CMPE 252
C PROGRAMMING

SPRING 2021

WEEK 13

PROGRAMMING IN THE LARGE
CHAPTER 12

Problem Solving & Program Design in C

Eighth Edition
Global Edition

Jeri R. Hanly & Elliot B. Koffman

Chapter Objectives

- To learn how procedural abstraction help the programmer separate concerns about what a function does from the details of how to code the function
- To understand how data abstraction enables us to describe what information is stored in an object and what operations we want to perform on the object without knowing the specifics of how the object's information is organized and represented

Chapter Objectives

- To learn how to create your own personal library with a separate header file and implementation file and to understand what should be stored in each file
- To understand the purpose of different storage classes in C
- To learn how to use conditional compilation to prevent multiple declarations of the identifiers in a header file

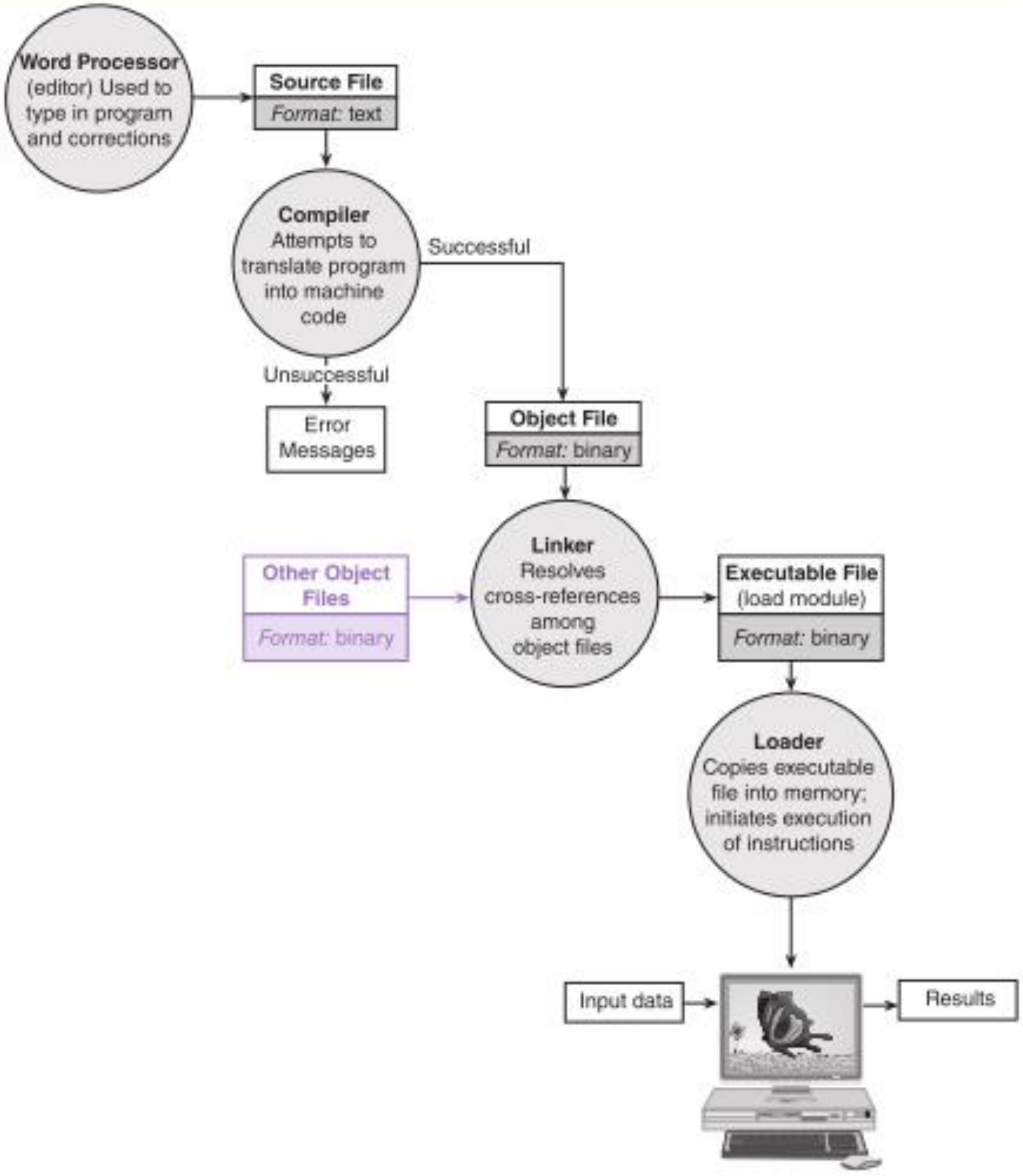
Chapter Objectives

- To learn how to use multidimensional arrays for storing tables of data
- To learn how to declare parameters for function main and how to pass data such as file names through command-line arguments
- To learn how to define macros with parameters and understand what happens when a macro is expanded

Personal Libraries

- header file
  - text file containing the interface information about a library needed by a compiler to translate a program system that uses the library or by a person to understand and use the library
- Until now we have seen C’s standard libraries, but they are not enough for custom projects

FIGURE 12.1 Preparing a Program for Execution


Word Processor
(editor) Used to type in program and corrections
Source File
Format: text
Compiler
Attempts to translate program into machine code
Successful
Unsuccessful
Error Messages
Object File
Format: binary
Other Object Files
Format: binary
Linker
Resolves cross-references among object files
Executable File
(load module)
Format: binary
Loader
Copies executable file into memory; initiates execution of instructions
Input data
Results

Personal Libraries

- Typical header file includes:
  - a block comment summarizing the library’s purpose
  - `#define` directives naming constant macros
  - type definitions
  - block comment summarizing each function’s purpose and the declarations in the form: `extern` *`prototype`*

notifies the compiler that the function’s definition will be provided to the linker

- Create header file and put it to the folder where your source file resides

extern in functions

- When a function is declared or defined, the extern keyword is implicitly assumed. When we write.

-
  ```
      int foo(int arg1, char arg2);
  ```
- The compiler treats it as:

-
  ```
      extern int foo(int arg1, char arg2);
  ```

- Since the extern keyword extends the function’s visibility to the whole program, the function can be used (called) anywhere in any of the files of the whole program, provided those files contain a declaration of the function.
- (With the declaration of the function in place, the compiler knows the definition of the function exists somewhere else and it goes ahead and compiles the file). So that’s all about extern and functions.

- extern in variables are different → we’ll see in following slides.

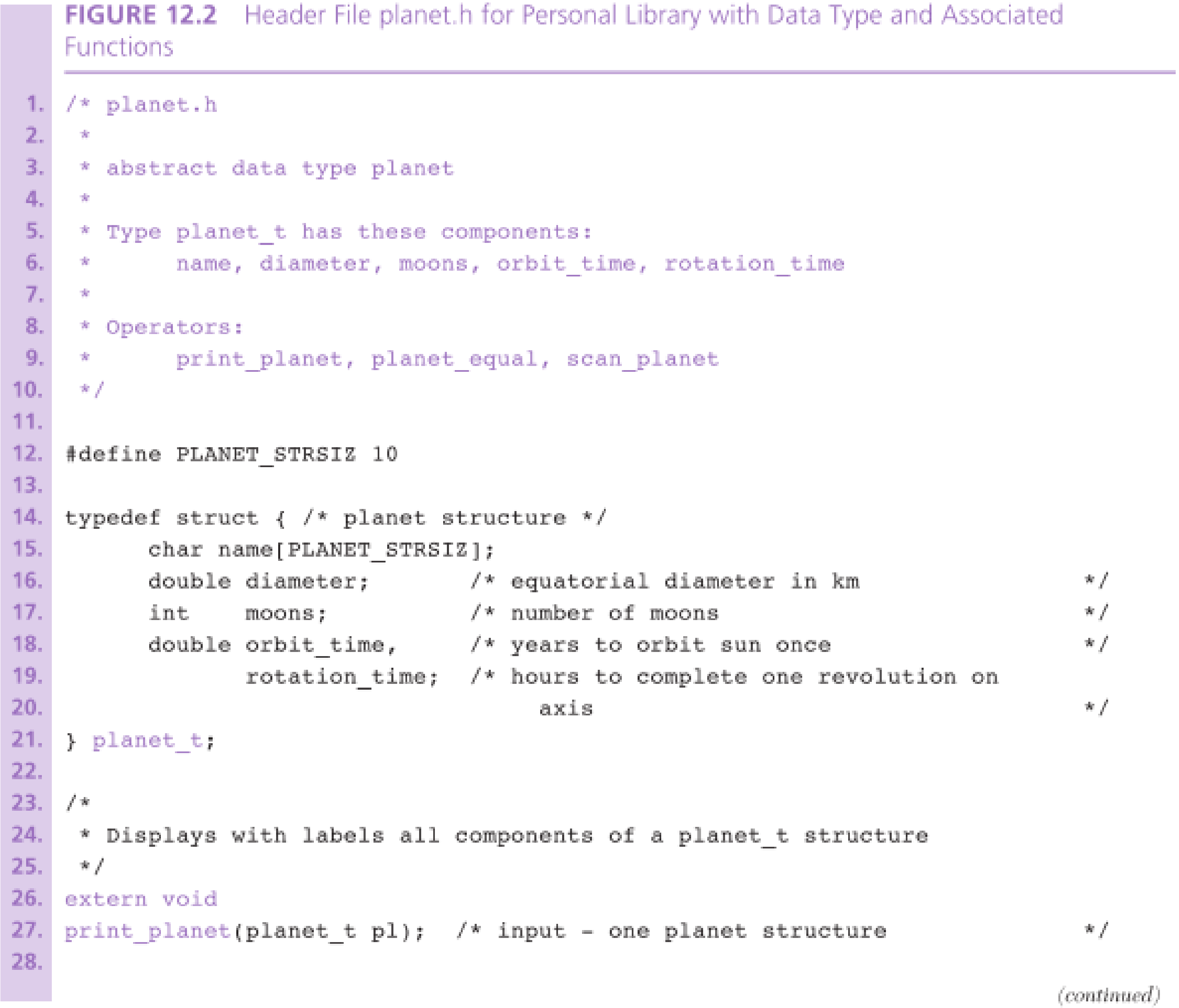
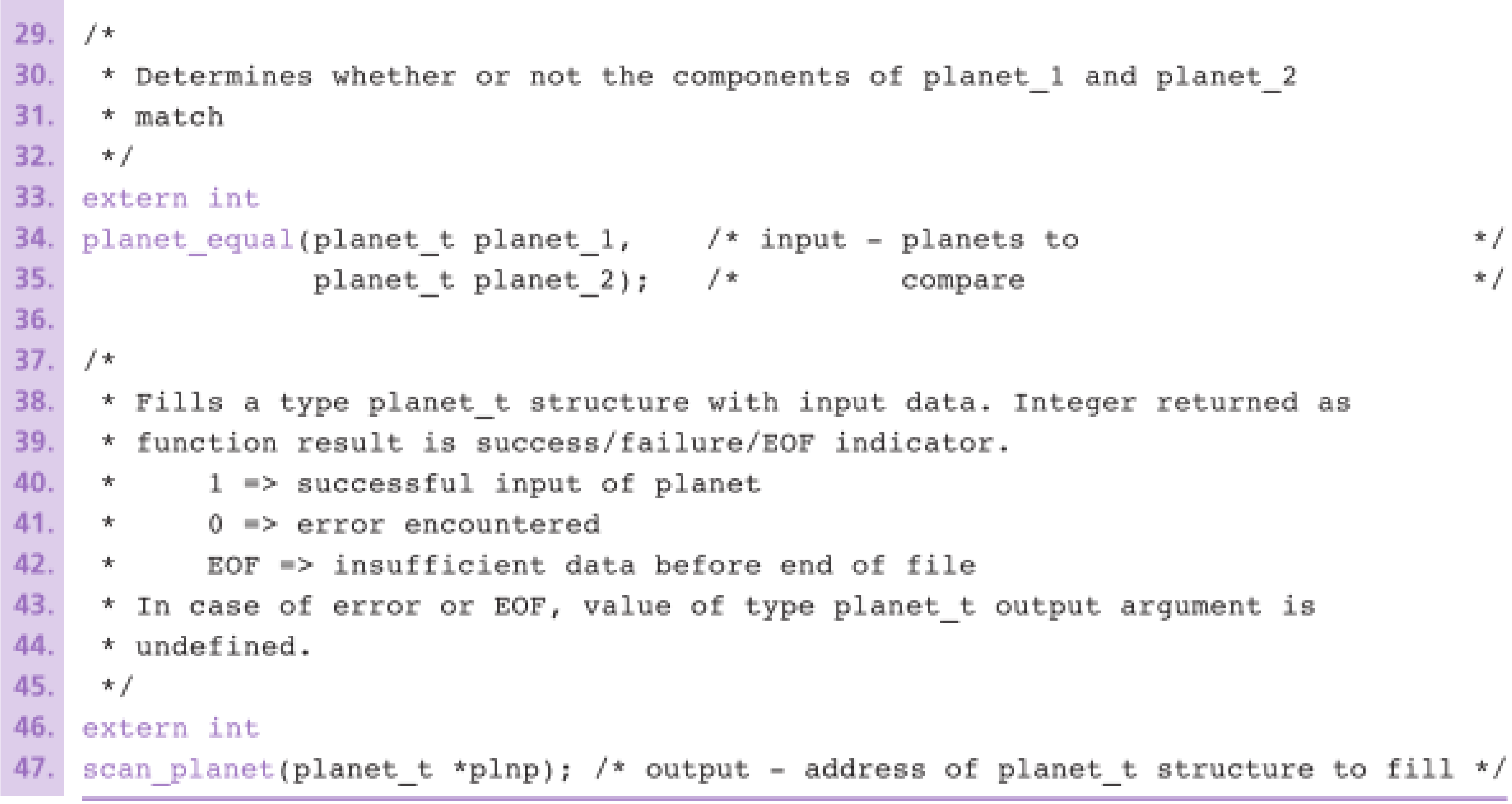


FIGURE 12.2 Header File planet.h for Personal Library with Data Type and Associated Functions

```
/* planet.h
 *
 * abstract data type planet
 *
 * Type planet_t has these components:
 *      name, diameter, moons, orbit_time, rotation_time
 *
 * Operators:
 *      print_planet, planet_equal, scan_planet
 */

#define PLANET_STRSIZ 10

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

/*
 * Displays with labels all components of a planet_t structure
 */
extern void
print_planet(planet_t pl);  /* input - one planet structure              */

```

(continued)

FIGURE 12.2 (continued)



```
/*
 * Determines whether or not the components of planet_1 and planet_2
 * match
 */
extern int
planet_equal(planet_t planet_1,    /* input - planets to                      */
             planet_t planet_2);   /*         compare                         */

/*
 * Fills a type planet_t structure with input data. Integer returned as
 * function result is success/failure/EOF indicator.
 *     1 => successful input of 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.
 */
extern int
scan_planet(planet_t *plnp); /* output - address of planet_t structure to fill */
```

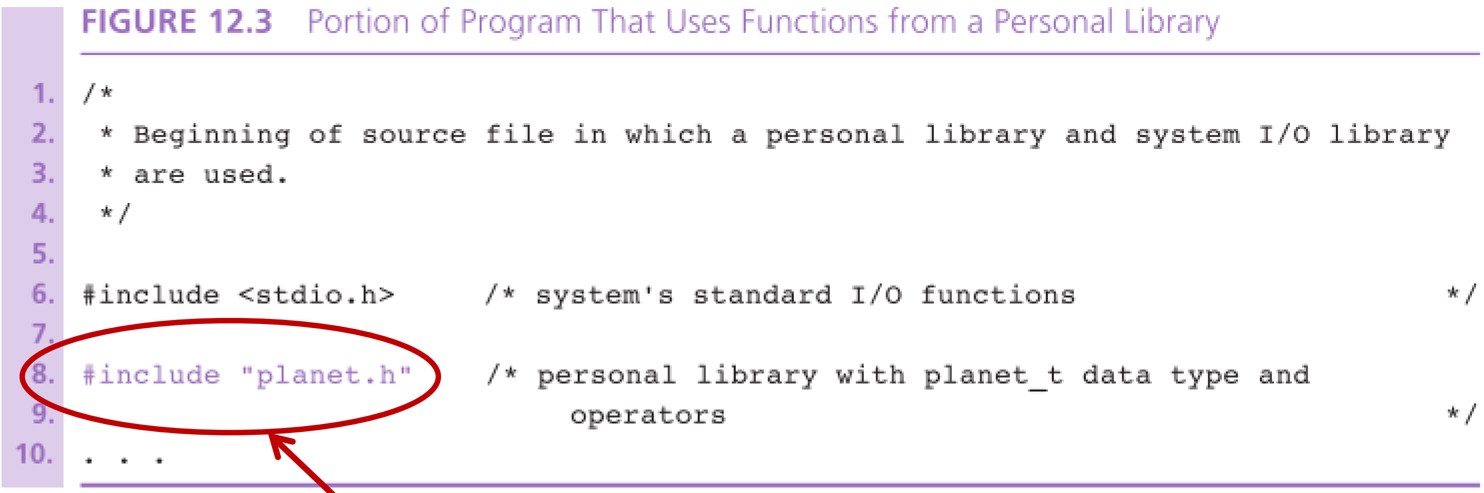
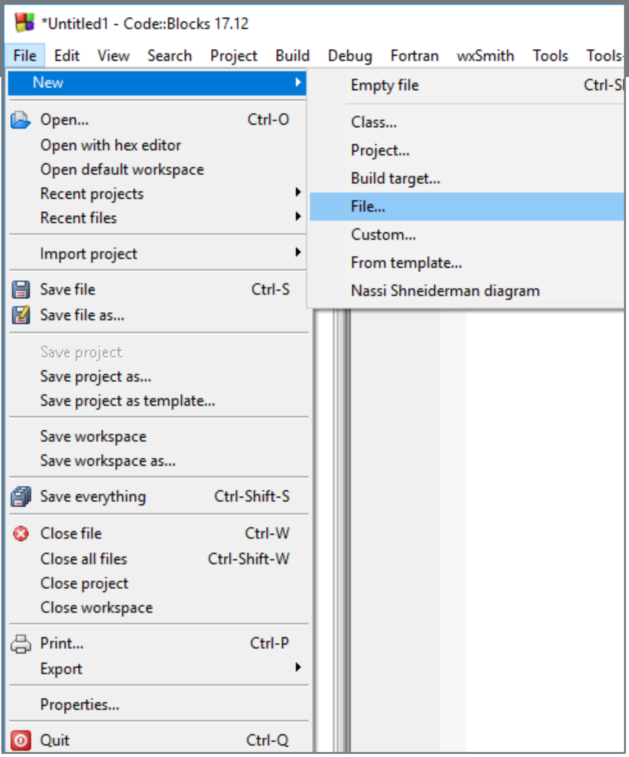
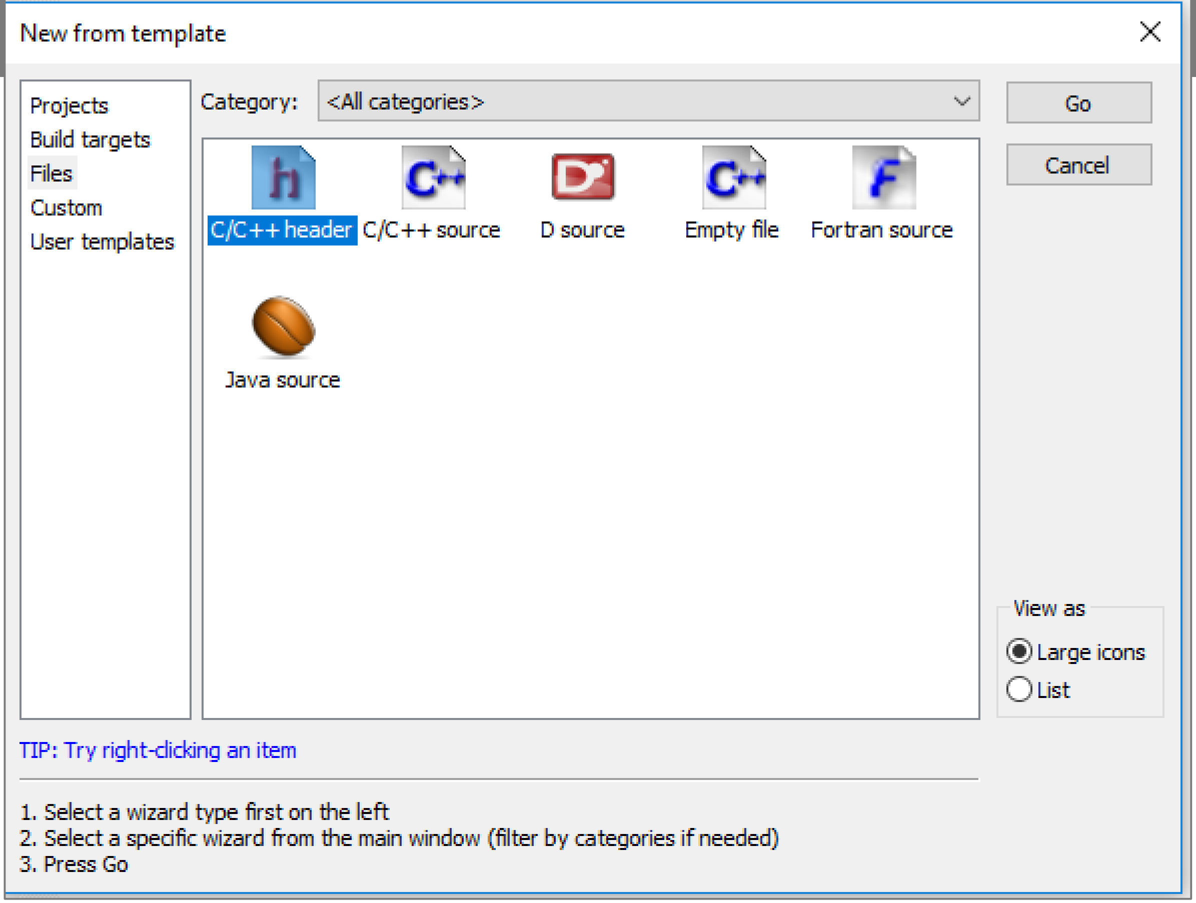


FIGURE 12.3 Portion of Program That Uses Functions from a Personal Library

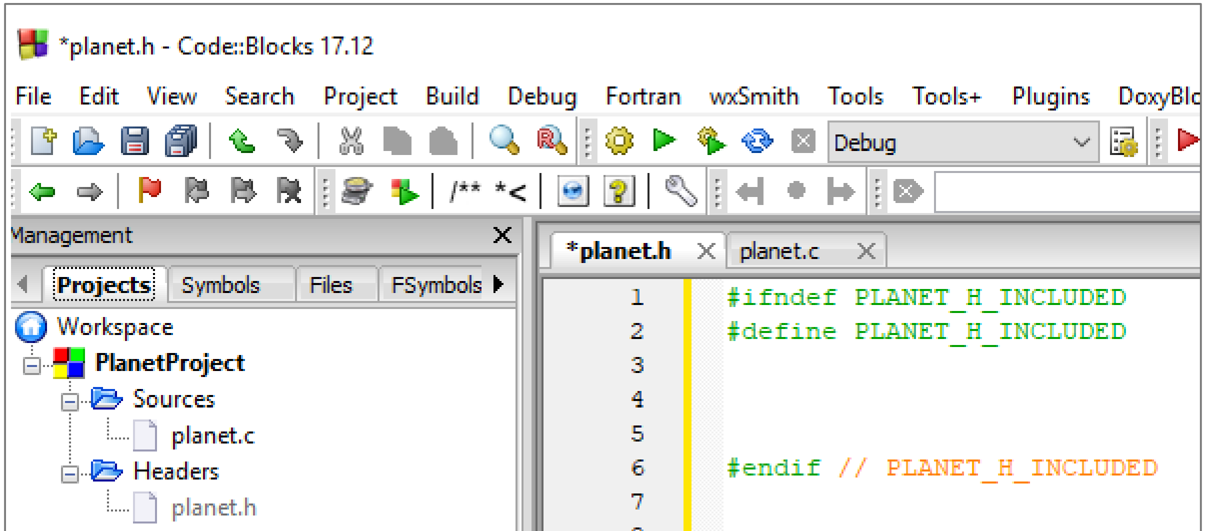
```
/*
 * Beginning of source file in which a personal library and system I/O library
 * are used.
 */

#include <stdio.h>     /* system's standard I/O functions                     */

#include "planet.h"    /* personal library with planet_t data type and
                            operators                                         */
. . .
```

This library belongs to the programmer.





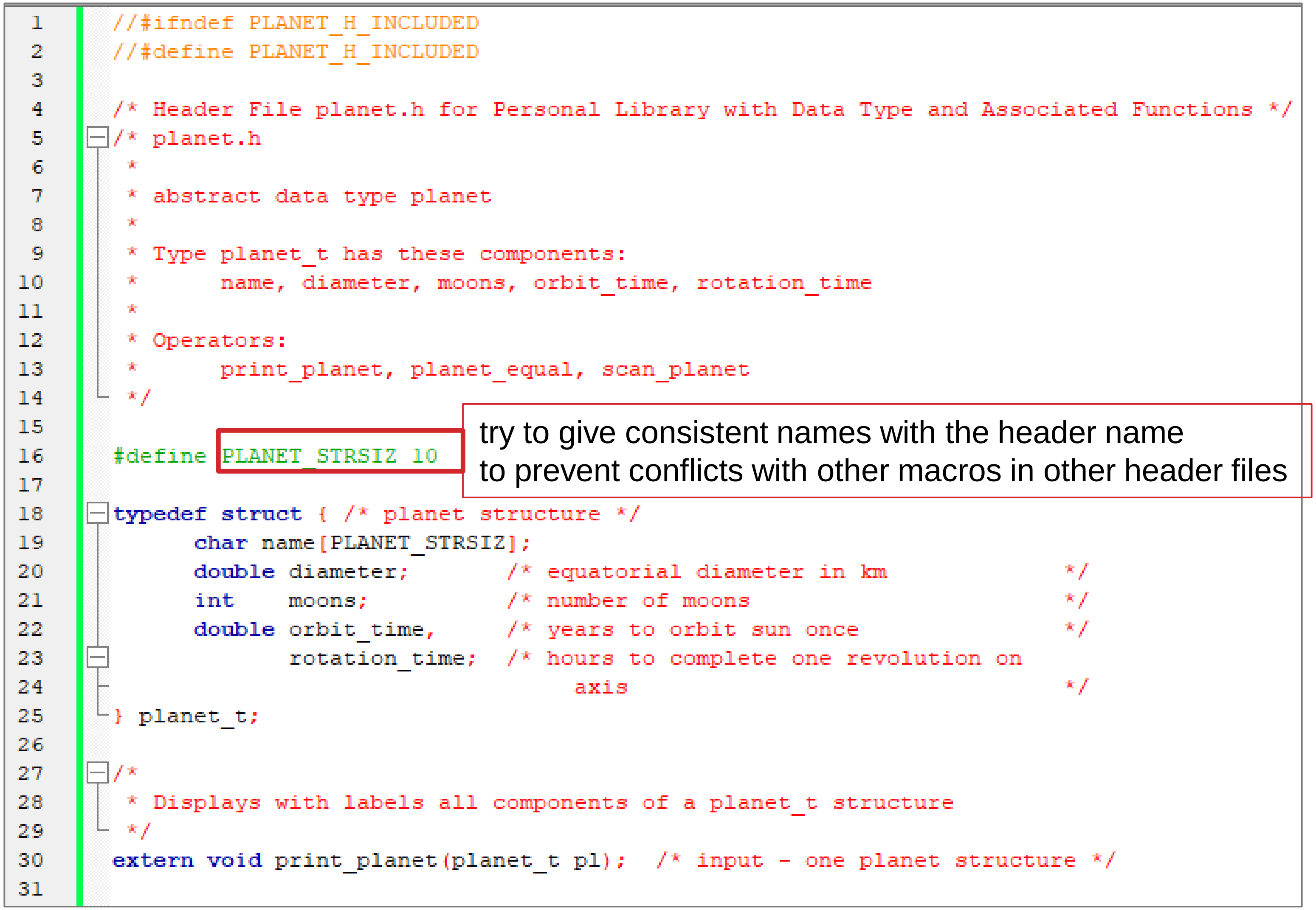
1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go


*planet.h - Code::Blocks 17.12
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlo
Debug
Management
Projects Symbols Files FSymbols
Workspace
PlanetProject
Sources
planet.c
Headers
planet.h
*planet.h
planet.c

```
#ifndef PLANET_H_INCLUDED
#define PLANET_H_INCLUDED



#endif // PLANET_H_INCLUDED
```

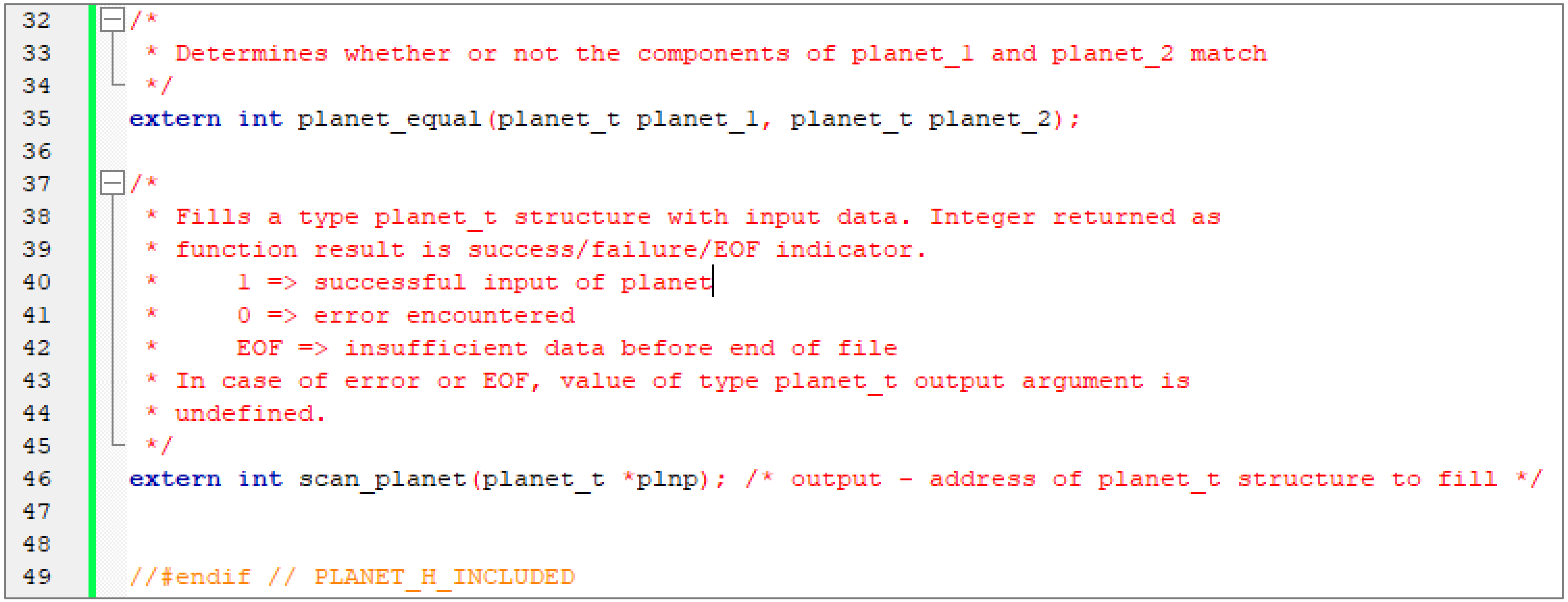
Conditional compilation.
Prevents multiple inclusion
of the same header file content.
For now, remove/comment them.

planet.h



```c
//#ifndef PLANET_H_INCLUDED
//#define PLANET_H_INCLUDED

/* Header File planet.h for Personal Library with Data Type and Associated Functions */
/* planet.h
 *
 * abstract data type planet
 *
 * Type planet_t has these components:
 *      name, diameter, moons, orbit_time, rotation_time
 *
 * Operators:
 *      print_planet, planet_equal, scan_planet
 */

#define PLANET_STRSIZ 10

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

/*
 * Displays with labels all components of a planet_t structure
 */
extern void print_planet(planet_t pl);  /* input - one planet structure */

```

try to give consistent names with the header name
to prevent conflicts with other macros in other header files



```
/*
 * Determines whether or not the components of planet_1 and planet_2 match
 */
extern int planet_equal(planet_t planet_1, planet_t planet_2);

/*
 * Fills a type planet_t structure with input data. Integer returned as
 * function result is success/failure/EOF indicator.
 *     1 => successful input of 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.
 */
extern int scan_planet(planet_t *plnp); /* output - address of planet_t structure to fill */


//#endif // PLANET_H_INCLUDED
```

Personal Libraries

- implementation file
  - file containing the C source code of a library’s functions and any other information needed for compilation of these functions

Using a Personal Library

To use a personal library, one must complete these steps:

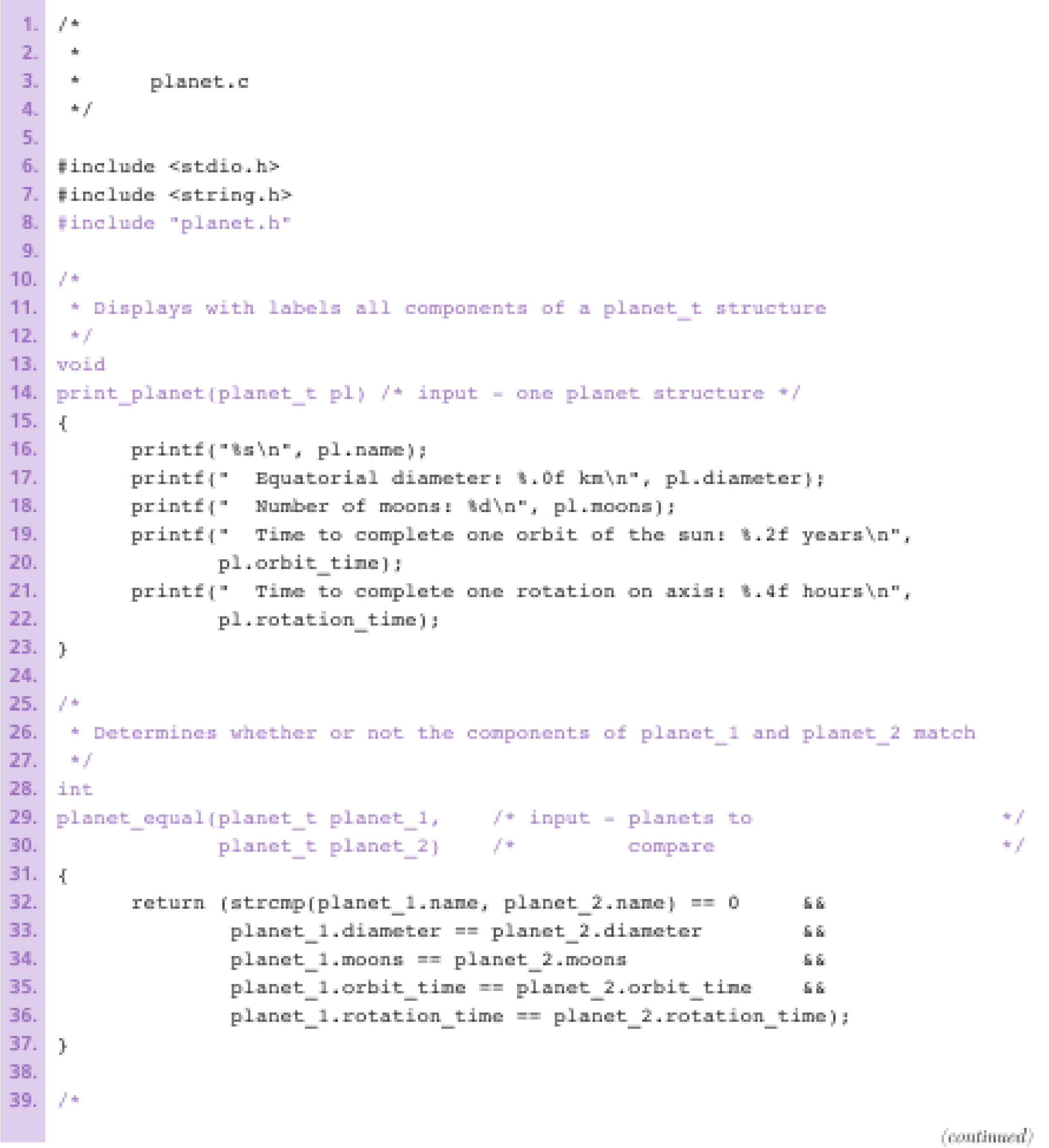
- **Creation:**
- C1
  - create a header file containing the interface information for a program needing the library
- C2
  - create an implementation file containing the code of the library functions and other details of the implementation that are hidden from the user program
- C3
  - compile the implementation file
  - this step must be repeated any time either the header file or the implementation file is revised

Steps for Use

- U1
  - include the library's header file in the user program through an #include directive

- U2
  - after compiling the user program, include both its object file and the object file created in C3 in the command that activates the linker

FIGURE 12.4 Implementation File planet.c Containing Library with Planet Data Type and Operators



```c
/*
 *
 *     planet.c
 */

#include <stdio.h>
#include <string.h>
#include "planet.h"

/*
 * 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);
}

/*
 * 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);
}

/*
```

(continued)

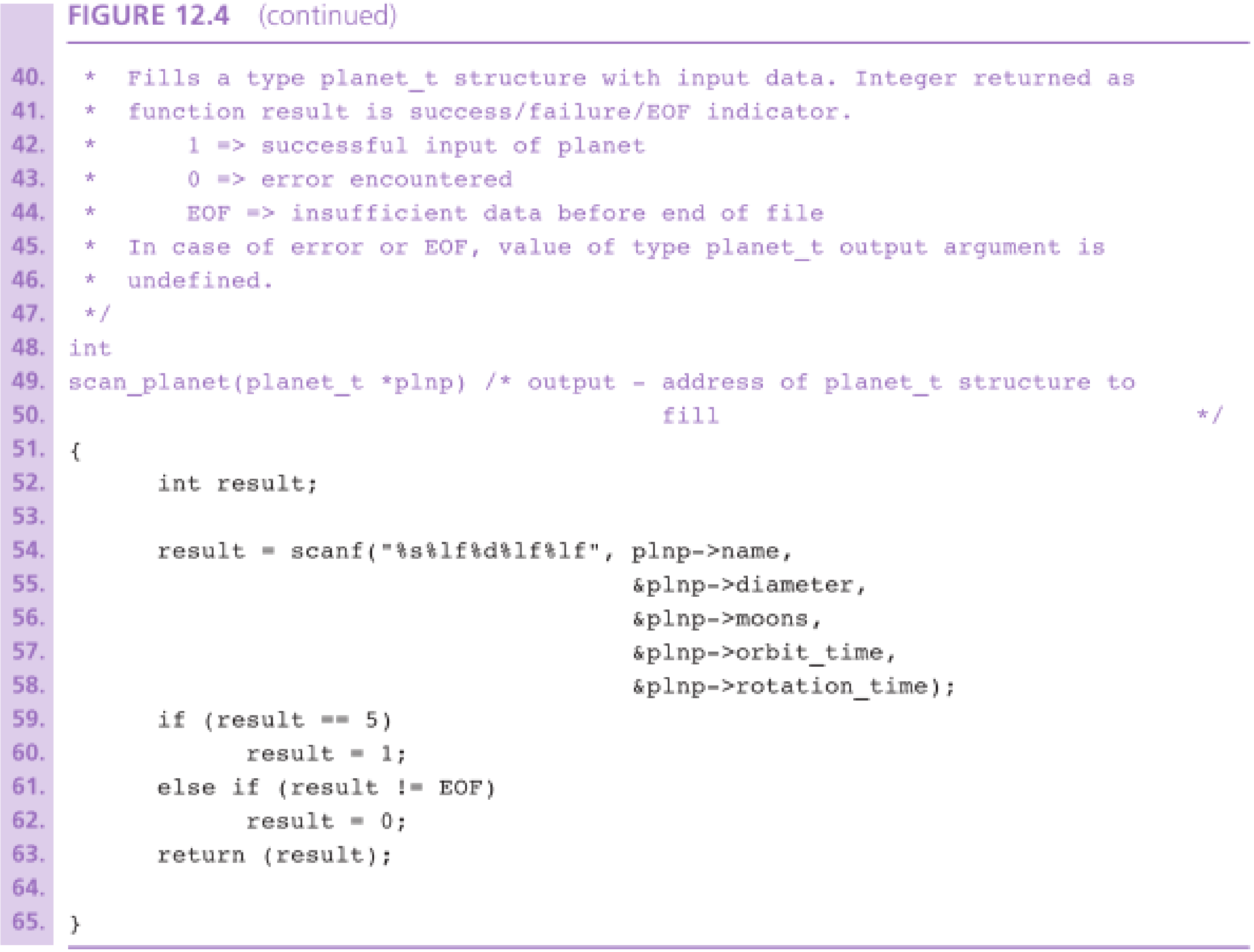
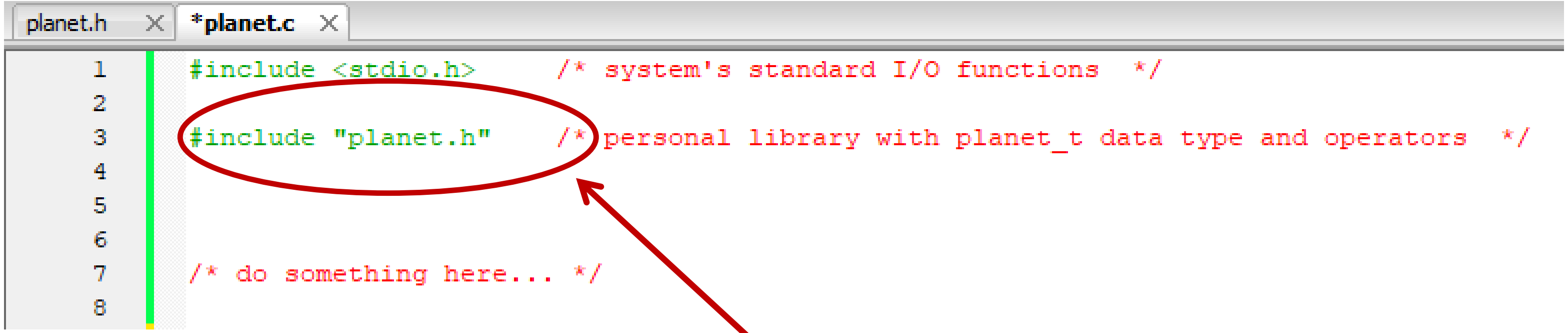


FIGURE 12.4 (continued)

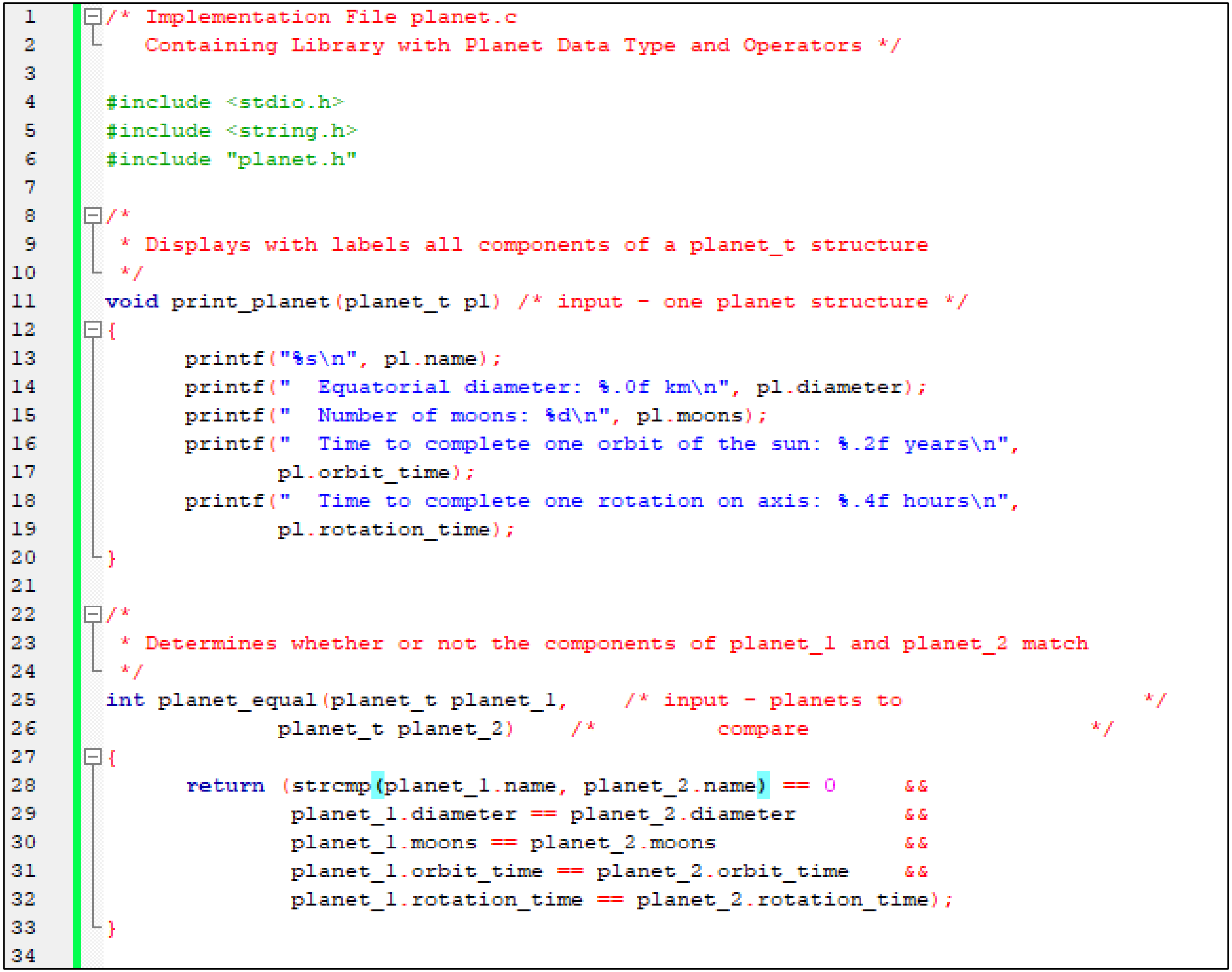
```
 *  Fills a type planet_t structure with input data. Integer returned as
 *  function result is success/failure/EOF indicator.
 *      1 => successful input of planet
 *      0 => error encountered
 *      EOF => insufficient data before end of file
 *  In case of error or EOF, value of type planet_t 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);

}
```



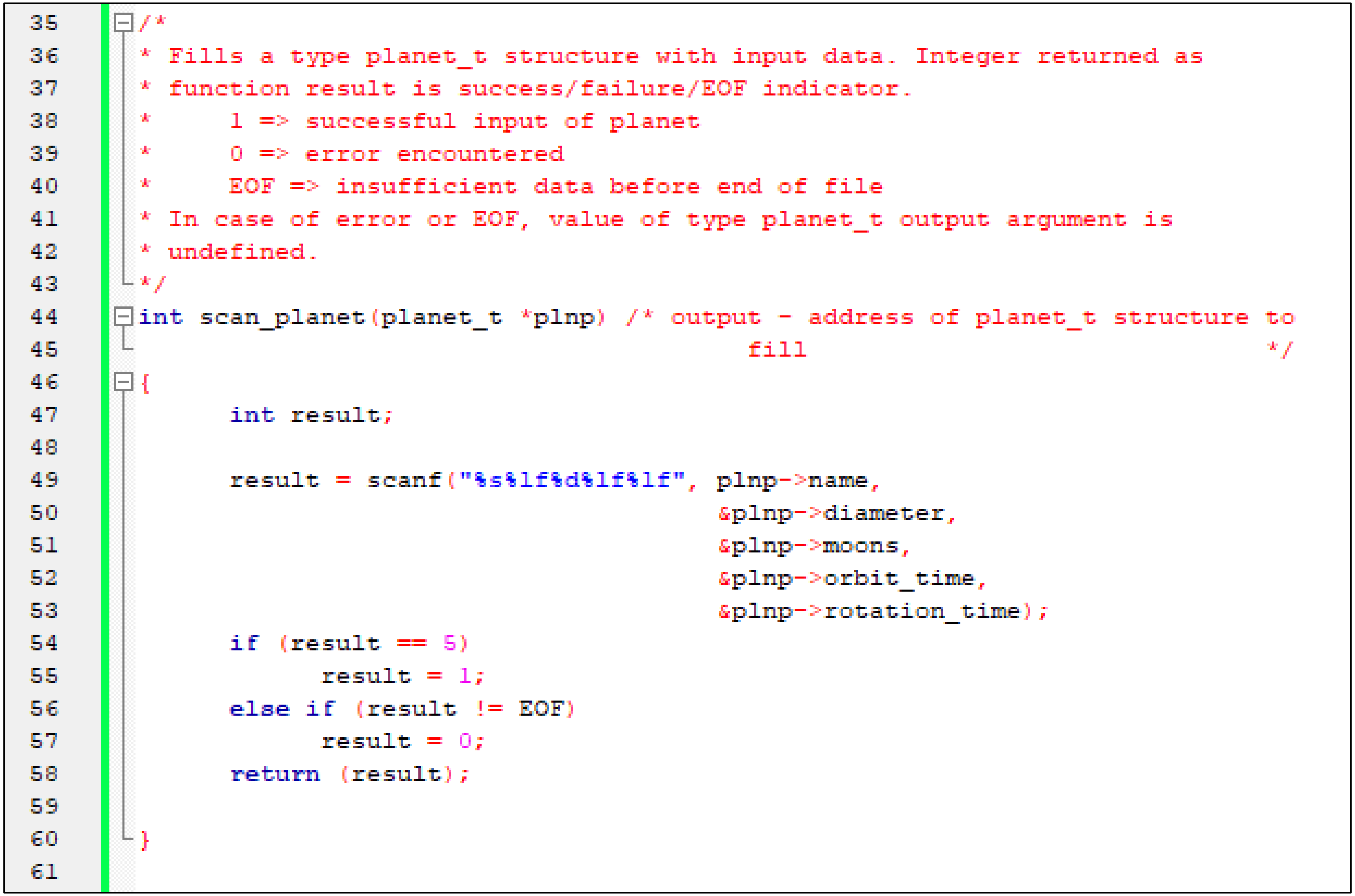
```
#include <stdio.h>      /* system's standard I/O functions  */

#include "planet.h"     /* personal library with planet_t data type and operators  */



/* do something here... */

```

This library belongs to the programmer. Instead of <>, use " "

When revising a source file, the C preprocessor replaces each #include line with the contents of the header file it references.



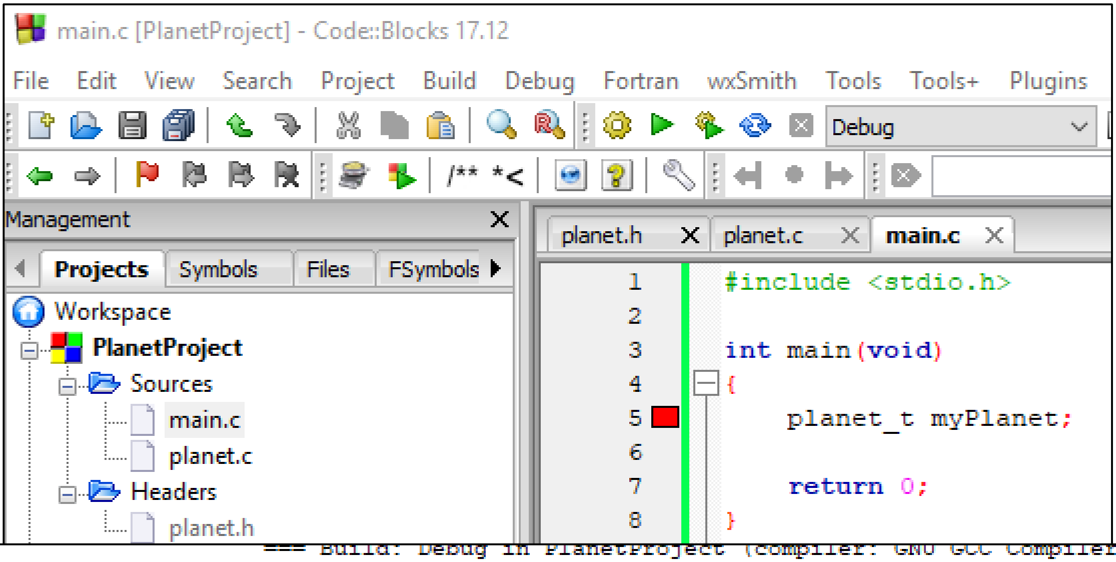
```c
/* Implementation File planet.c
   Containing Library with Planet Data Type and Operators */

#include <stdio.h>
#include <string.h>
#include "planet.h"

/*
 * 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);
}

/*
 * 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);
}
```



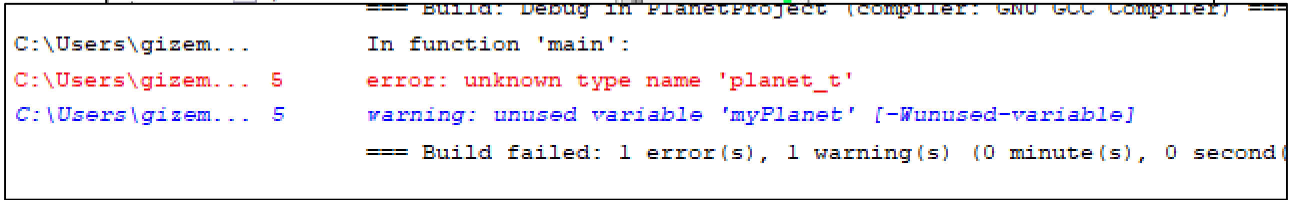
```
/*
* Fills a type planet_t structure with input data. Integer returned as
* function result is success/failure/EOF indicator.
*       1 => successful input of planet
*       0 => error encountered
*       EOF => insufficient data before end of file
* In case of error or EOF, value of type planet_t 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);

}
```

Library Basics

- Insert a main.c file to your project
- planet.c includes planet.h
- All 3 files are in the same folder and the same project
- main function is defined in main.c, not in planet.c

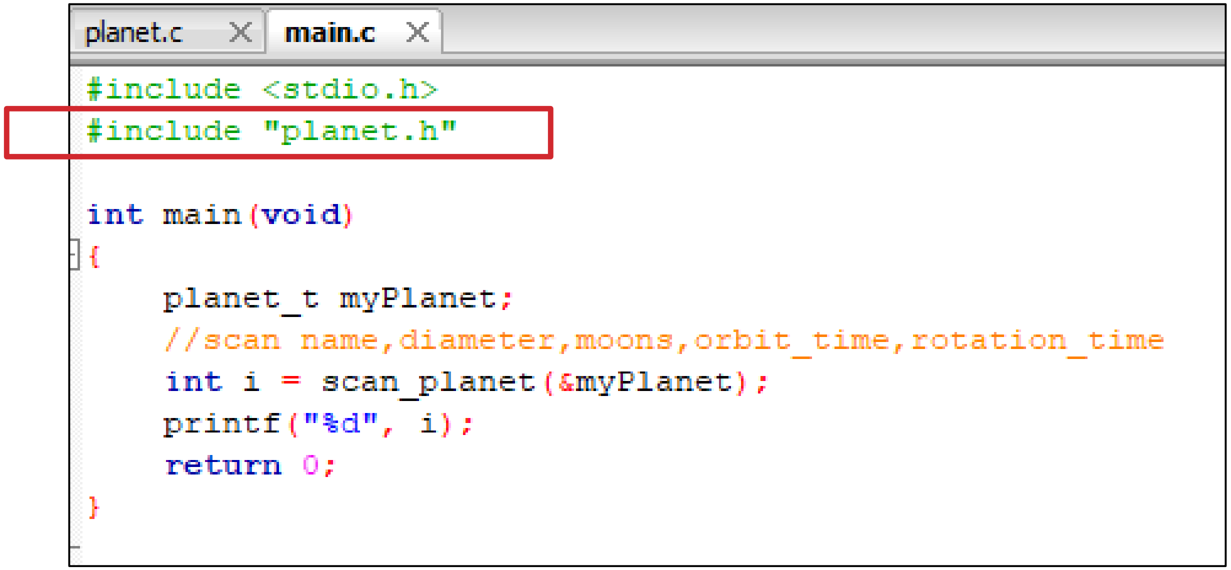


```c
#include <stdio.h>

int main(void)
{
    planet_t myPlanet;

    return 0;
}
```

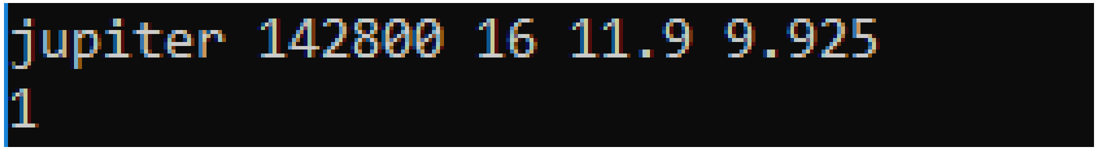


```
                          === Build: Debug in PlanetProject (compiler: GNU GCC Compiler) ===
C:\Users\gizem...         In function 'main':
C:\Users\gizem...  5      error: unknown type name 'planet_t'
C:\Users\gizem...  5      warning: unused variable 'myPlanet' [-Wunused-variable]
                          === Build failed: 1 error(s), 1 warning(s) (0 minute(s), 0 second(
```

Library Basics



```c
#include <stdio.h>
#include "planet.h"

int main(void)
{
    planet_t myPlanet;
    //scan name,diameter,moons,orbit_time,rotation_time
    int i = scan_planet(&myPlanet);
    printf("%d", i);
    return 0;
}
```



```
jupiter 142800 16 11.9 9.925
1
```

Storage Class Specifiers

- There are four storage class specifiers that you can prepend to your variable declarations which change how the variables are stored in memory:
  - **auto, extern, register, and static.**

Storage Classes

- auto
  - default storage class of function parameters and local variables
  - storage is automatically allocated on the stack at the time of a function call and deallocated when the function returns
- extern
  - storage class of names known to the linker
  - `extern` *`prototype`* does not create a function of storage class extern, but notifies the compiler that such a function exists and that the linker will know where to find it.

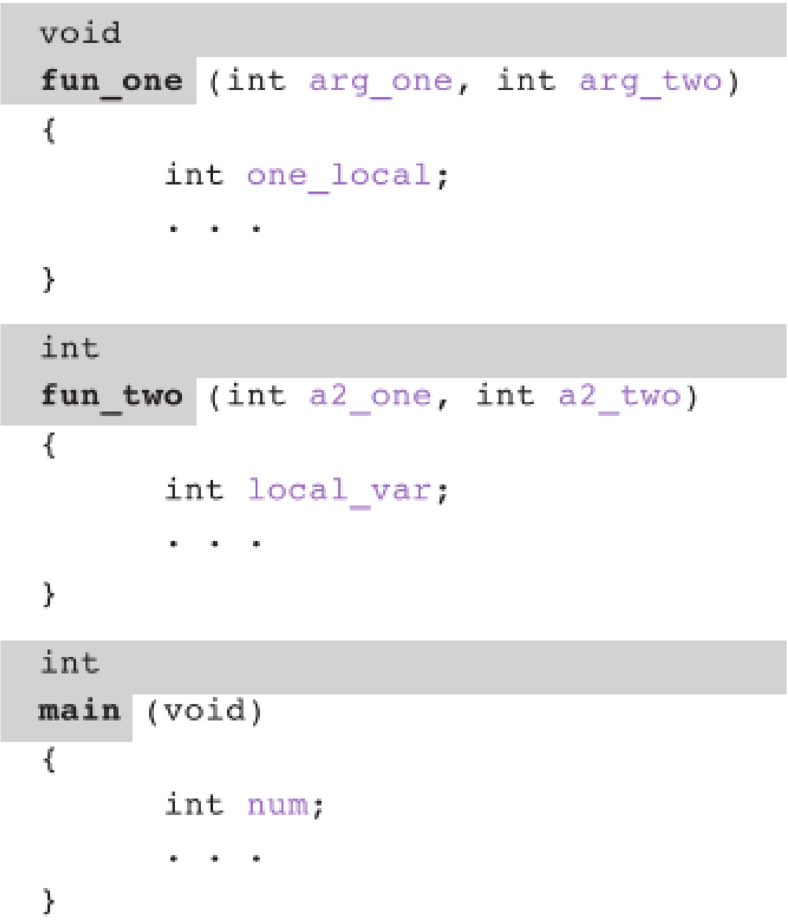
Storage Classes

- **global variable**
  - a variable that may be accessed by many functions in a program

- It is possible (though usually inadvisable) to declare variables at the top level.
- The scope of such a variable name extends from the point of declaration to the end of the source file, except in functions where the same name is declared as a formal parameter or local variable.

- !! If we need to reference a top-level variable in the region of its source file that;
  - precedes its declaration or
  - in another source file,
- the compiler can be alerted to the variable’s existence by placing a declaration of the variable that begins with the keyword **extern**

FIGURE 12.5 Storage Classes auto and extern as Previously Seen

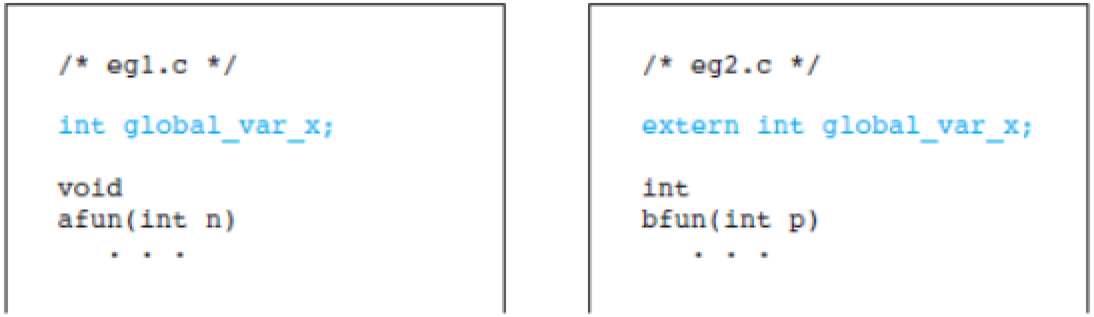


```
void
fun_one (int arg_one, int arg_two)
{
        int one_local;
        . . .
}

int
fun_two (int a2_one, int a2_two)
{
        int local_var;
        . . .
}

int
main (void)
{
        int num;
        . . .
}
```

purple: storage class auto
shaded: storage class extern

FIGURE 12.6

Declaration of a Global Variable



```
/* eg1.c */

int global_var_x;

void
afun(int n)
   . . .
```

```
/* eg2.c */

extern int global_var_x;

int
bfun(int p)
   . . .
```

Figure 12.6 shows the declaration at the top level of int variable global_var_x of storage class extern in file eg1.c and an extern statement in eg2.c that makes the global variable accessible throughout this file as well.

!!!! Only the *defining declaration,* the one in eg1.c , allocates space for global_var_x .
A declaration beginning with the keyword extern allocates no memory; it simply provides information for the compiler.

```c
/* fileone.c */

typedef struct {
        double real,
               imag;
} complex_t;

/* Defining declarations of
   global structured constant
   complex_zero and of global
   constant array of month
   names */
const complex_t complex_zero
      = {0, 0};
const char *months[12] =
      {"January", "February",
       "March", "April", "May",
       "June", "July", "August",
       "September", "October",
       "November", "December"};

int
f1_fun1(int n)
{ . . . }

double
f1_fun2(double x)
{ . . . }

char
f1_fun3(char c1, char c2)
{   double months; . . . }
```

```c
/* filetwo.c */

/* #define's and typedefs
    including  complex_t */


void
f2_fun1(int x)
{ . . . }

/* Compiler-notifying
   declarations -- no
   storage allocated */
extern const complex_t
       complex_zero;
extern const char
       *months[12];

void
f2_fun2(void)
{ . . . }

int
f2_fun3(int n)
{ . . . }
```

Function(s)	Can Access Variables of Class Extern
f1_fun1 and f1_fun2	complex_zero and months
f1_fun3	complex_zero only
f2_fun1	none
f2_fun2 and f2_fun3	complex_zero and months

Other Storage Classes

- **static**
  - storage class of variables allocated only once, prior to program execution
    - value retains for each function call
    - `static double matrix [50][40]`

- when static is used with global variables, its scope is limited to the file
- when static is used with local variables, memory is allotted statically instead of automatically (static variables are used to preserve state and values are not lost when function execution ends)
    - `(same for global static as well, local statics can only be accessed in the function’s scope)`

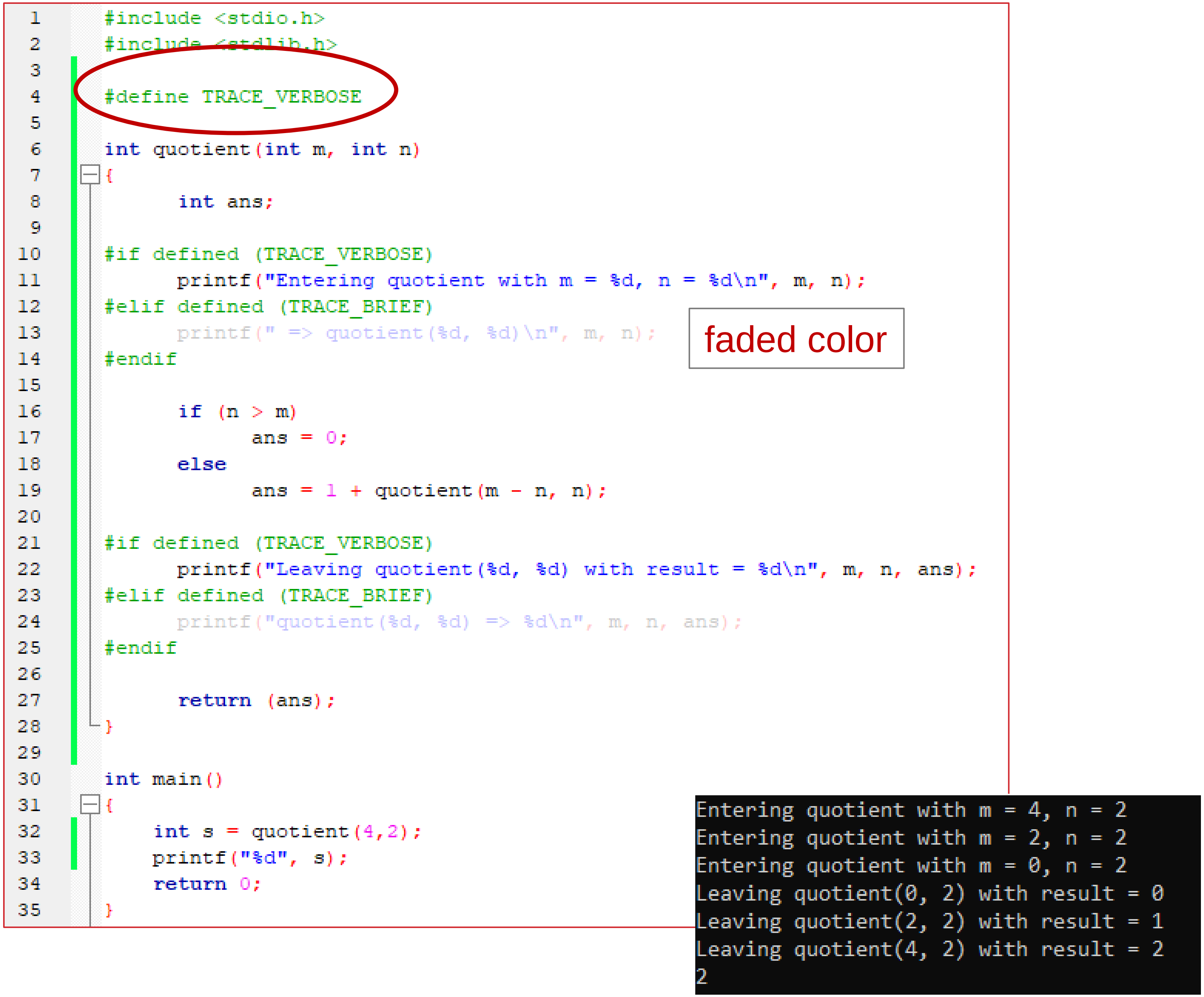
Other Storage Classes

- register
  - storage class of automatic variables that the programmer would like to have store in registers
  - used for frequently called variables (local variables only) to keep them in registers (high speed memory location inside the CPU)
    - e.g. variables serving as subscripts of large arrays
      - `register int row, col;`

Conditional Compilation

- C’s preprocessor recognizes commands that allow the user to select parts of a program to be compiled and parts to be omitted.

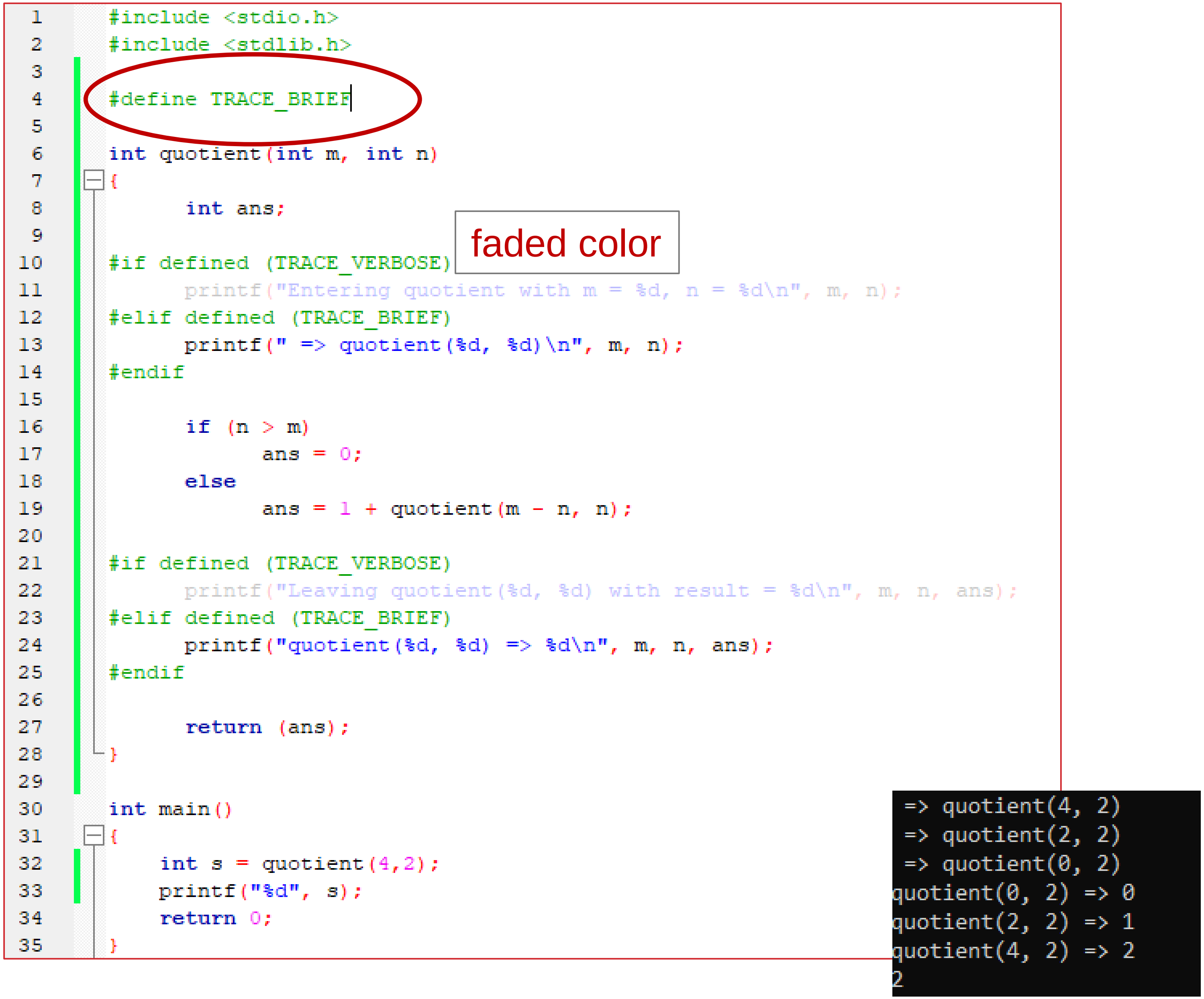
- This ability can be helpful in a variety of situations.
  - For example, one can build in debugging printf calls when writing a function and then include these statements in the compiled program only when they are needed.
  - Inclusion of header files is another activity that may need to be done conditionally.



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

#define TRACE_VERBOSE

int quotient(int m, int n)
{
        int ans;

#if defined (TRACE_VERBOSE)
        printf("Entering quotient with m = %d, n = %d\n", m, n);
#elif defined (TRACE_BRIEF)
        printf(" => quotient(%d, %d)\n", m, n);
#endif

        if (n > m)
                ans = 0;
        else
                ans = 1 + quotient(m - n, n);

#if defined (TRACE_VERBOSE)
        printf("Leaving quotient(%d, %d) with result = %d\n", m, n, ans);
#elif defined (TRACE_BRIEF)
        printf("quotient(%d, %d) => %d\n", m, n, ans);
#endif

        return (ans);
}

int main()
{
    int s = quotient(4,2);
    printf("%d", s);
    return 0;
}
```

faded color

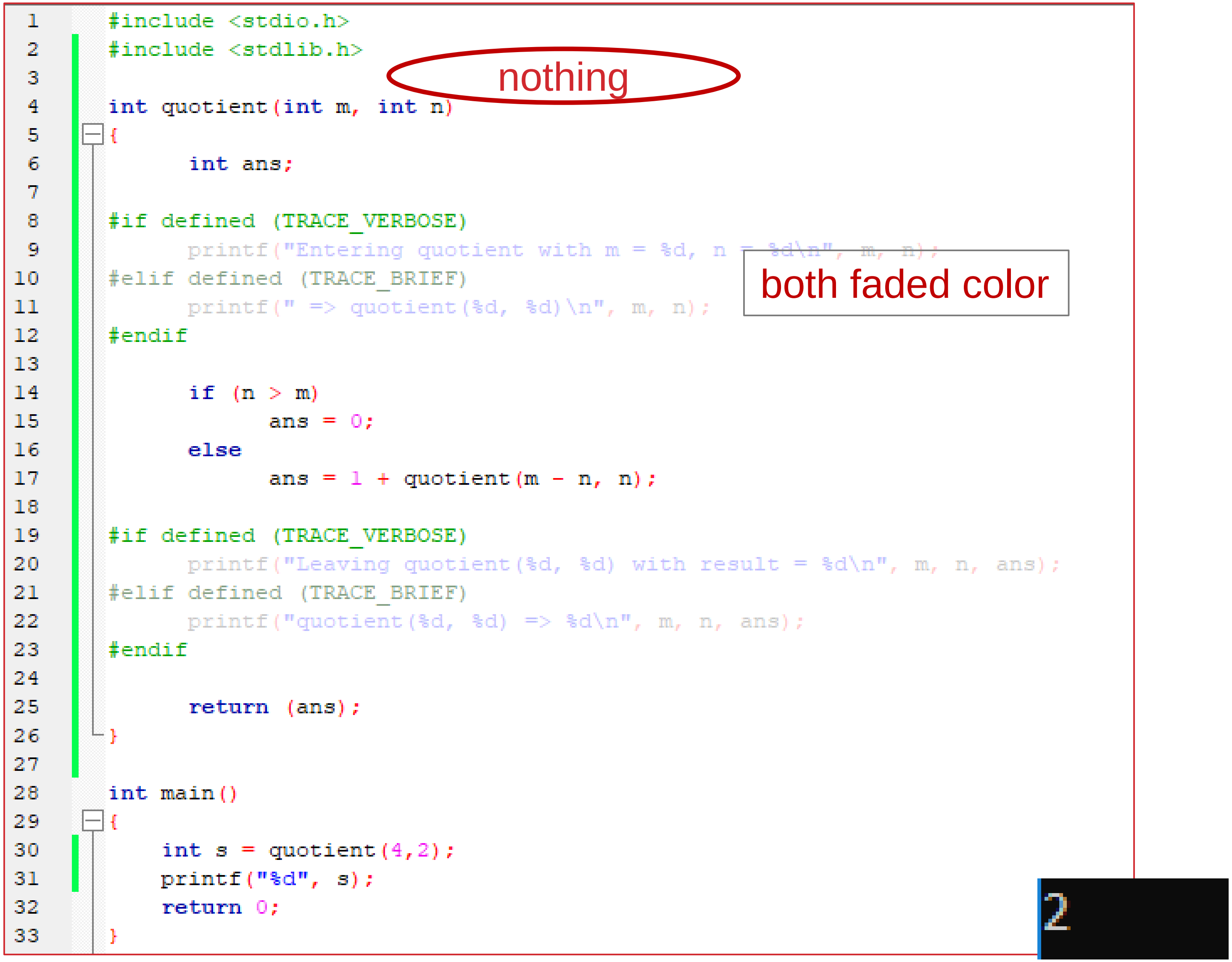
```
Entering quotient with m = 4, n = 2
Entering quotient with m = 2, n = 2
Entering quotient with m = 0, n = 2
Leaving quotient(0, 2) with result = 0
Leaving quotient(2, 2) with result = 1
Leaving quotient(4, 2) with result = 2
2
```



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

#define TRACE_BRIEF

int quotient(int m, int n)
{
	int ans;

#if defined (TRACE_VERBOSE)
	printf("Entering quotient with m = %d, n = %d\n", m, n);
#elif defined (TRACE_BRIEF)
	printf(" => quotient(%d, %d)\n", m, n);
#endif

	if (n > m)
		ans = 0;
	else
		ans = 1 + quotient(m - n, n);

#if defined (TRACE_VERBOSE)
	printf("Leaving quotient(%d, %d) with result = %d\n", m, n, ans);
#elif defined (TRACE_BRIEF)
	printf("quotient(%d, %d) => %d\n", m, n, ans);
#endif

	return (ans);
}

int main()
{
    int s = quotient(4,2);
    printf("%d", s);
    return 0;
}
```

faded color

```
 => quotient(4, 2)
 => quotient(2, 2)
 => quotient(0, 2)
quotient(0, 2) => 0
quotient(2, 2) => 1
quotient(4, 2) => 2
2
```



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

int quotient(int m, int n)
{
        int ans;

#if defined (TRACE_VERBOSE)
        printf("Entering quotient with m = %d, n = %d\n", m, n);
#elif defined (TRACE_BRIEF)
        printf(" => quotient(%d, %d)\n", m, n);
#endif

        if (n > m)
                ans = 0;
        else
                ans = 1 + quotient(m - n, n);

#if defined (TRACE_VERBOSE)
        printf("Leaving quotient(%d, %d) with result = %d\n", m, n, ans);
#elif defined (TRACE_BRIEF)
        printf("quotient(%d, %d) => %d\n", m, n, ans);
#endif

        return (ans);
}

int main()
{
    int s = quotient(4,2);
    printf("%d", s);
    return 0;
}
```

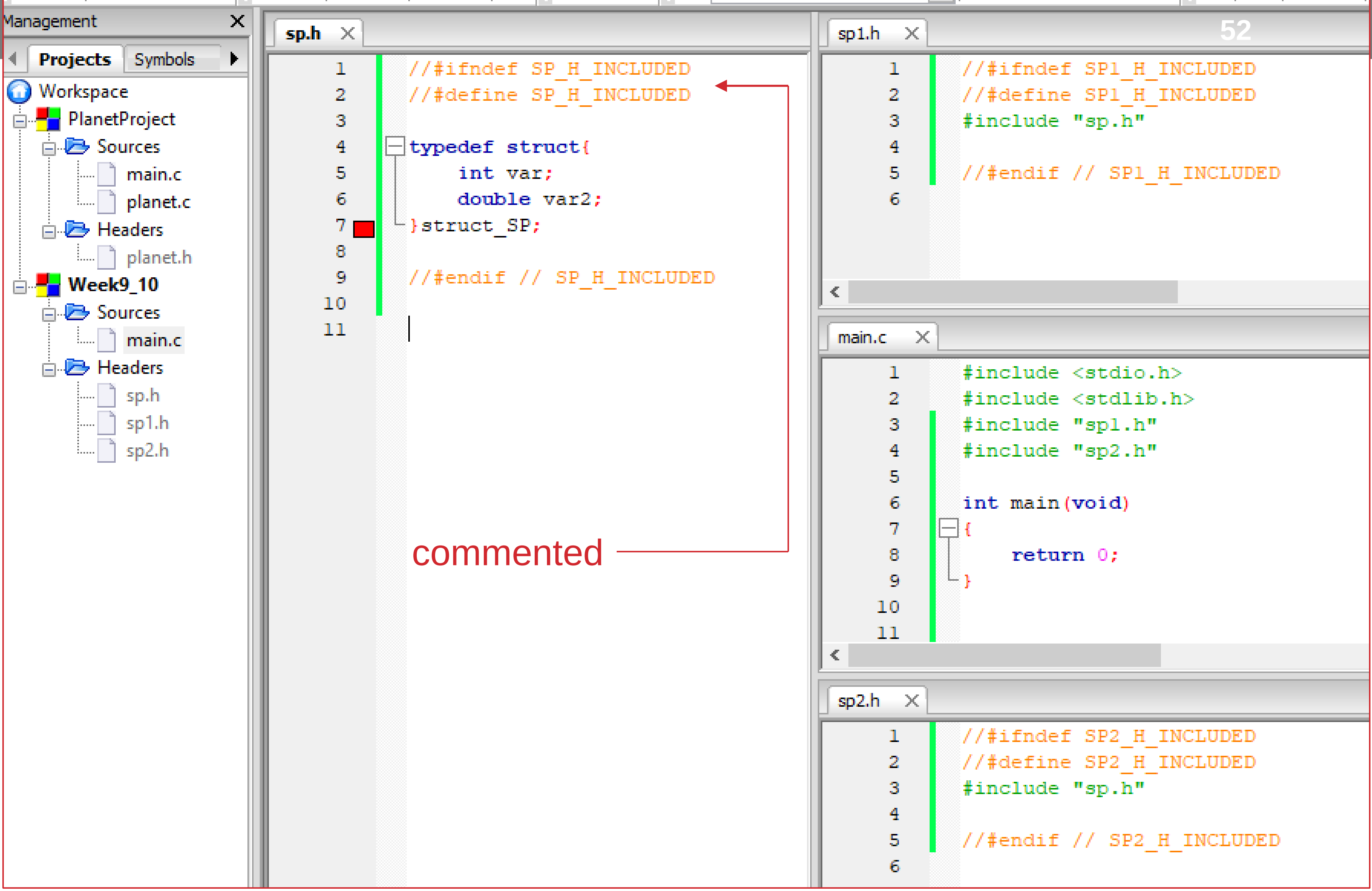
```
2
```

nothing

both faded color

Multiple Inclusion Problem

- Assume that you have 3 header files: sp.h, sp1.h, sp2.h
  - sp1.h and sp2.h both include sp.h
  - our main function should use the facilities of both sp1.h and sp2.h
  - leads to duplicate declarations of sp.h data types and functions



sp.h

```
//#ifndef SP_H_INCLUDED
//#define SP_H_INCLUDED

typedef struct{
    int var;
    double var2;
}struct_SP;

//#endif // SP_H_INCLUDED

```

commented

sp1.h

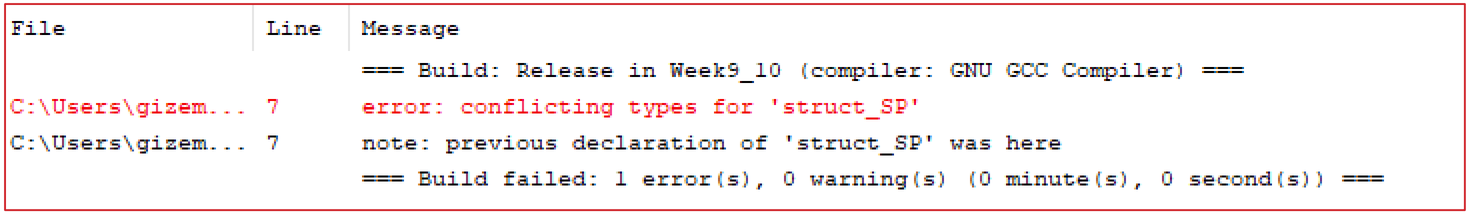
```
//#ifndef SP1_H_INCLUDED
//#define SP1_H_INCLUDED
#include "sp.h"

//#endif // SP1_H_INCLUDED
```

main.c

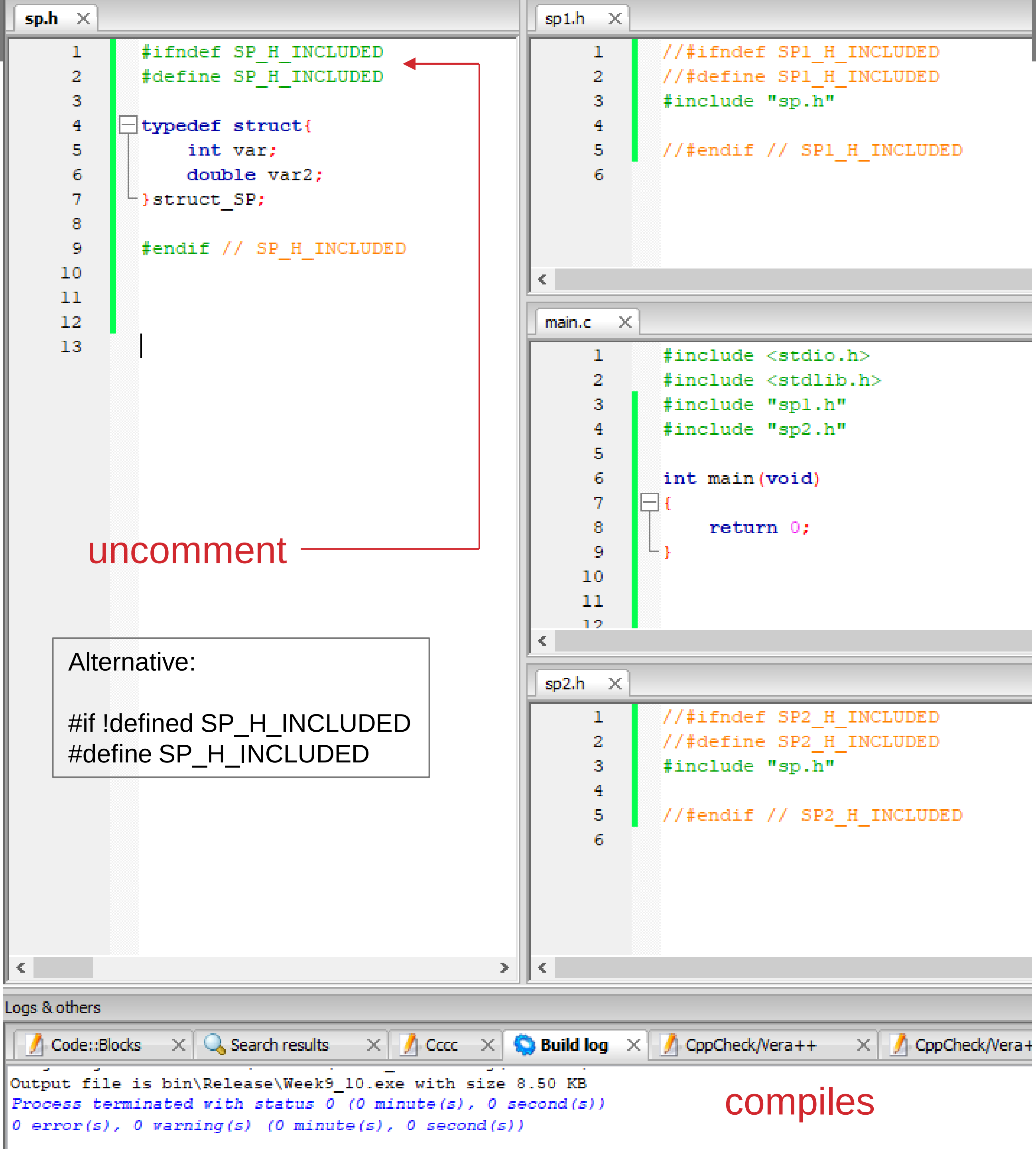
```
#include <stdio.h>
#include <stdlib.h>
#include "sp1.h"
#include "sp2.h"

int main(void)
{
    return 0;
}
```

sp2.h

```
//#ifndef SP2_H_INCLUDED
//#define SP2_H_INCLUDED
#include "sp.h"

//#endif // SP2_H_INCLUDED
```



```
File                 Line   Message
                            === Build: Release in Week9_10 (compiler: GNU GCC Compiler) ===
C:\Users\gizem...    7      error: conflicting types for 'struct_SP'
C:\Users\gizem...    7      note: previous declaration of 'struct_SP' was here
                            === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```



sp.h

```c
#ifndef SP_H_INCLUDED
#define SP_H_INCLUDED

typedef struct{
    int var;
    double var2;
}struct_SP;

#endif // SP_H_INCLUDED




```

uncomment

Alternative:

#if !defined SP_H_INCLUDED
#define SP_H_INCLUDED

sp1.h

```c
//#ifndef SP1_H_INCLUDED
//#define SP1_H_INCLUDED
#include "sp.h"

//#endif // SP1_H_INCLUDED

```

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "sp1.h"
#include "sp2.h"

int main(void)
{
    return 0;
}



```

sp2.h

```c
//#ifndef SP2_H_INCLUDED
//#define SP2_H_INCLUDED
#include "sp.h"

//#endif // SP2_H_INCLUDED

```

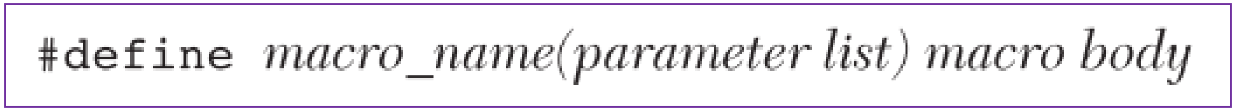
```
Output file is bin\Release\Week9_10.exe with size 8.50 KB
Process terminated with status 0 (0 minute(s), 0 second(s))
0 error(s), 0 warning(s) (0 minute(s), 0 second(s))
```

compiles

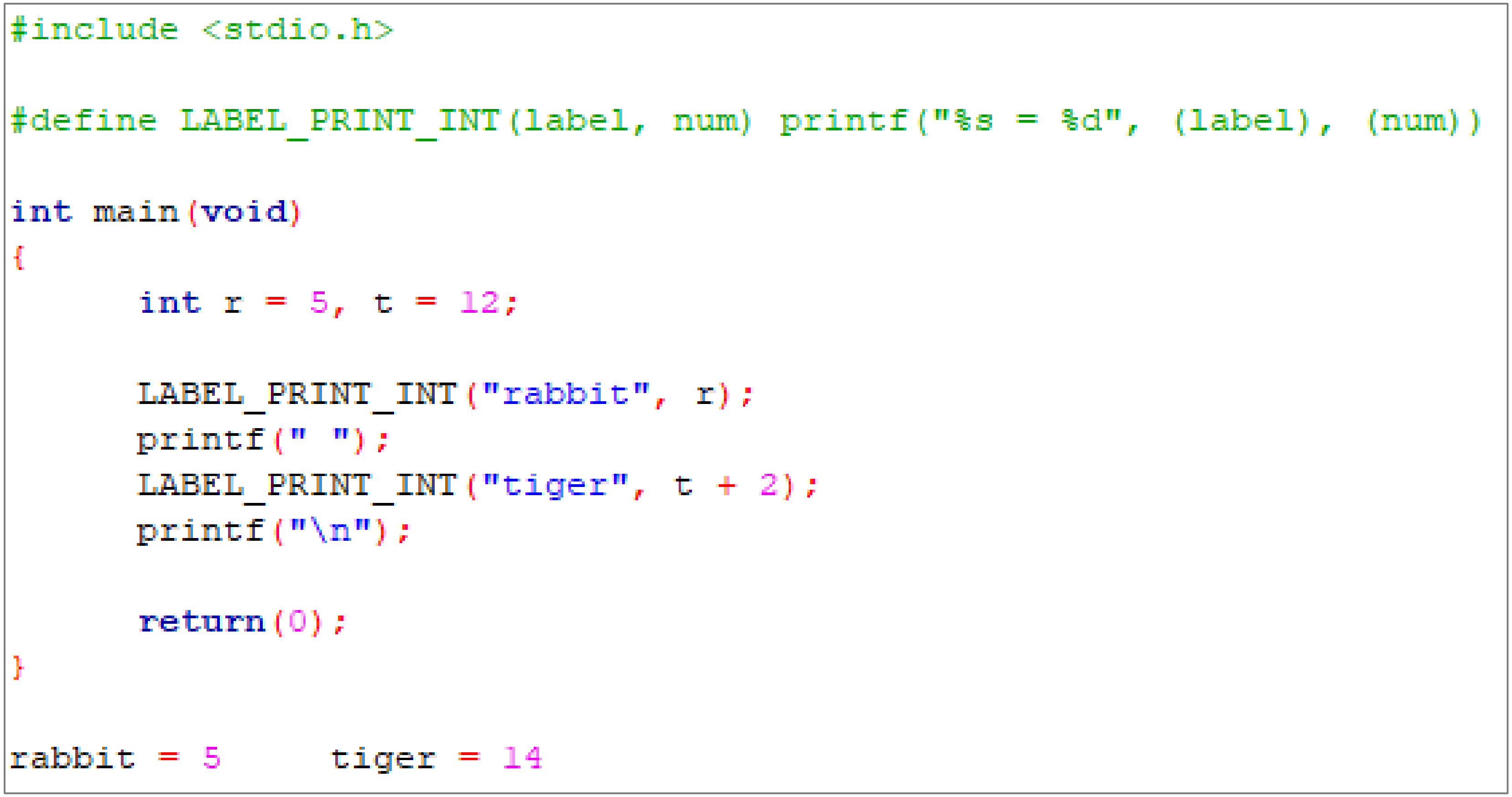
Defining Macros with Parameters

- macro
  - facility for naming a commonly used statement or operation

- macro expansion
  - process of replacing a macro call by its meaning

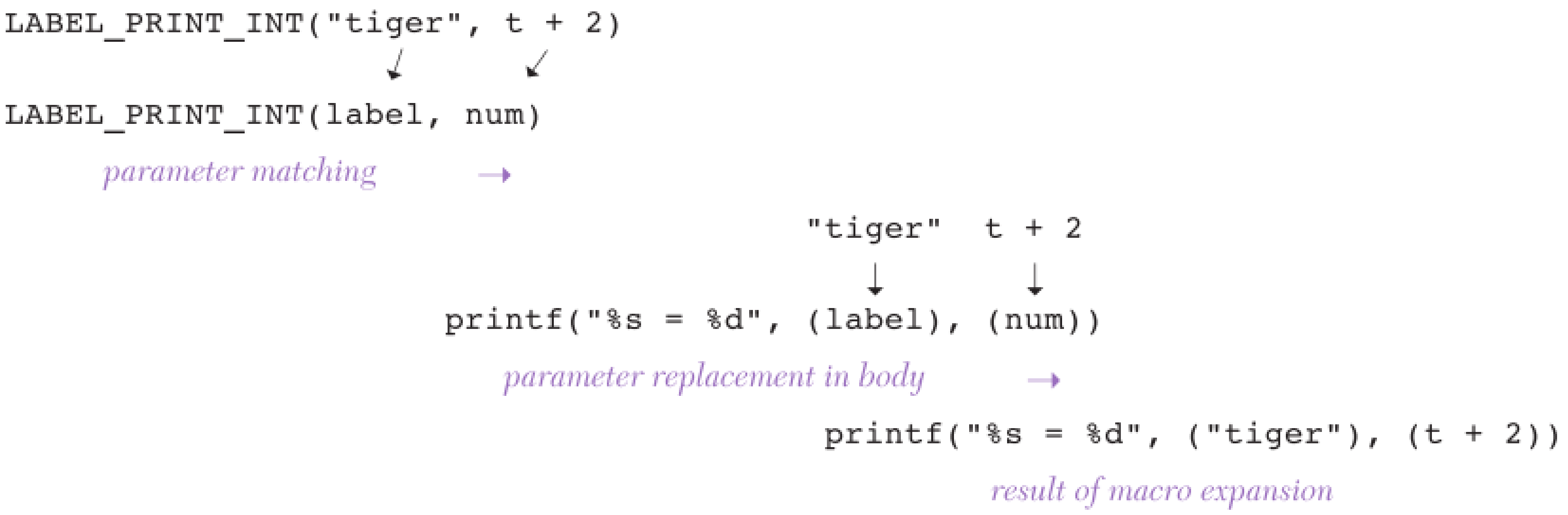


`#define` *macro_name(parameter list) macro body*



```
#include <stdio.h>

#define LABEL_PRINT_INT(label, num) printf("%s = %d", (label), (num))

int main(void)
{
      int r = 5, t = 12;

      LABEL_PRINT_INT("rabbit", r);
      printf(" ");
      LABEL_PRINT_INT("tiger", t + 2);
      printf("\n");

      return(0);
}

rabbit = 5     tiger = 14
```

FIGURE 12.14 Macro Expansion of Second Macro Call of Program in Fig. 12.13



```
LABEL_PRINT_INT("tiger", t + 2)
```

```
LABEL_PRINT_INT(label, num)
```

parameter matching →

```
"tiger"  t + 2
```

```
printf("%s = %d", (label), (num))
```

parameter replacement in body →

```
printf("%s = %d", ("tiger"), (t + 2))
```

result of macro expansion

#define Preprocessor Directive: Macros

- [*Note:* A symbolic constant is a type of macro.]
- Consider the following *macro definition* with one *argument* for the area of a circle:

```
#define CIRCLE_AREA(x) ((PI) * (x) * (x))
```

- Wherever CIRCLE_AREA(y) appears in the file, the value of y is substituted for x in the replacement-text, the symbolic constant PI is replaced by its value (defined previously) and the macro is expanded in the program.

#define Preprocessor Directive: Macros

- For example, the statement
  - area = **CIRCLE_AREA(4);**

  is expanded to
  - area = ((**3.14159) * (4) * (4));**

  then, at compile time, the value of the expression is evaluated and assigned to variable area.
- The *parentheses* around each x in the replacement text *force the proper order of evaluation when the macro argument is an expression*.

#define Preprocessor Directive: Macros

- For example, the statement
    - area = **CIRCLE_AREA**(c + 2);

  is expanded to
    - area = ((**3.14159)** * (c + **2)** * (c + **2**));

  which evaluates *correctly* because the parentheses force the proper order of evaluation.

#define Preprocessor Directive: Macros

- If the parentheses in the macro definition are omitted, the macro expansion is
  - area = **3.14159** * c + **2** * c + **2**;

  which evaluates *incorrectly* as
  - area = (**3.14159** * c) + (**2** * c) + **2**;

  because of the rules of operator precedence.

#define Preprocessor Directive: Macros

- Macro CIRCLE_AREA could be defined more safely as a function.
- Function circleArea

  -
    ```
    double circleArea(double x)
    {
       return 3.14159 * x * x;
    }
    ```

performs the same calculation as macro CIRCLE_AREA

Wrap Up

- C’s facility for creating a personal library provides a means of encapsulating an abstract data type.
- Dividing a library definition into a header file and an implementation file provides a natural separation of the description of what the library functions do from how they do it.
- Defining a macro gives a name to a frequently used statement or operation.
- The exit function allows premature termination of program execution.

Wrap Up

- Conditional compilation provides a means of customizing code for different implementations and of creating library header files that protect themselves from duplicate inclusion.
- Designing function main with parameters argc and argv allows the use of command-line arguments.
- Library functions must have meaningful names, have clearly defined interfaces, and be as independent as possible from globally defined constants.

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.