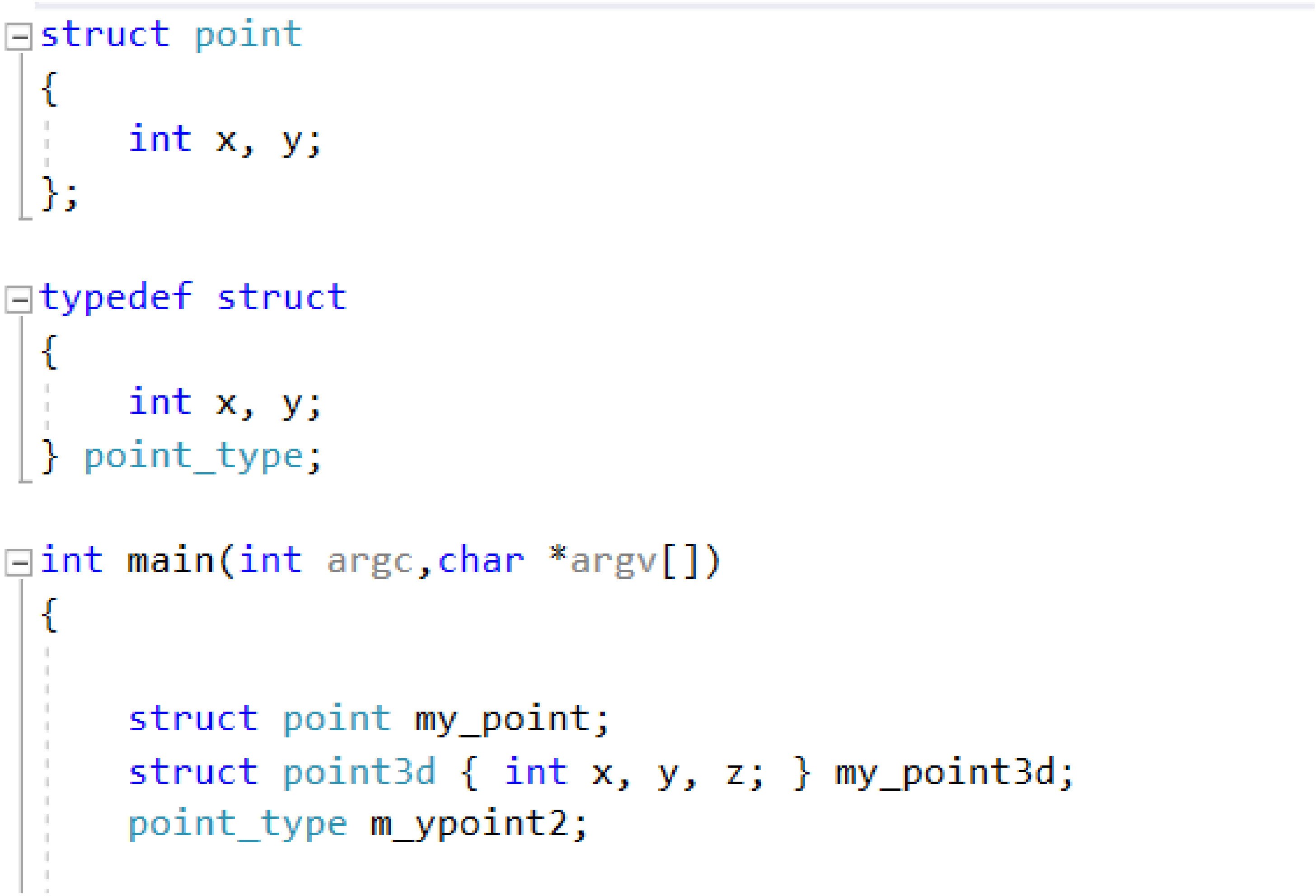
- This typedef definition itself allocates no memory. To allocate, declare a variable of this struct type:



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

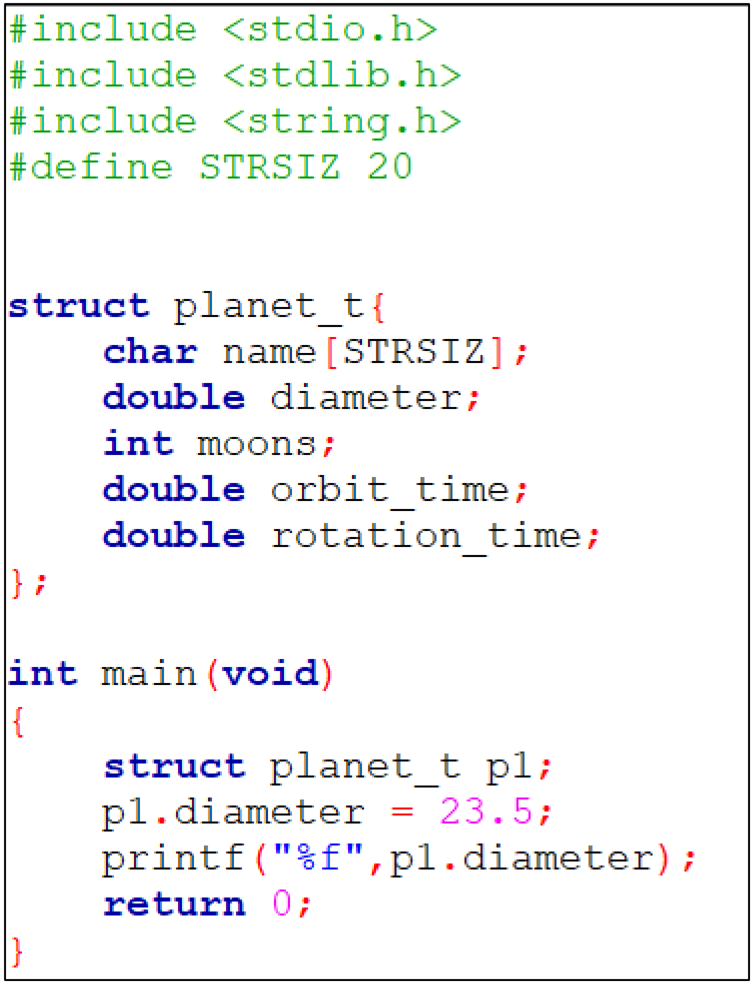
If there are fewer initializers in the list than members in the structure, the rest are automatically initialized to 0 or NULL.

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



```c
#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

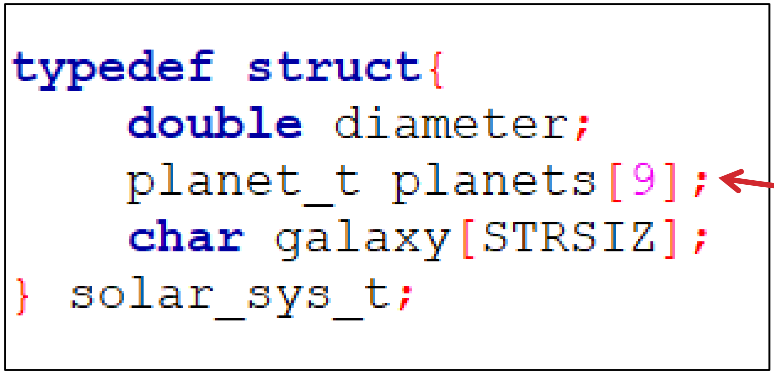
User-Defined Structure Types

Quick Check: Create a complex number structure.

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

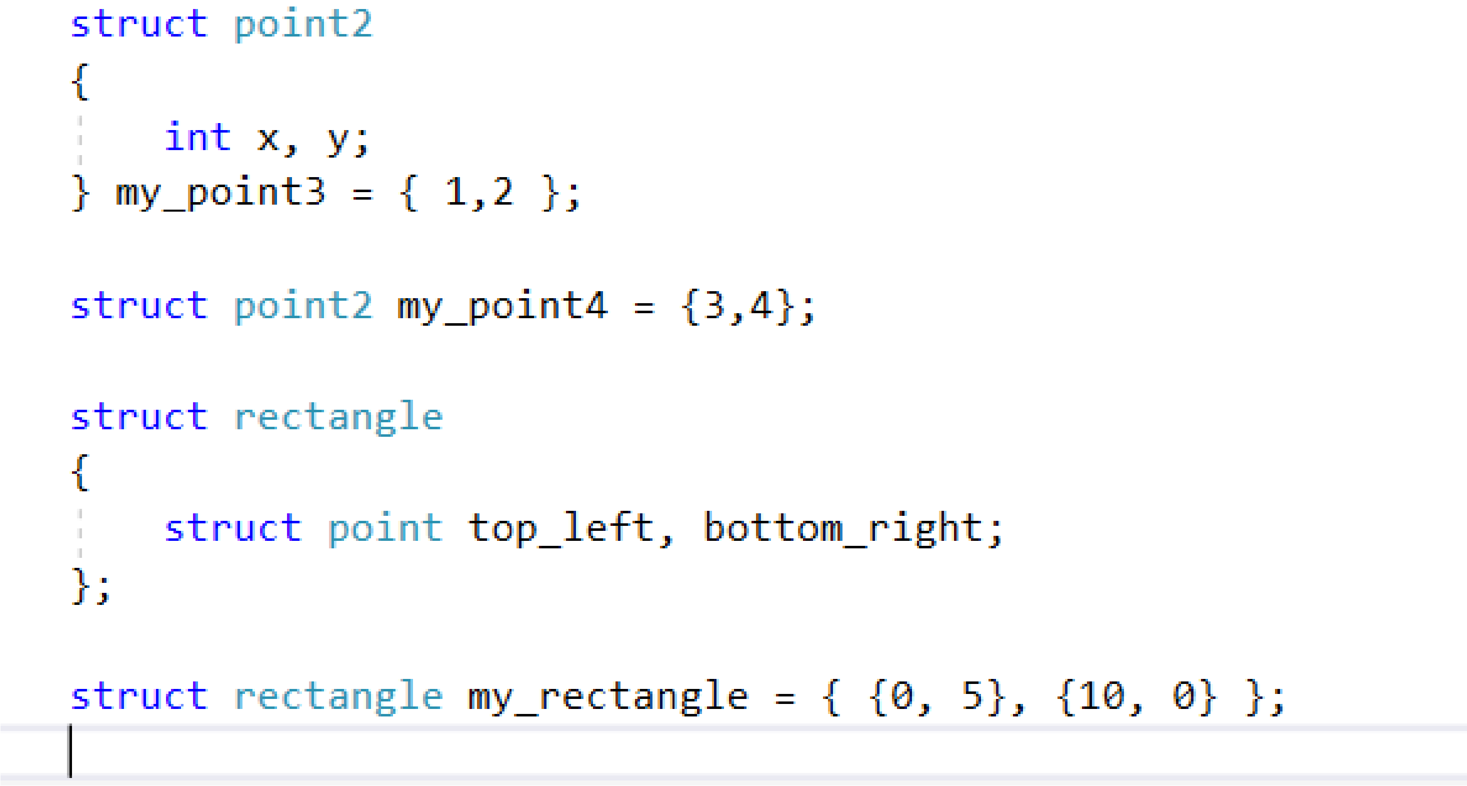
Hierarchical Structure

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;
```

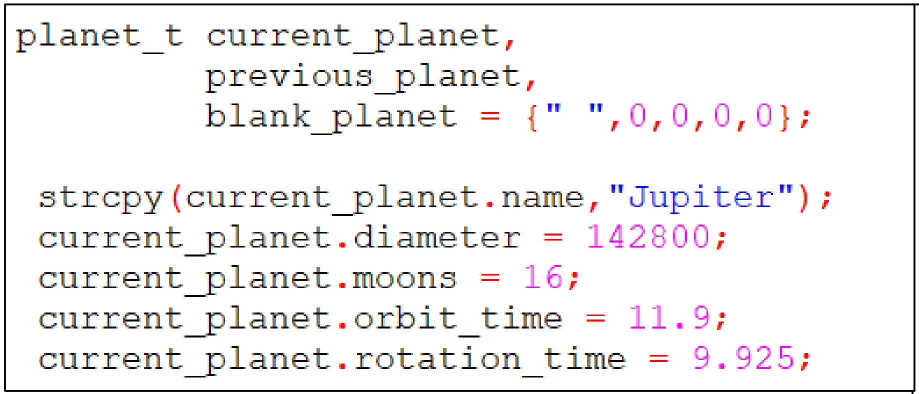
Initializing Structure Members



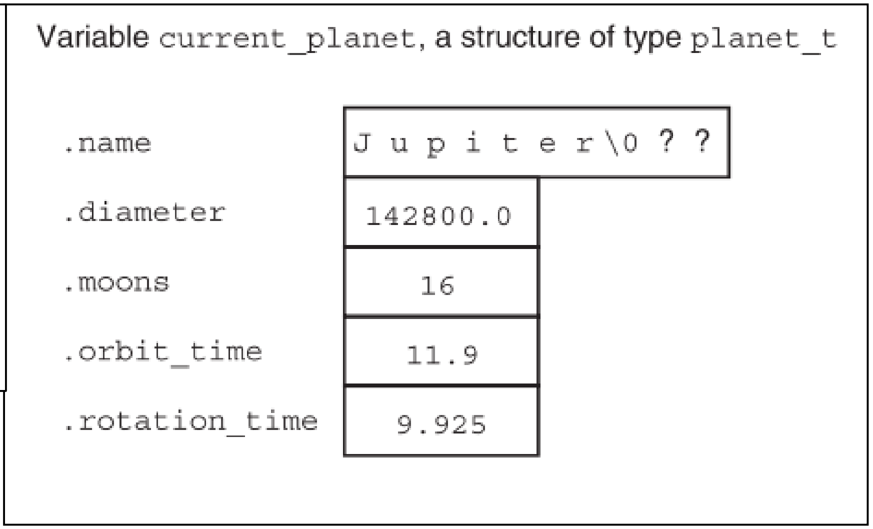
```
struct point2
{
    int x, y;
} my_point3 = { 1,2 };

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	Jupiter\0??
.diameter	142800.0
.moons	16
.orbit_time	11.9
.rotation_time	9.925

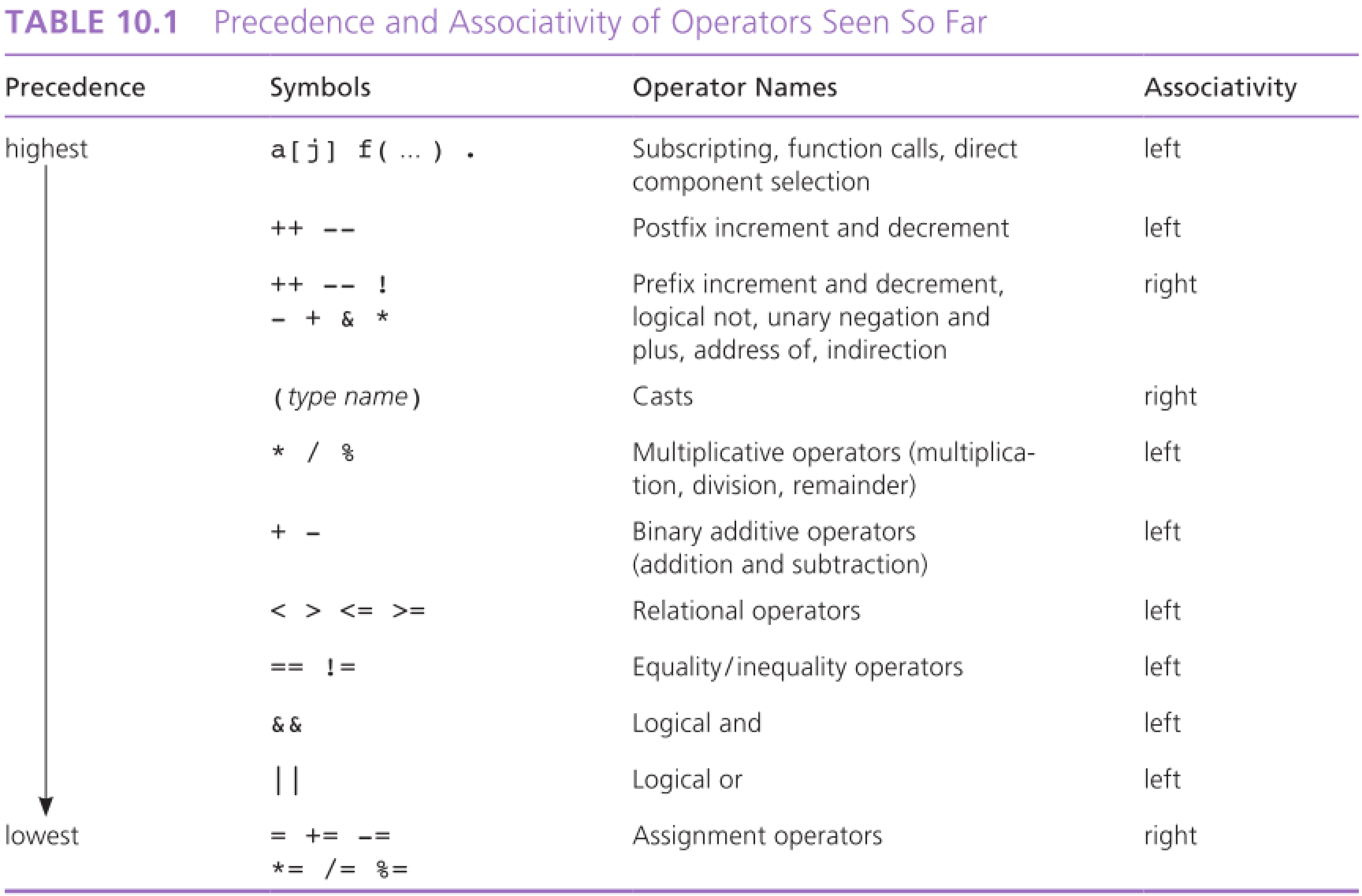


TABLE 10.1 Precedence and Associativity of Operators Seen So Far

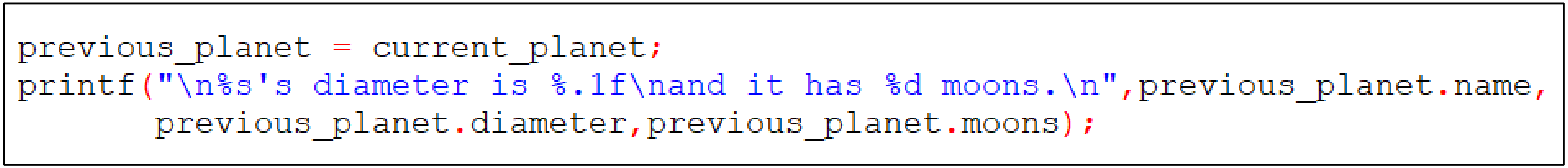
Precedence	Symbols	Operator Names	Associativity
highest	`a[j] f(…) .`	Subscripting, function calls, direct component selection	left
	`++ --`	Postfix increment and decrement	left

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

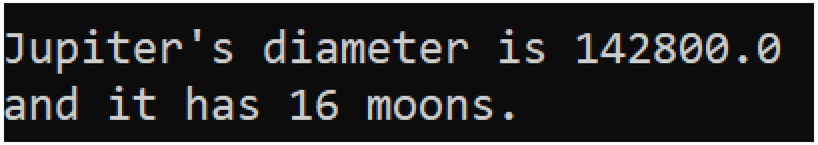
	`(`*type name*`)`	Casts	right
	`* / %`	Multiplicative operators (multiplication, division, remainder)	left
	`+ -`	Binary additive operators (addition and subtraction)	left
	`< > <= >=`	Relational operators	left
	`== !=`	Equality/inequality operators	left
	`&&`	Logical and	left
	`||`	Logical or	left

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

Assignment Operator



```
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);
```



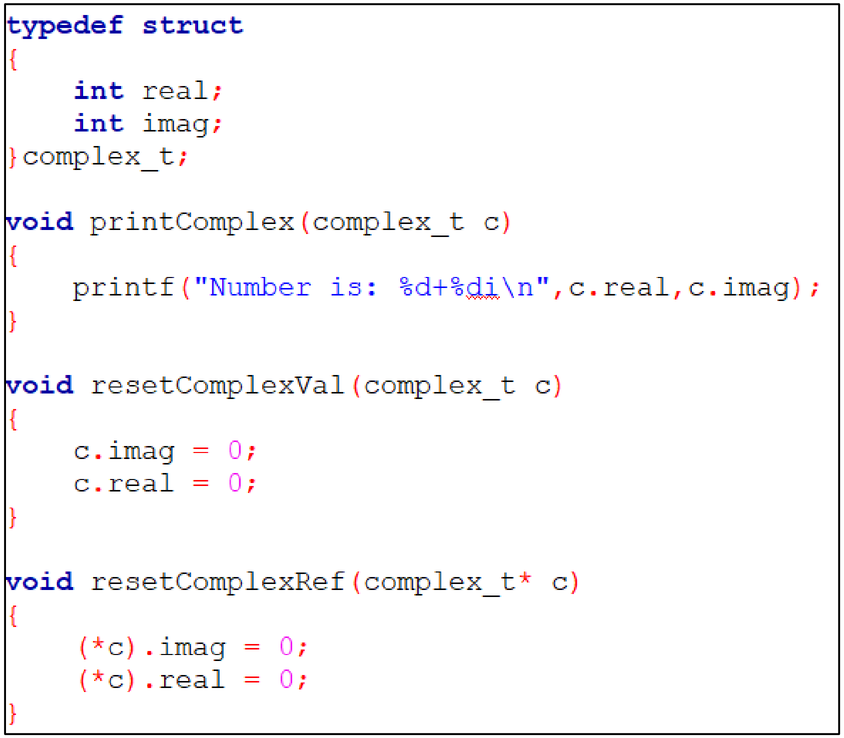
```
Jupiter's diameter is 142800.0
and it has 16 moons.
```

What if structure has pointer variables ?

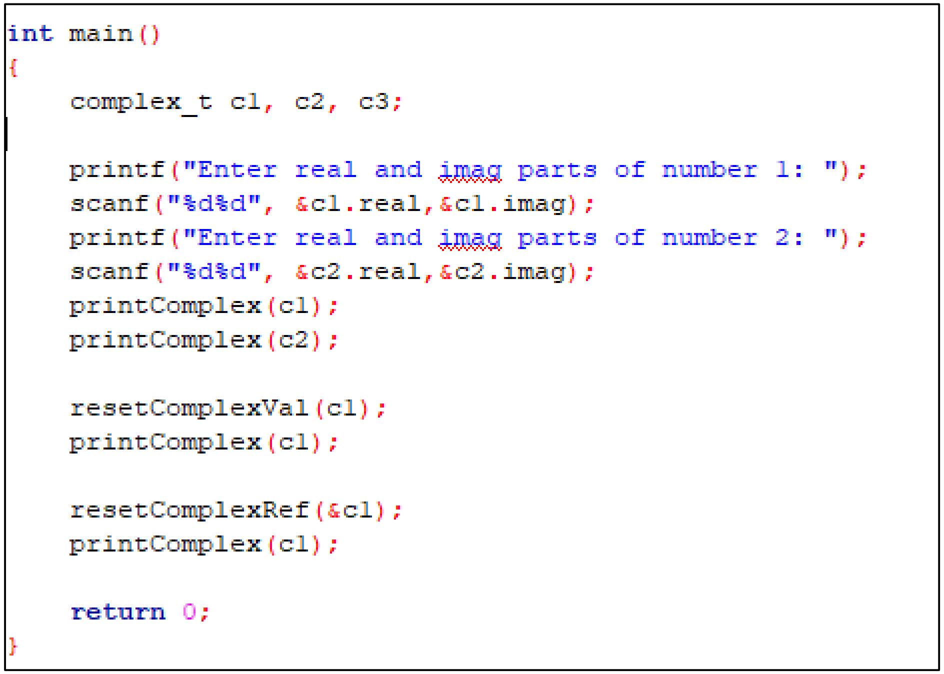
Structure Data Type as Input and Output Parameters

- 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.

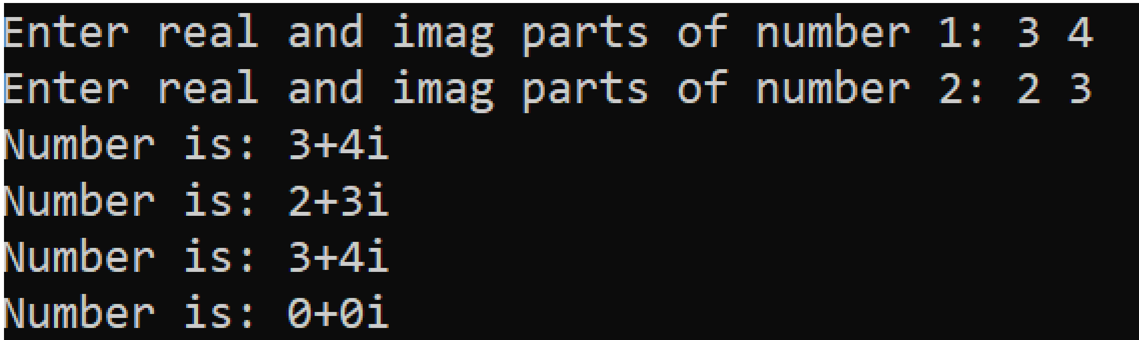
Pass by Value - Pass by Reference



```
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;
}
```

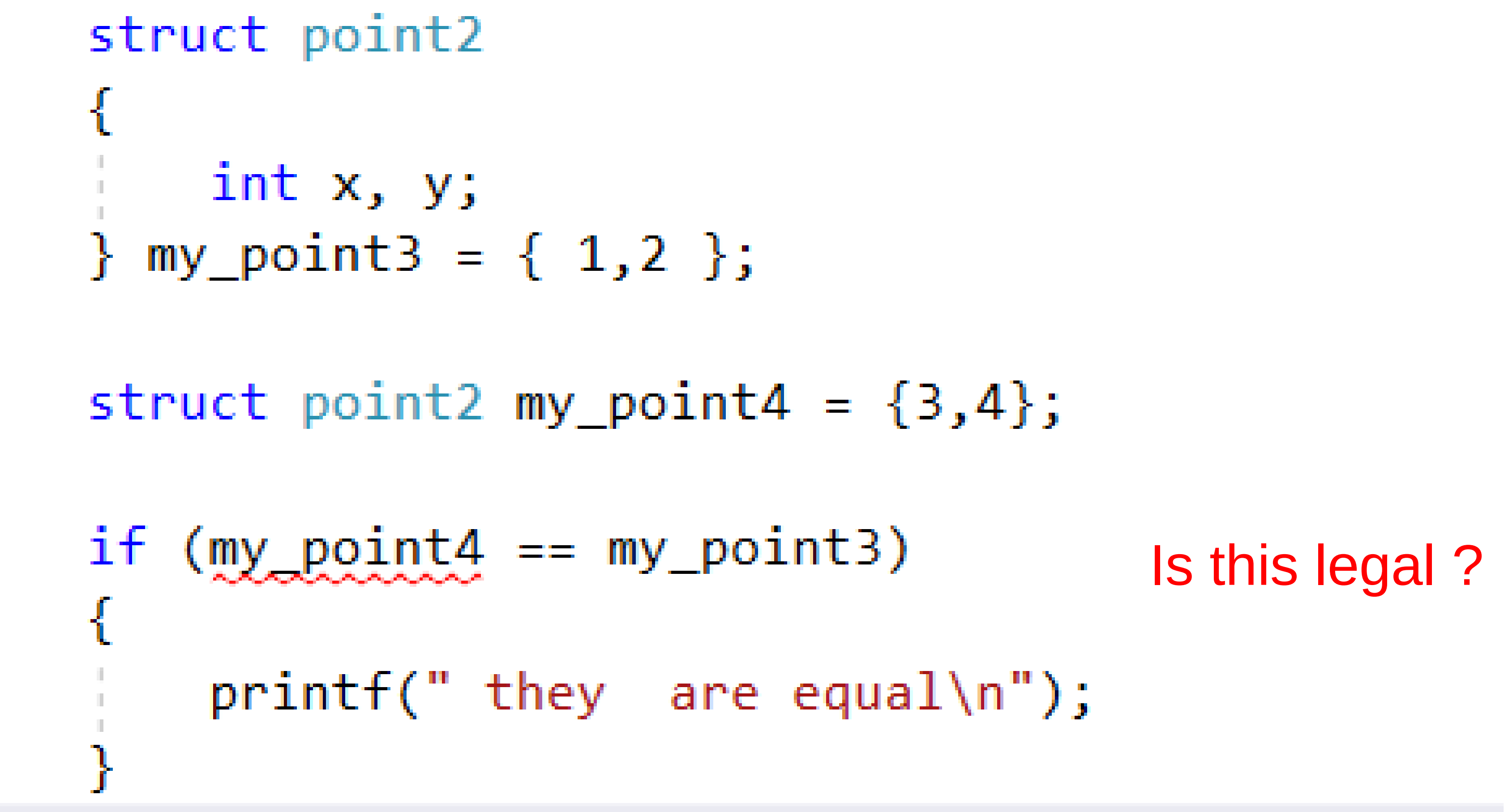


```
int main()
{
    complex_t c1, c2, c3;

    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
```

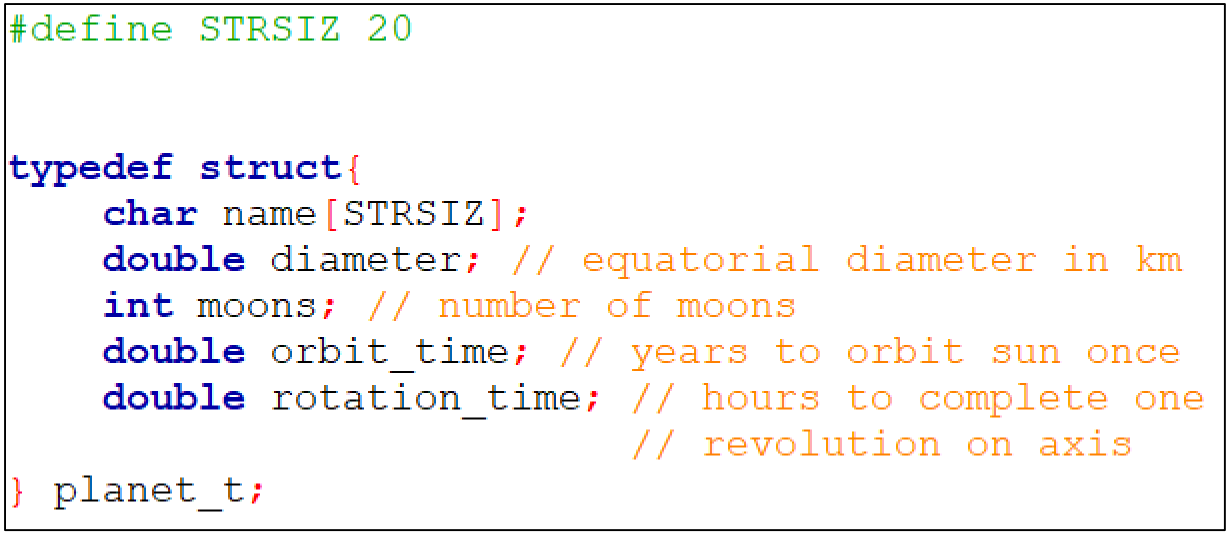
Equality Check



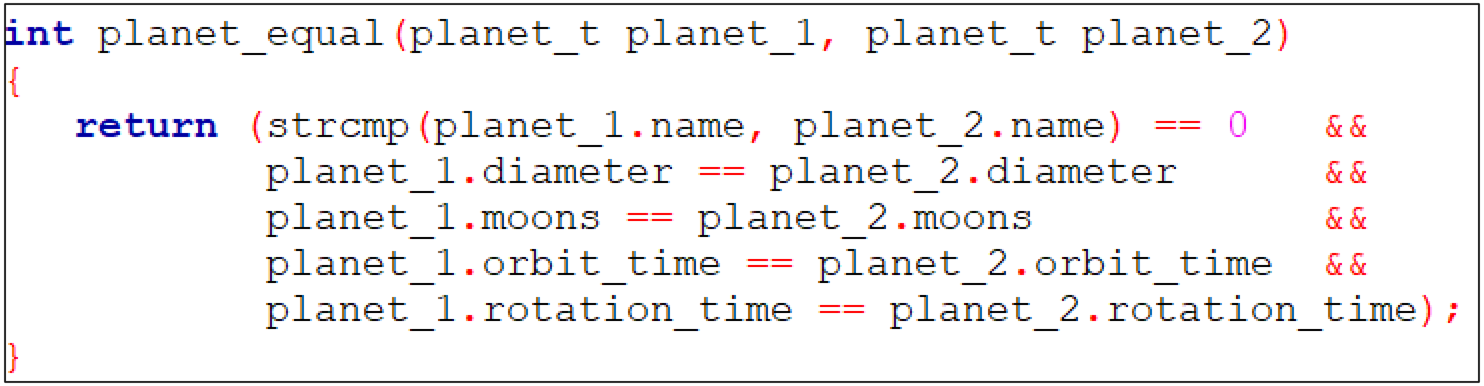
```
struct point2
{
    int x, y;
} my_point3 = { 1,2 };

struct point2 my_point4 = {3,4};

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

Is this legal ?

Equality Check

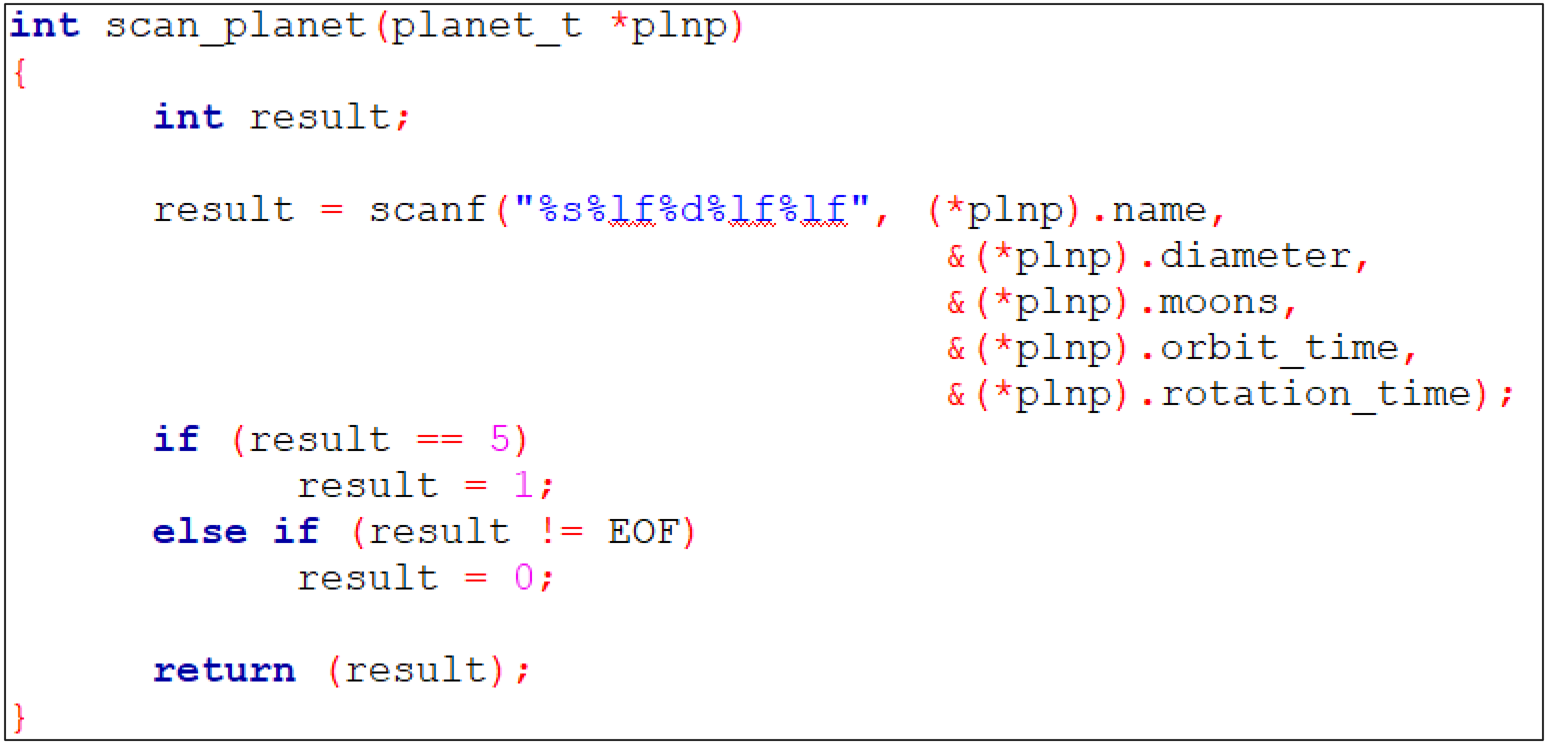


```
#define STRSIZ 20


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



```c
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);
}
```

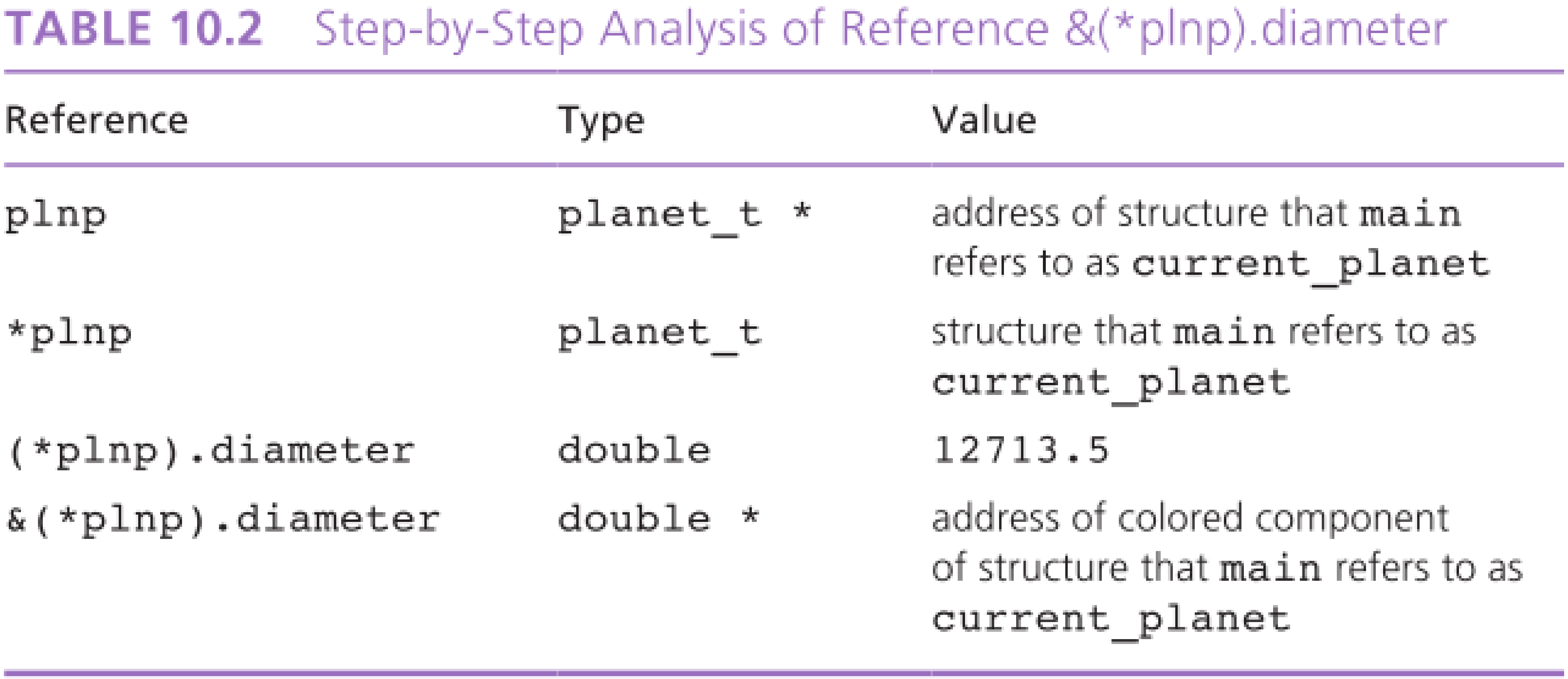
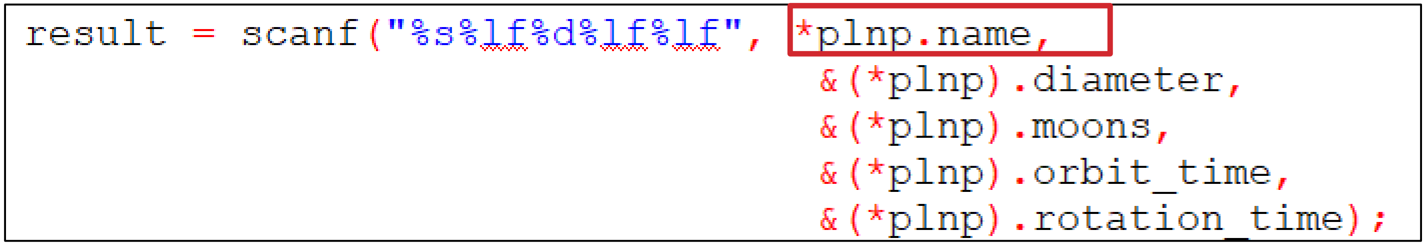


TABLE 10.2 Step-by-Step Analysis of Reference &(*plnp).diameter

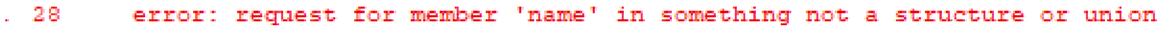
Reference	Type	Value
`plnp`	`planet_t *`	address of structure that `main` refers to as `current_planet`
`*plnp`	`planet_t`	structure that `main` refers to as `current_planet`
`(*plnp).diameter`	`double`	`12713.5`
`&(*plnp).diameter`	`double *`	address of colored component of structure that `main` refers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

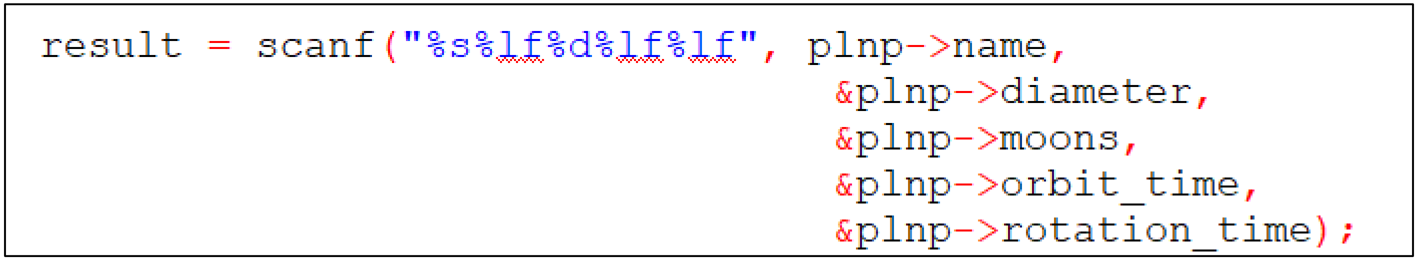
. (direct component selection dot) comes before

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

Put parantheses!!

Structure Data Type as Input and Output Parameters

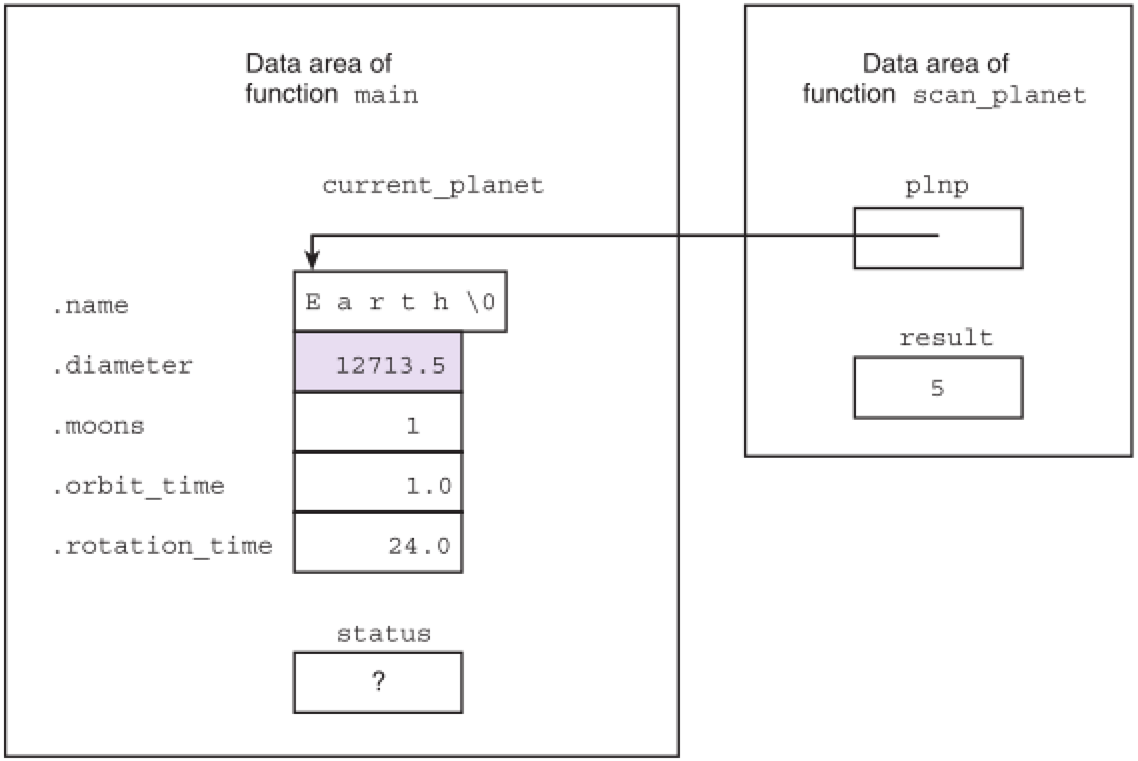
- indirect component selection operator
  - the character sequence **->** placed between a pointer variable and a component name creates a reference that follows the pointer to a structure and selects the component



```
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);`



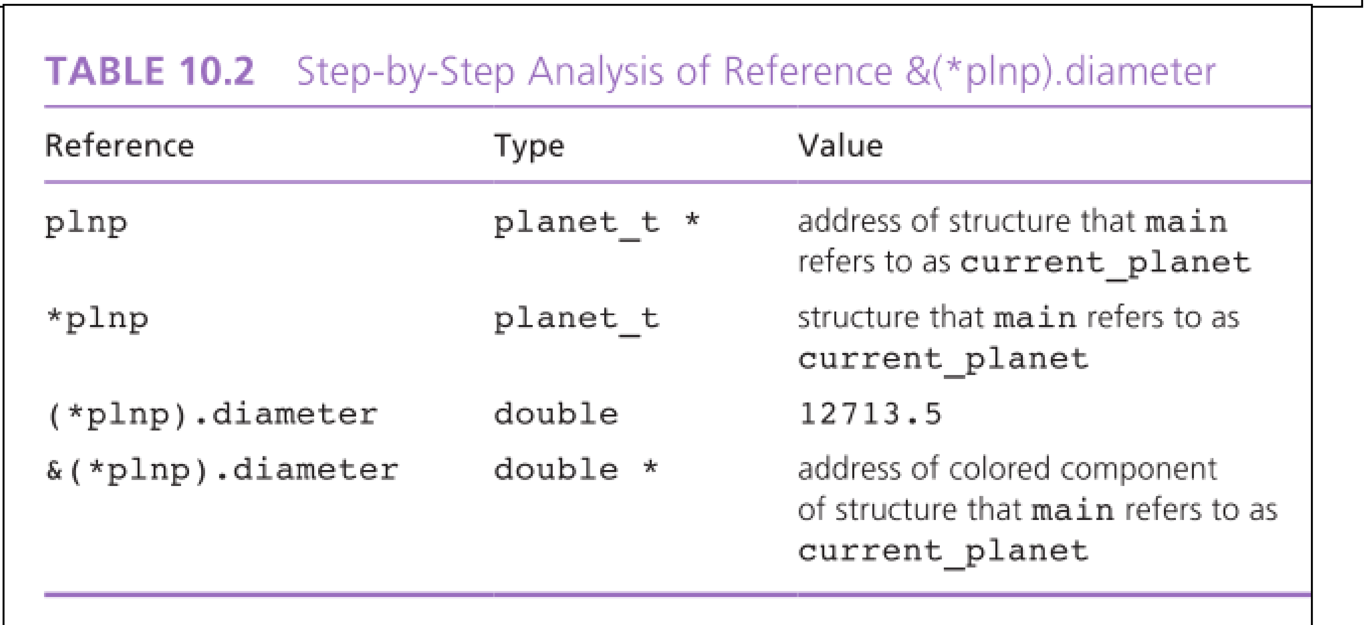


TABLE 10.2 Step-by-Step Analysis of Reference &(*plnp).diameter

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

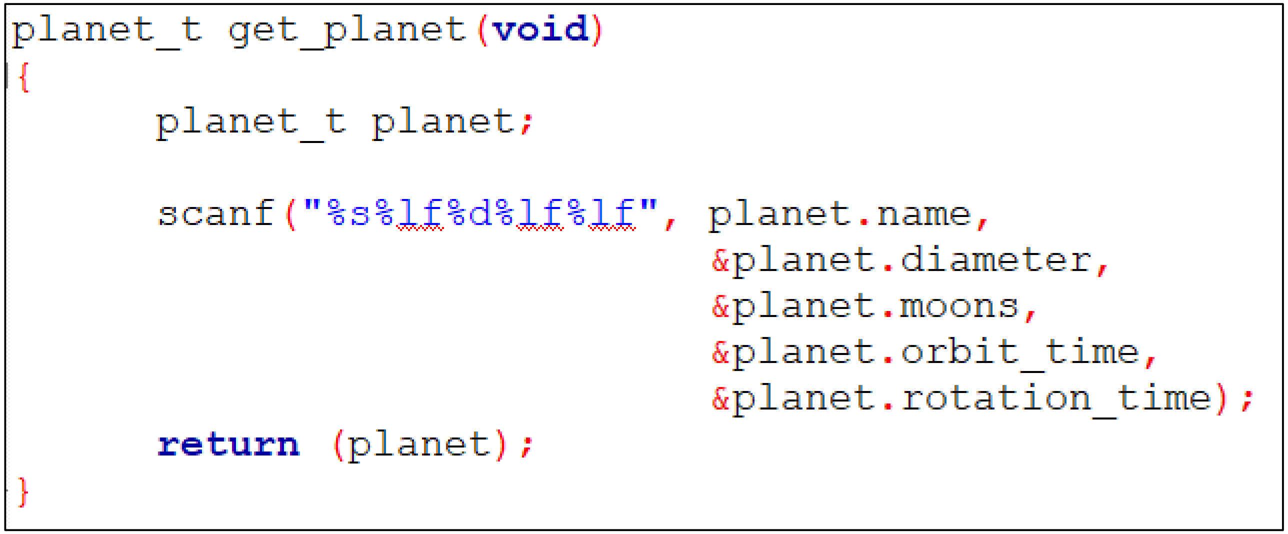
- A function that computes a structured result can be modeled on a function computing a simple result.

- 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);
}
```

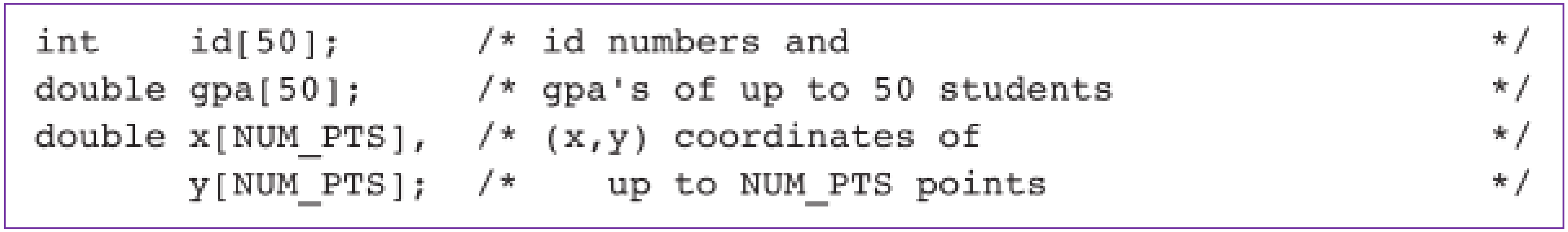
current_planet = get_planet()

has the same effect as:

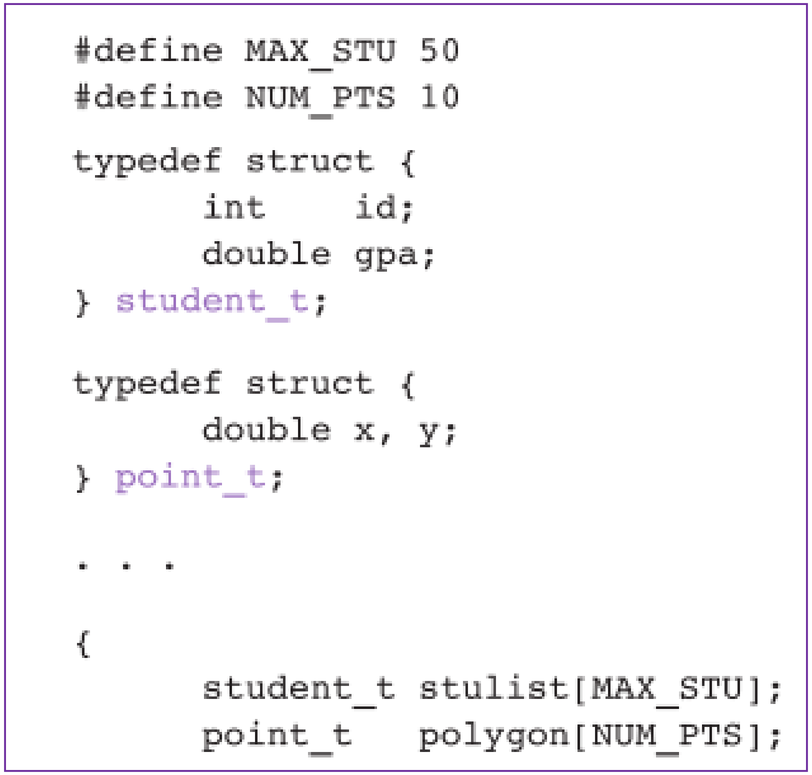
scan_planet(&current_planet)

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.



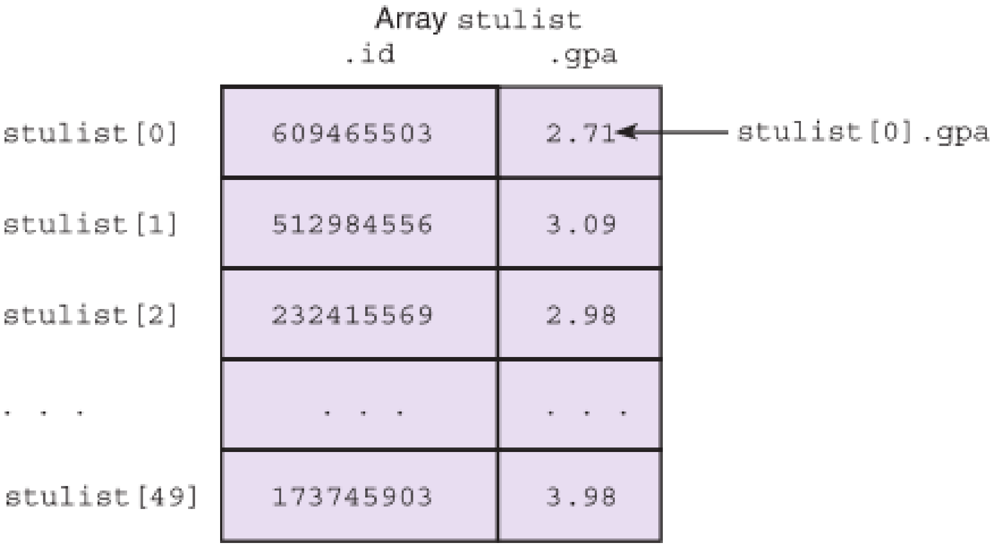
```
int    id[50];       /* id numbers and                                  */
double gpa[50];      /* gpa's of up to 50 students                      */
double x[NUM_PTS],   /* (x,y) coordinates of                            */
       y[NUM_PTS];   /*    up to NUM_PTS points                        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



```
#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

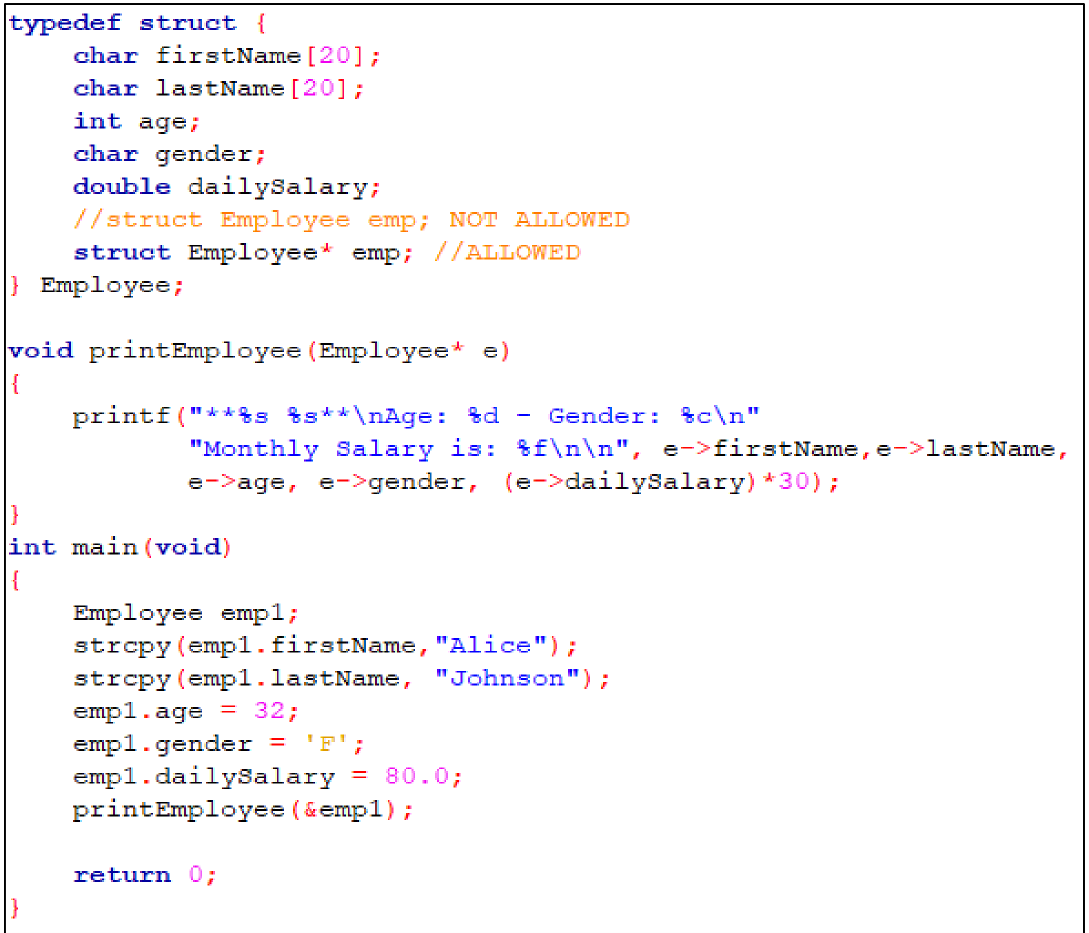
	.id	.gpa
stulist[0]	609465503	2.71
stulist[1]	512984556	3.09
stulist[2]	232415569	2.98
. . .	. . .	. . .
stulist[49]	173745903	3.98

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

Self-Referential Structures

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

Where to use?

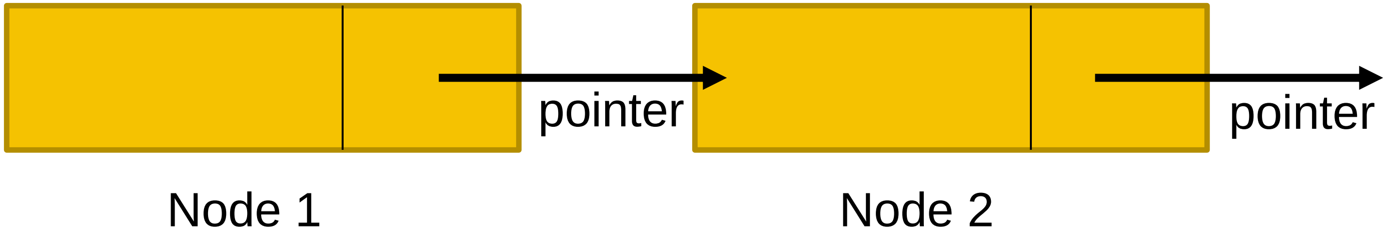


```
typedef struct {
    char firstName[20];
    char lastName[20];
    int age;
    char gender;
    double dailySalary;
    //struct Employee emp; NOT ALLOWED
    struct Employee* emp; //ALLOWED
} Employee;

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

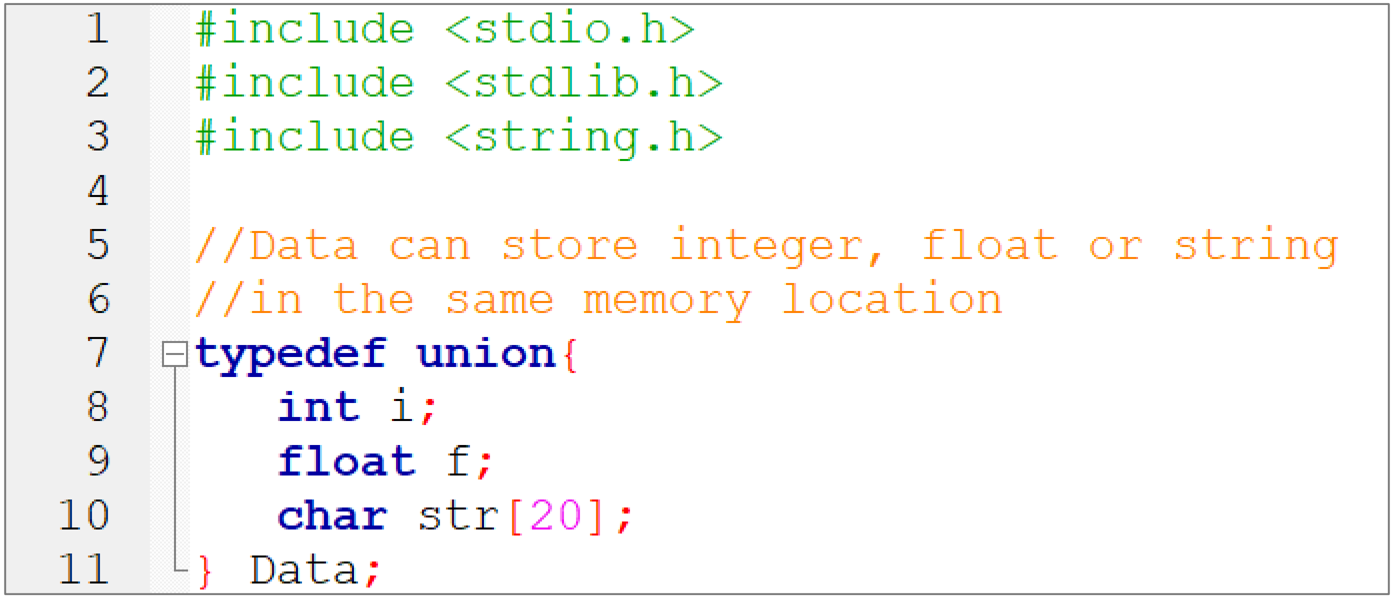
```
struct node_type {
            int data;
            struct node_type *next;
};
```



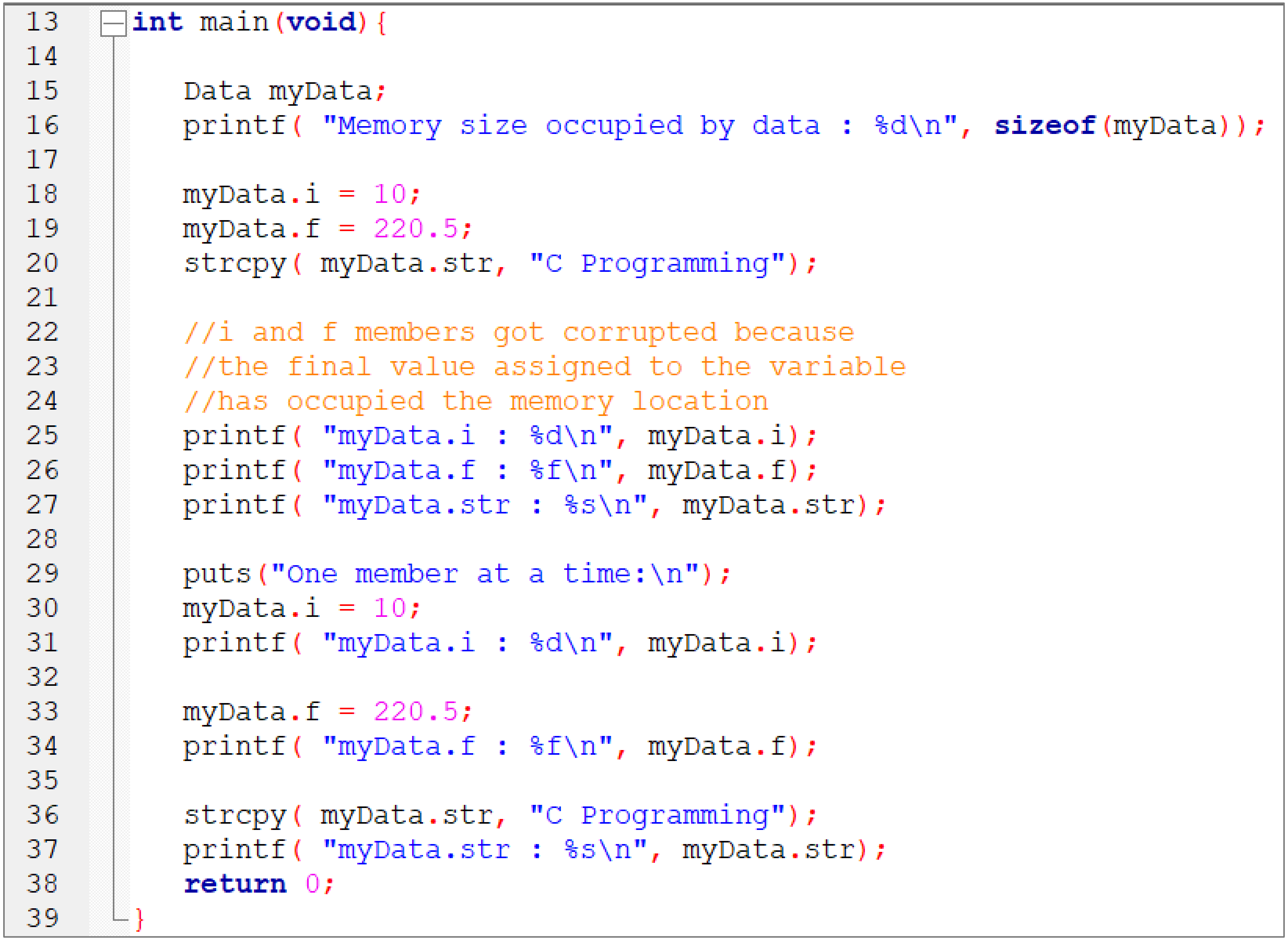
Union Types

- 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

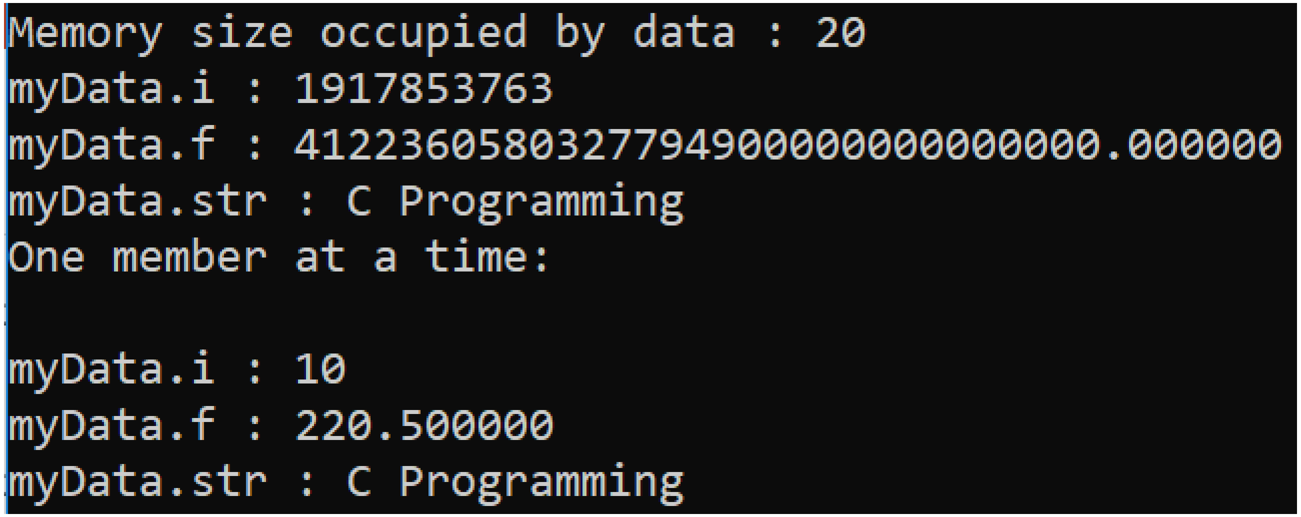
Union Types



```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;
```



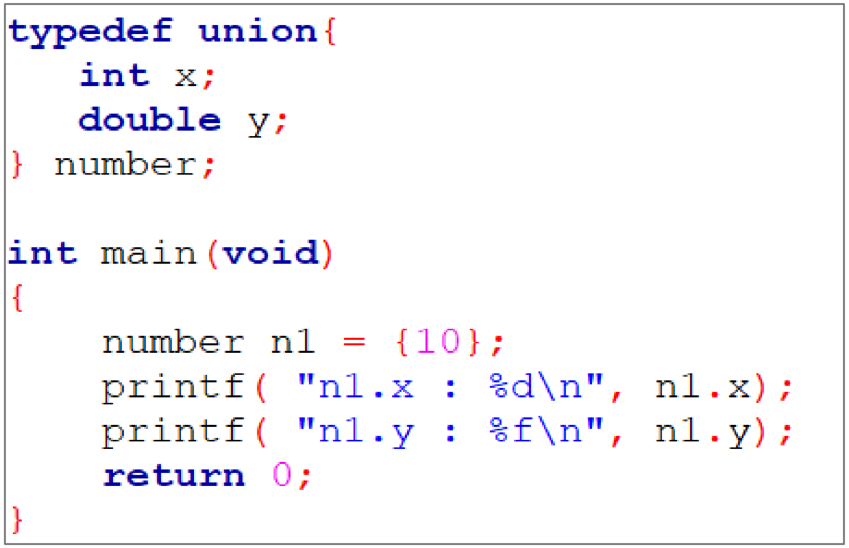
```c
int main(void){

    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;
}
```



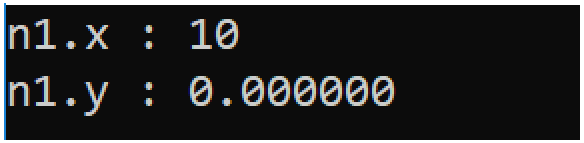
```
Memory size occupied by data : 20
myData.i : 1917853763
myData.f : 4122360580327794900000000000000.000000
myData.str : C Programming
One member at a time:

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

Initialization at Declaration Time

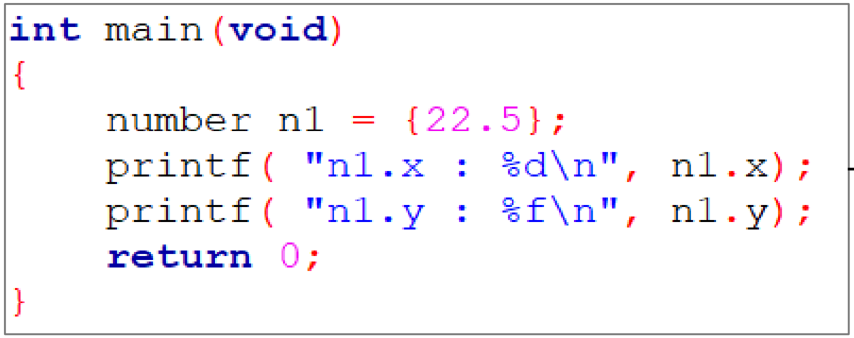
- Initialization with a value of the same type of the first member is allowed.



```
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;
}
```



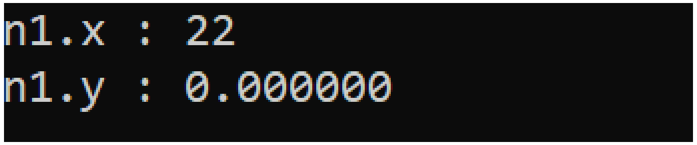
```
n1.x : 10
n1.y : 0.000000
```



```
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
```

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.