Computer Programming

Programming in the Large

Large Software Systems

- Abstraction
  - Procedural abstraction
  - Data abstraction
  - Information hiding
  - Code reuse
- Tools to reduce complexity
  - Using more than one files
    - Header files and function libraries
    - global variables and extern storage class
  - Conditional compilation
  - Arguments to function main
  - Macros

Abstraction to Manage Complexity

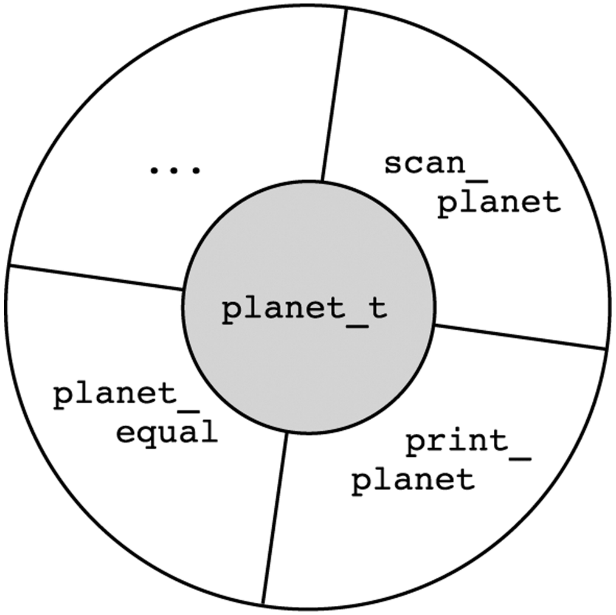
- Procedural Abstraction
  - Break down the problem into solvable chunks
    - Functional decomposition
  - Separate what is to e achieved from the details of how to be achieved
    - Ex: We use function fopen without knowing how it performs the job. We only need to know its parameters

Abstraction to Manage Complexity

- Data abstraction
  - Describe what information is stored without specifying how the information is organized in memory
  - Logical view vs physical view
  - Ex: double

Abstraction to Manage Complexity

- Information Hiding
  - Other modules only access the data through its operators
    - Internal implementation is hidden
    - Implementation can be changed

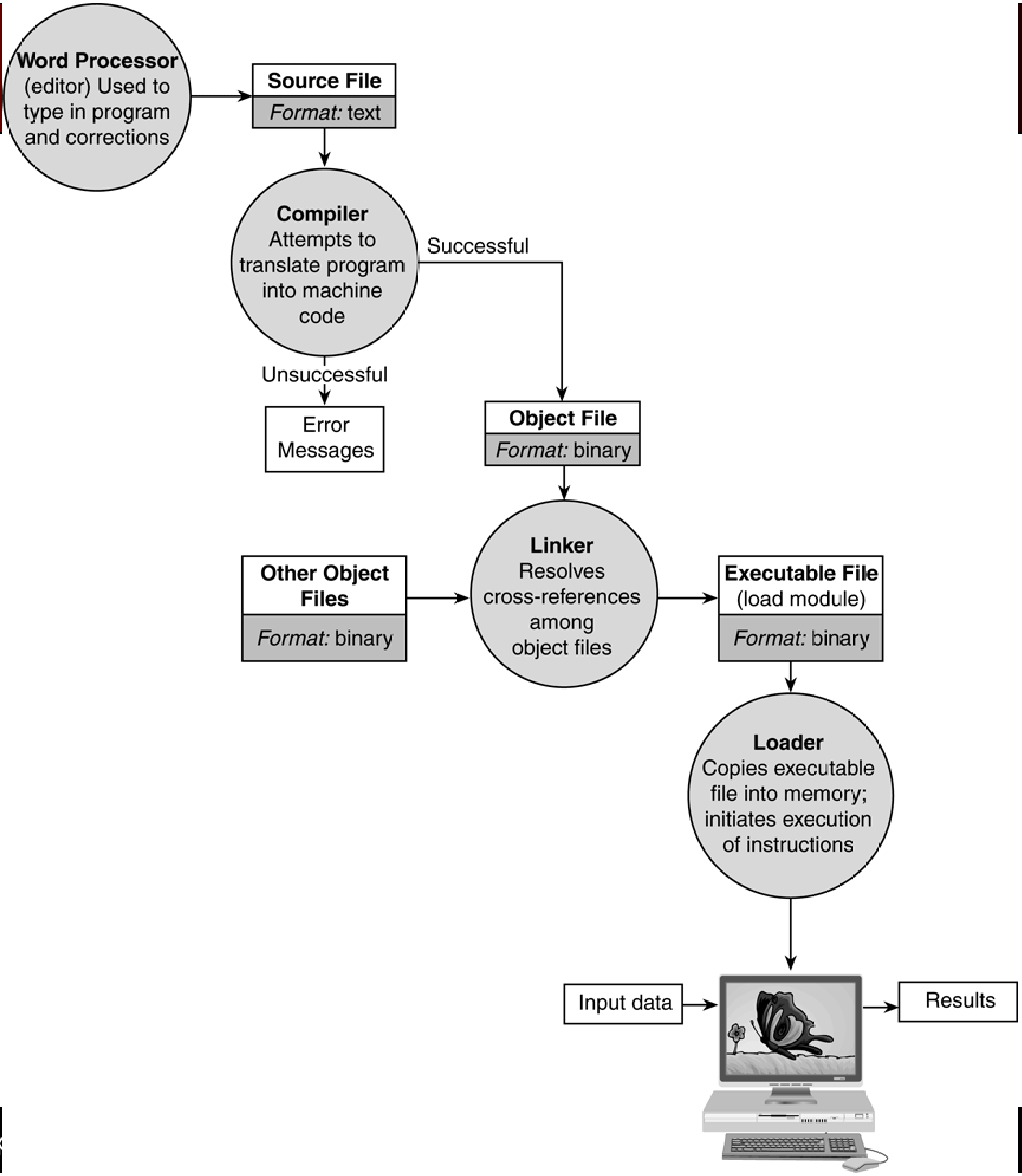


Abstraction to Manage Complexity

- Reusable code:
  - Code can be used in many applications
  - One way: **encapsulate** data and its operations in a library

Personal Libraries

- Standard libraries are very usable
- Personal libraries extends this
  - Provides abstraction
- Two files:
  - Header files: describes what the functions in the library do
  - Implementation files: shows how the functions do it



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

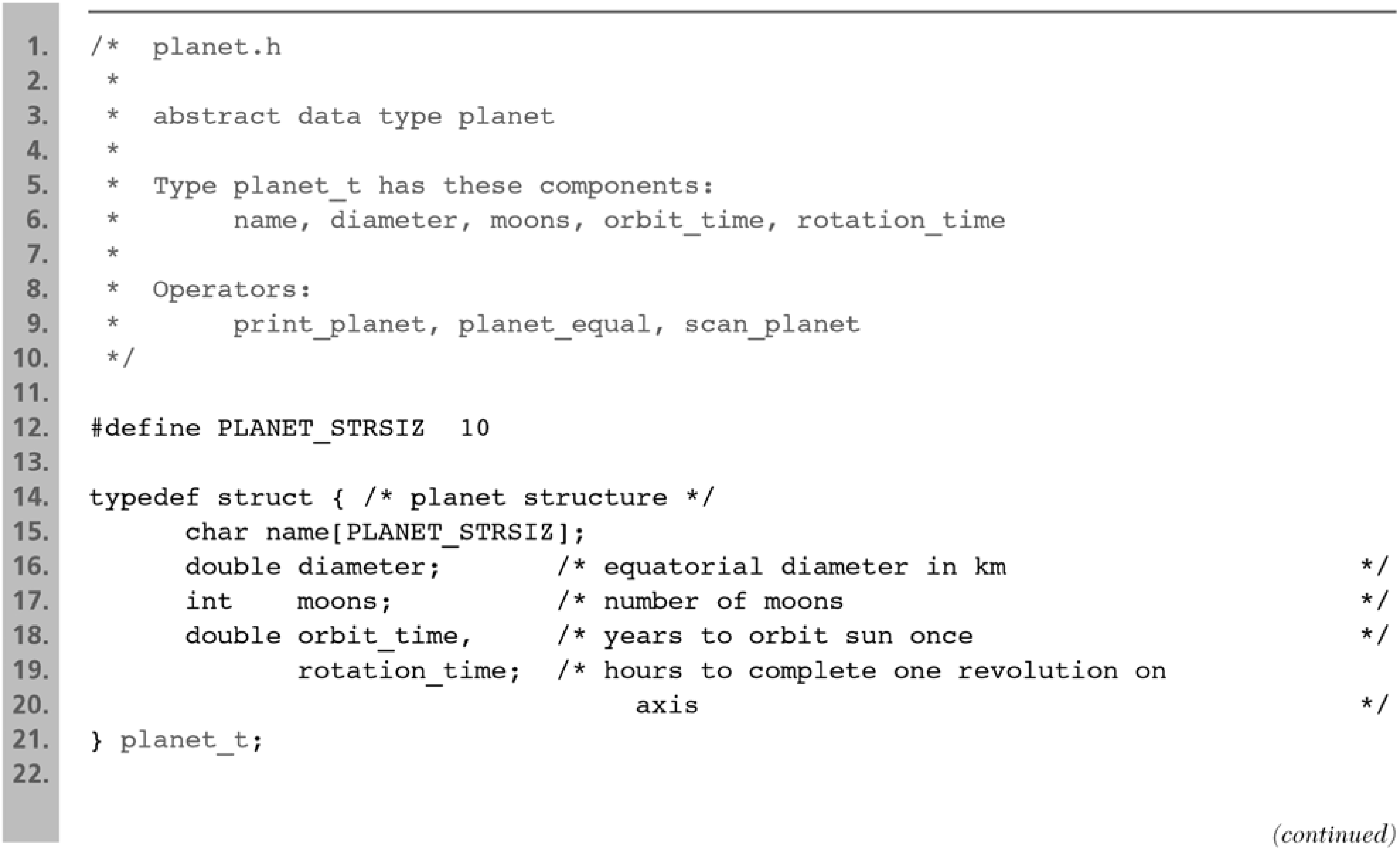
Results

Header file

- Contains
  - information about library for compilation
  - Information for programmers to use the library
- includes:
  - Macro definitions
  - Type definitions
  - Function prototypes
- Provides an interface between a library and programmer that uses the library

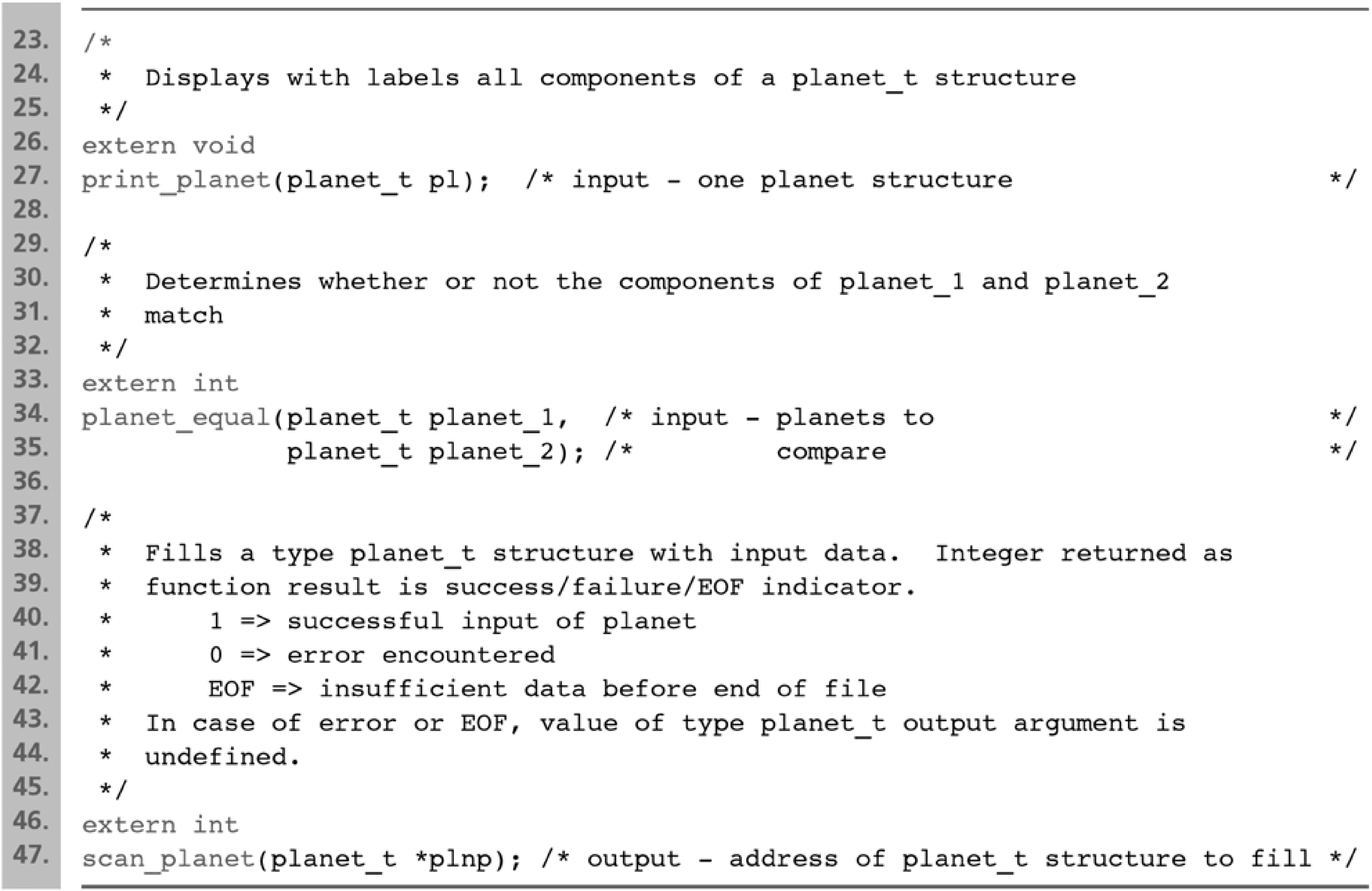
- Notes:
  - Use of extern in function prototypes
  - Use of "…." in include
  - Use of library name as a prefix in constants

Header File planet.h



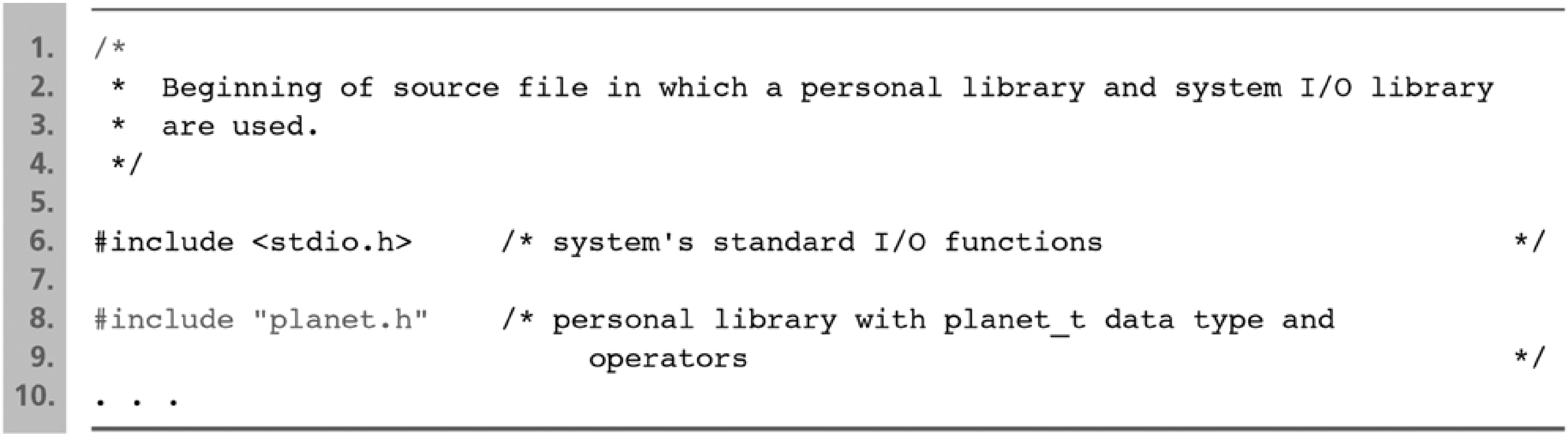
```
/*  planet.h
 *
 *  abstract data type planet
 *
 *  Type planet_t has these components:
 *       name, diameter, moons, orbit_time, rotation_time
 *
 *  Operators:
 *       print_planet, planet_equal, scan_planet
 */

#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;
```

(continued)



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

/*
 *  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 */
```

Portion of Program

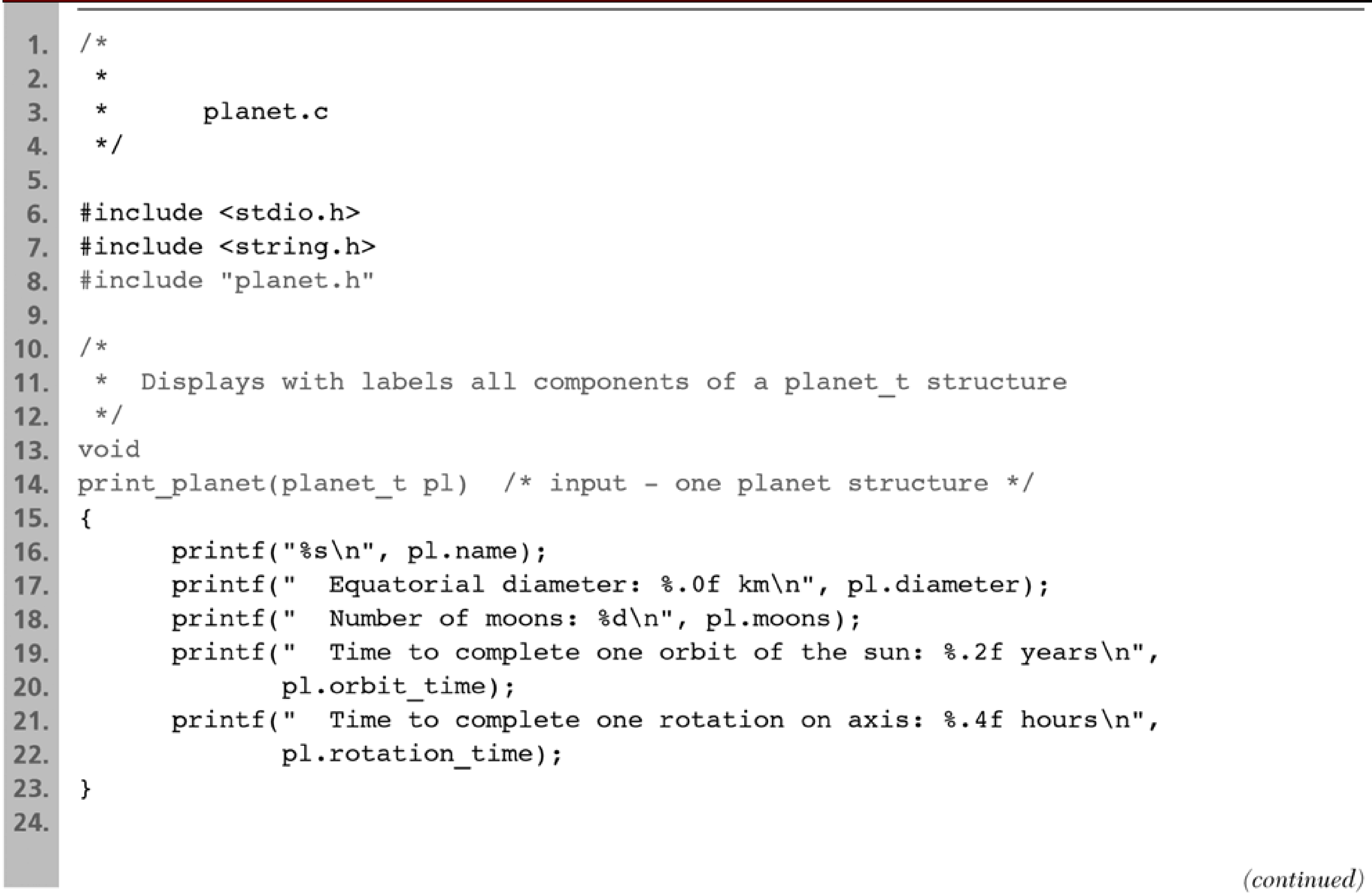


```
/*
 *  Beginning of source file in which a personal library and system I/O library
 *  are used.
 */

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

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

Implementation file

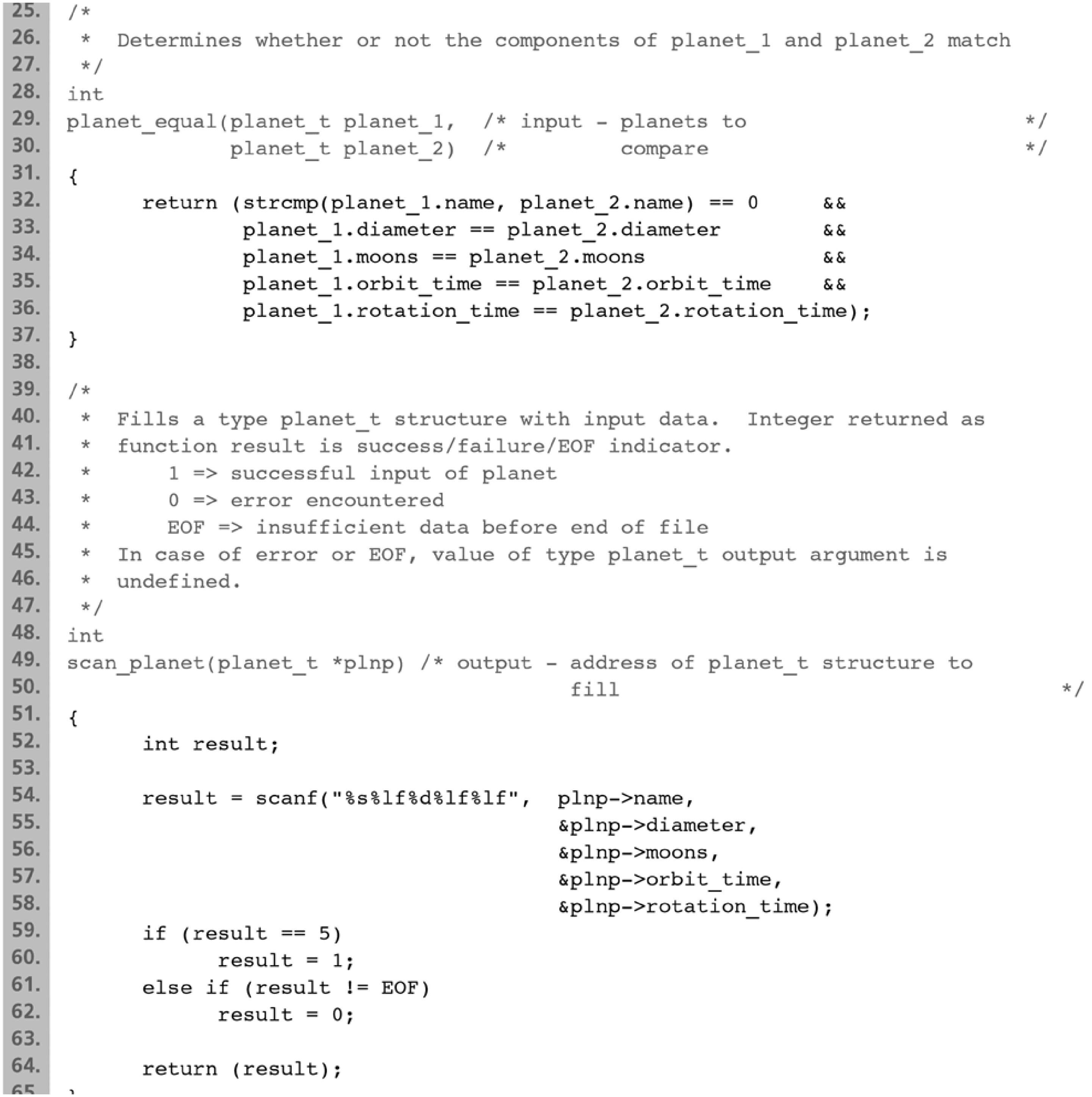
- Source file contains
  - Code of all library functions
  - Additional information for compilation of functions
- Includes
  - Comments
  - Include directives
  - Define directives needed inside the library
  - Type declarations needed inside the library
  - Function definitions

Implementation File planet.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);
}
```

(continued)

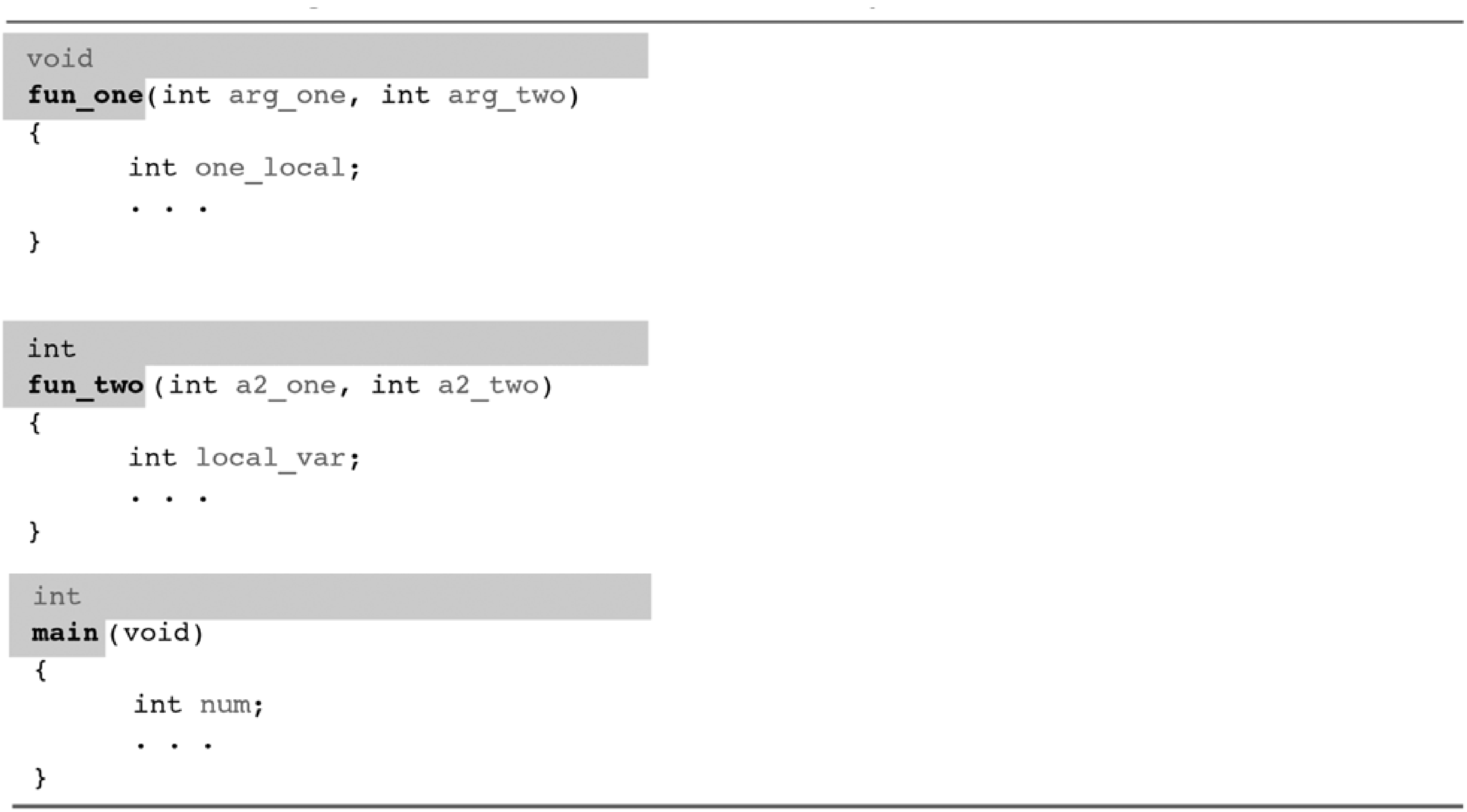


```c
/*
 *  Determines whether or not the components of planet_1 and planet_2 match
 */
int
planet_equal(planet_t planet_1,  /* input - planets to                        */
             planet_t planet_2)  /*         compare                           */
{
      return (strcmp(planet_1.name, planet_2.name)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

Storage Classes

- auto
  - Formal parameters and local variables of functions
  - Allocated on the stack and deallocated automatically
- extern
  - Names of functions
    - They are already at the top level

Storage Classes



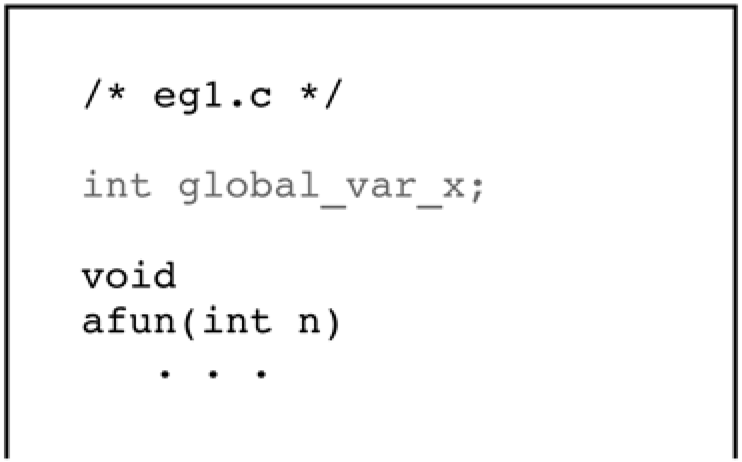
```
void
fun_one(int arg_one, int arg_two)
{
        int one_local;
        . . .
}


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

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

Colored names are auto and boldface ones are extern

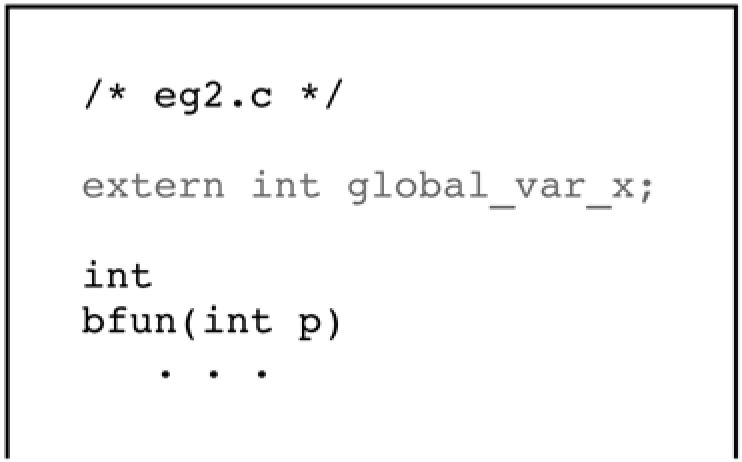
Storage Classes

- auto
  - Formal parameters and local variables of functions
  - Allocated on the stack and deallocated automatically
- extern
  - Names of functions
  - Global variables
    - Variables declared at the top level

Declaration of a Global Variable



```
/* eg1.c */

int global_var_x;

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

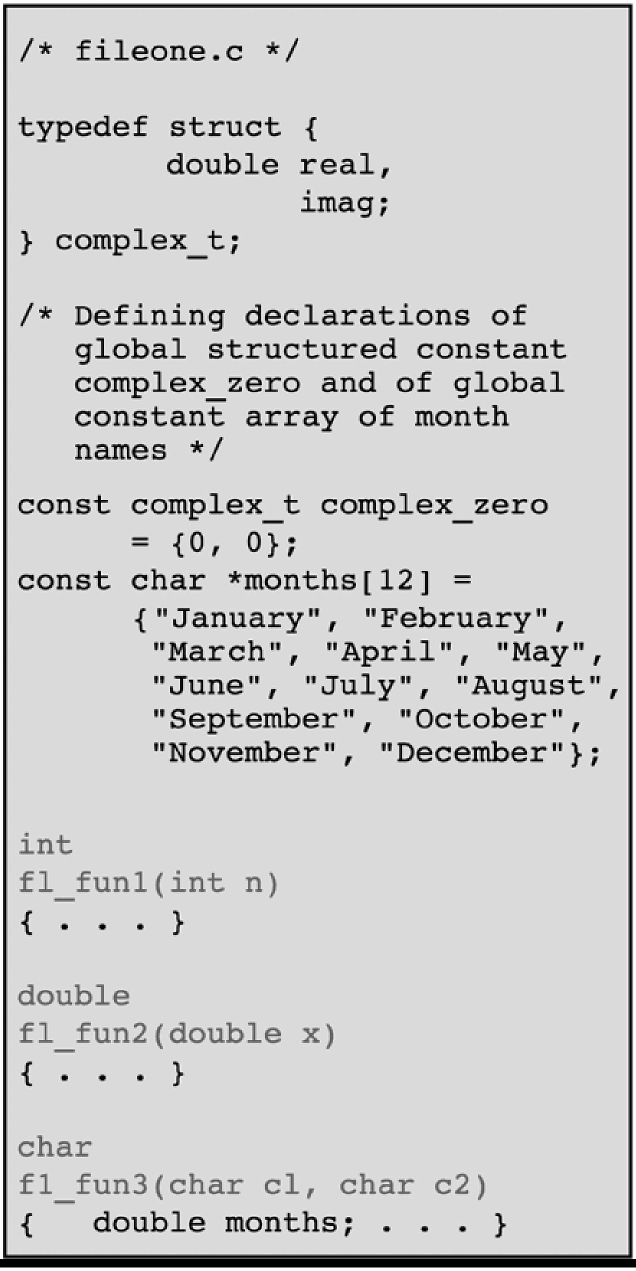


```
/* eg2.c */

extern int global_var_x;

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

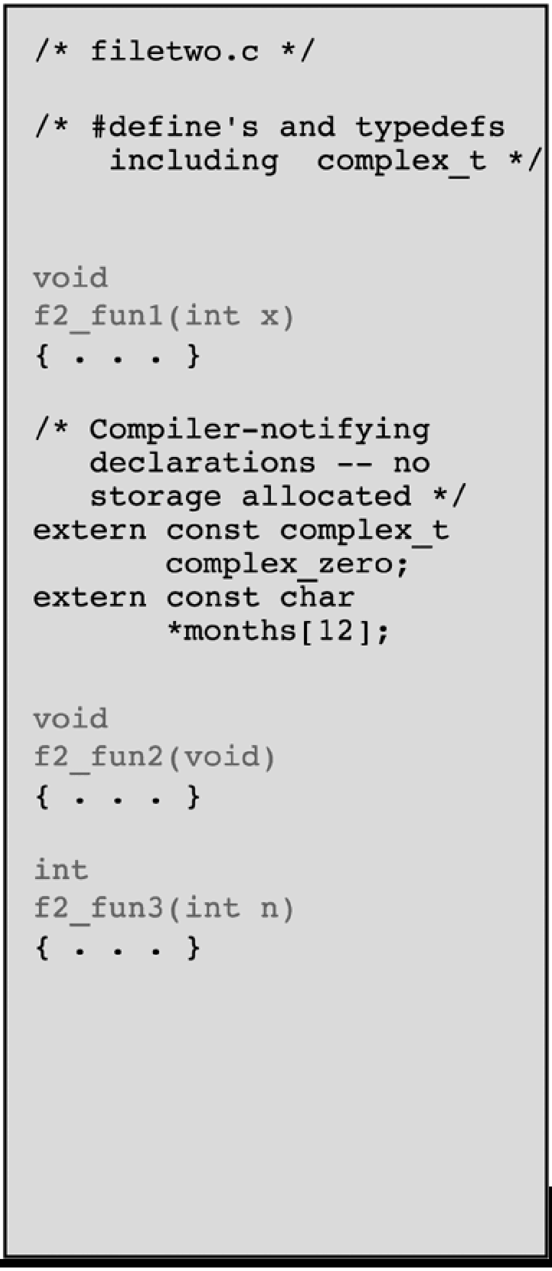
Global variables

- Should be avoided as much as possible
  - Unrestricted access
  - Reduces readability and maintainability
- Global constants are OK
  - Example in the following

Variables of Storage Class extern



```
/* 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; . . . }
```

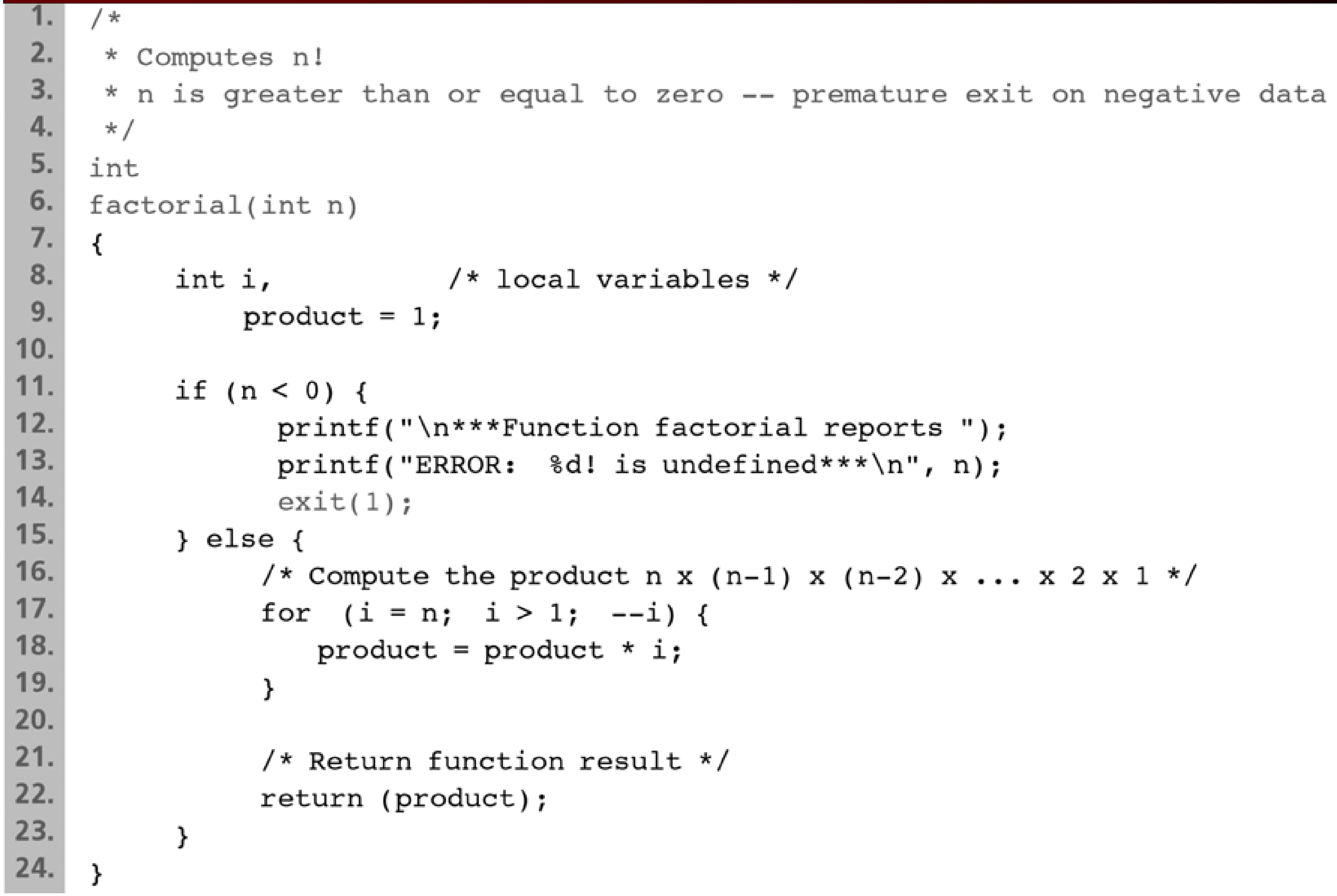


```
/* filetwo.c */

/* #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)
{ . . . }
```

Storage Classes

- auto
- extern
- static
  - Allocated and initialized once prior to program execution
    - Heap is used instead of stack
  - Remains until the program termination
  - Retains data from one call to another
    - Function does not behave solely based on the parameter values
  - Ex: static double matrix[50][40];
- register
  - Advise compiler to use register for the variable
    - Used for variable accessed more often than others
  - Ex: register int row, col;

Premature Exit on Negative Data



```
/*
 * Computes n!
 * n is greater than or equal to zero -- premature exit on negative data
 */
int
factorial(int n)
{
      int i,          /* local variables */
          product = 1;

      if (n < 0) {
            printf("\n***Function factorial reports ");
            printf("ERROR:  %d! is undefined***\n", n);
            exit(1);
      } else {
           /* Compute the product n x (n-1) x (n-2) x ... x 2 x 1 */
           for  (i = n;  i > 1;  --i) {
               product = product * i;
           }

           /* Return function result */
           return (product);
      }
}
```

Conditional Compilation

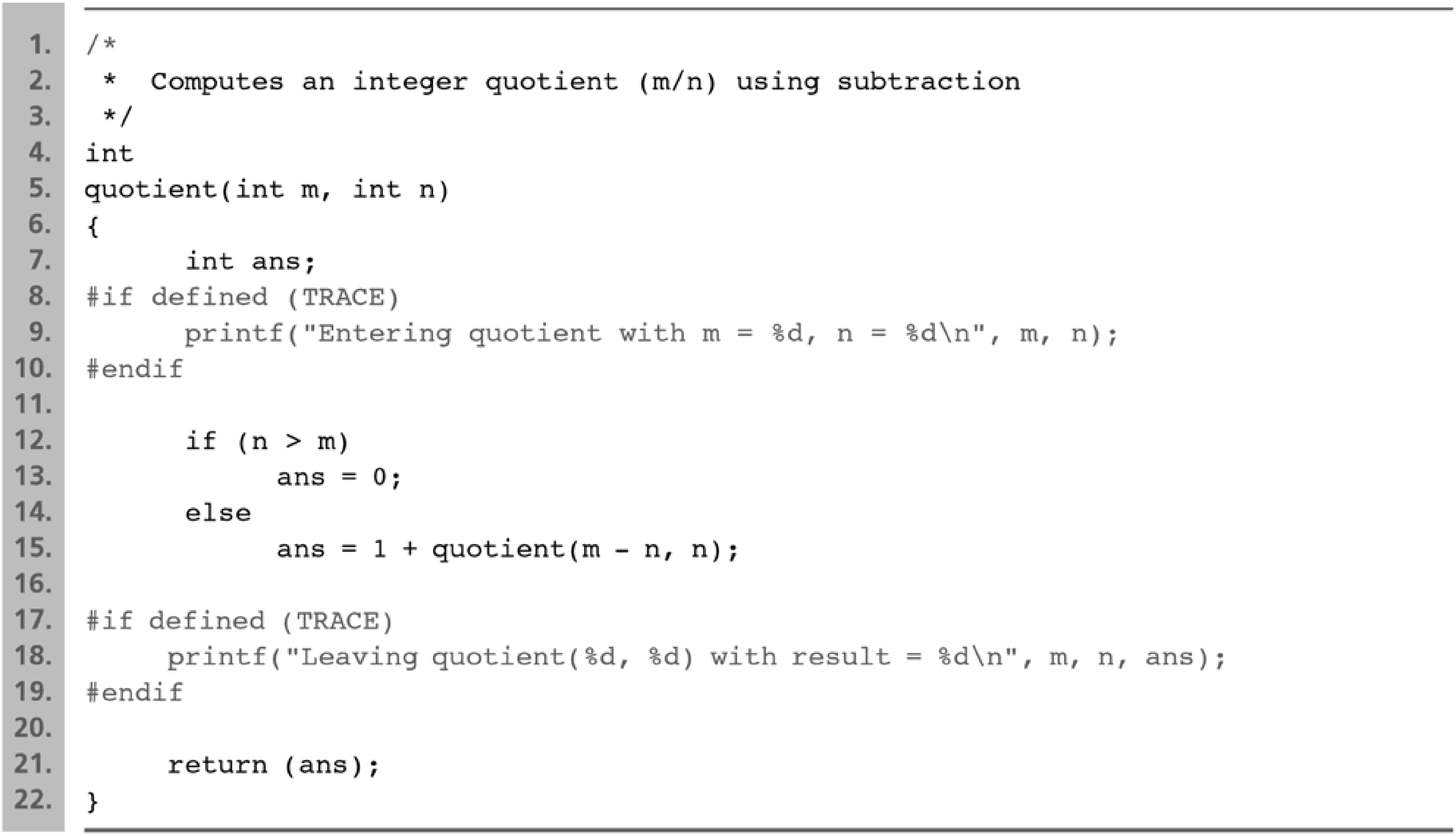
- Selecting parts of program to be compiled and omitted
  - Debugging (tracing) printf statements
  - including header files
  - Software design for variety of computers

```
#if defined (DEBUG)
    printf(….);
#endif
```

- Define constant macro DEBUG for debugging

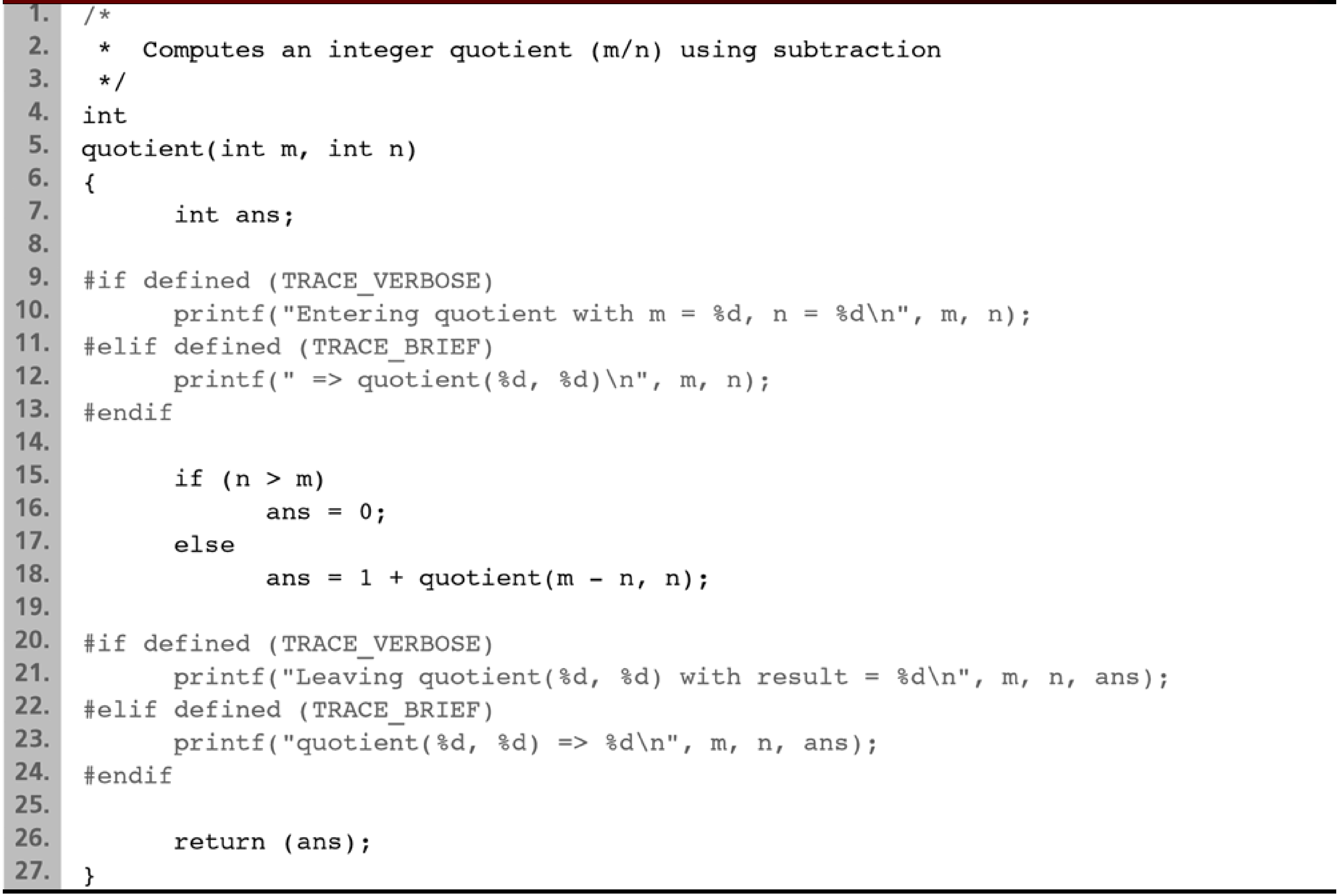
```
#elif
#else
#undef
```

Conditional Compilation



```
/*
 *  Computes an integer quotient (m/n) using subtraction
 */
int
quotient(int m, int n)
{
      int ans;
#if defined (TRACE)
      printf("Entering quotient with m = %d, n = %d\n", m, n);
#endif

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

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

     return (ans);
}
```

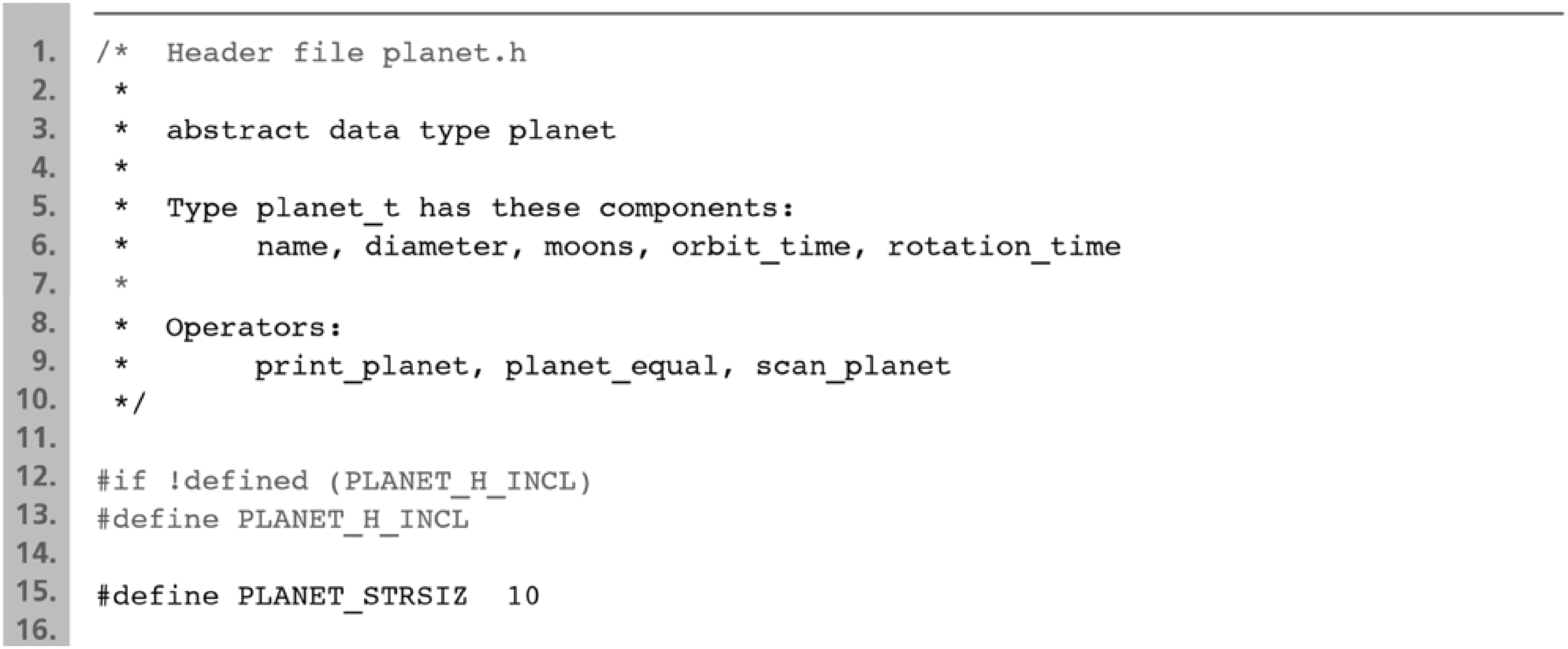
Conditional Compilation



```
/*
 *  Computes an integer quotient (m/n) using subtraction
 */
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);
}
```

Duplicate Inclusion

- Header file that protects itself from effects of duplicate inclusion



```
/*   Header file 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
 */

#if !defined (PLANET_H_INCL)
#define PLANET_H_INCL

#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                */

/*
 *  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                                */

#endif
```

Arguments to Function main

- Function main has two formal parameters
  - Integer: argument count
  - Array of pointers to strings: arguments

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

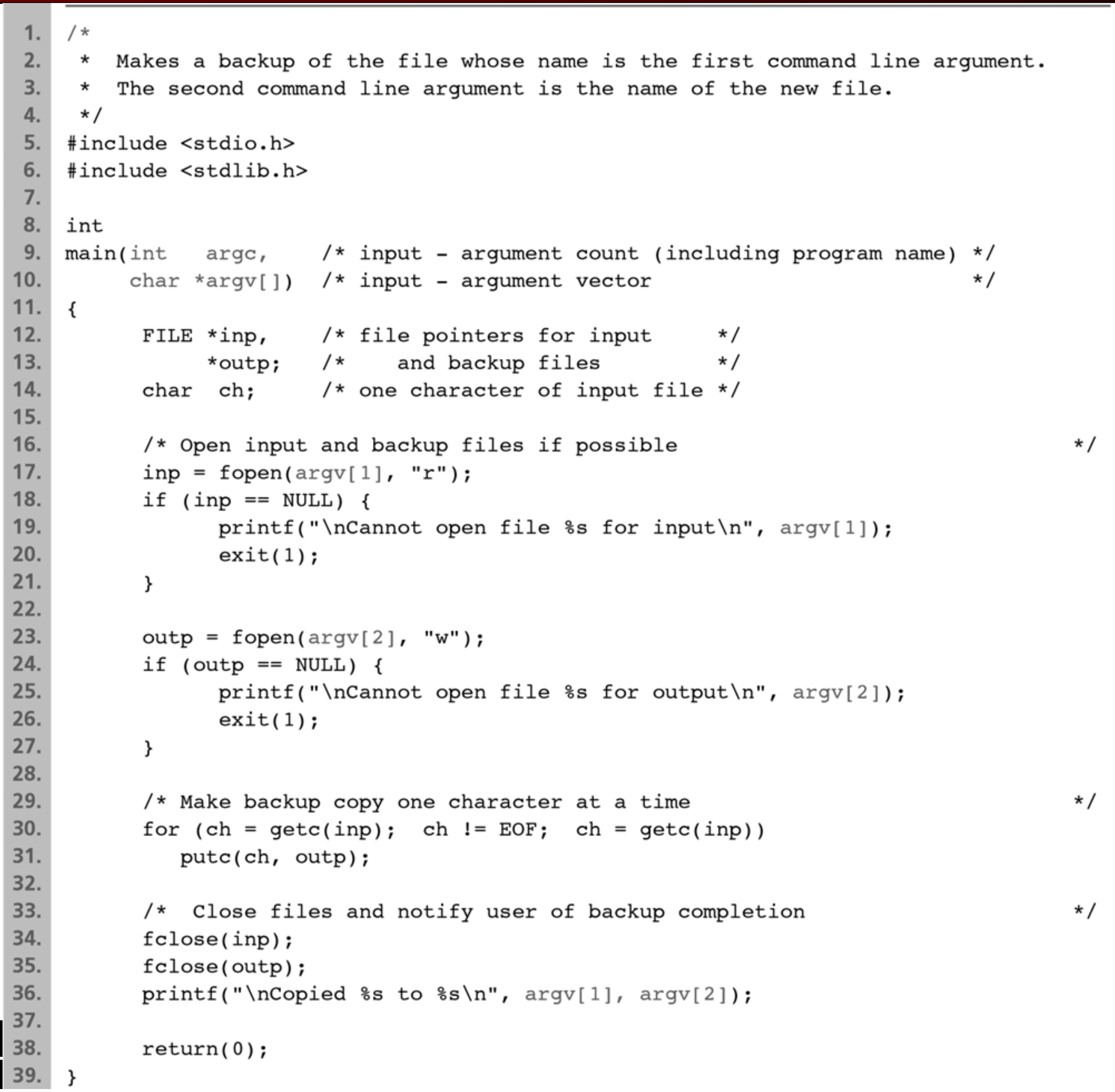
  - While you run your program

```
> prog opt1 opt2
```

  - argc has value of 3
  - argv[0] is “prog”, argv[1] is “opt1”, argv[2] is “opt2”

  - EX: backup program

Arguments to Function main



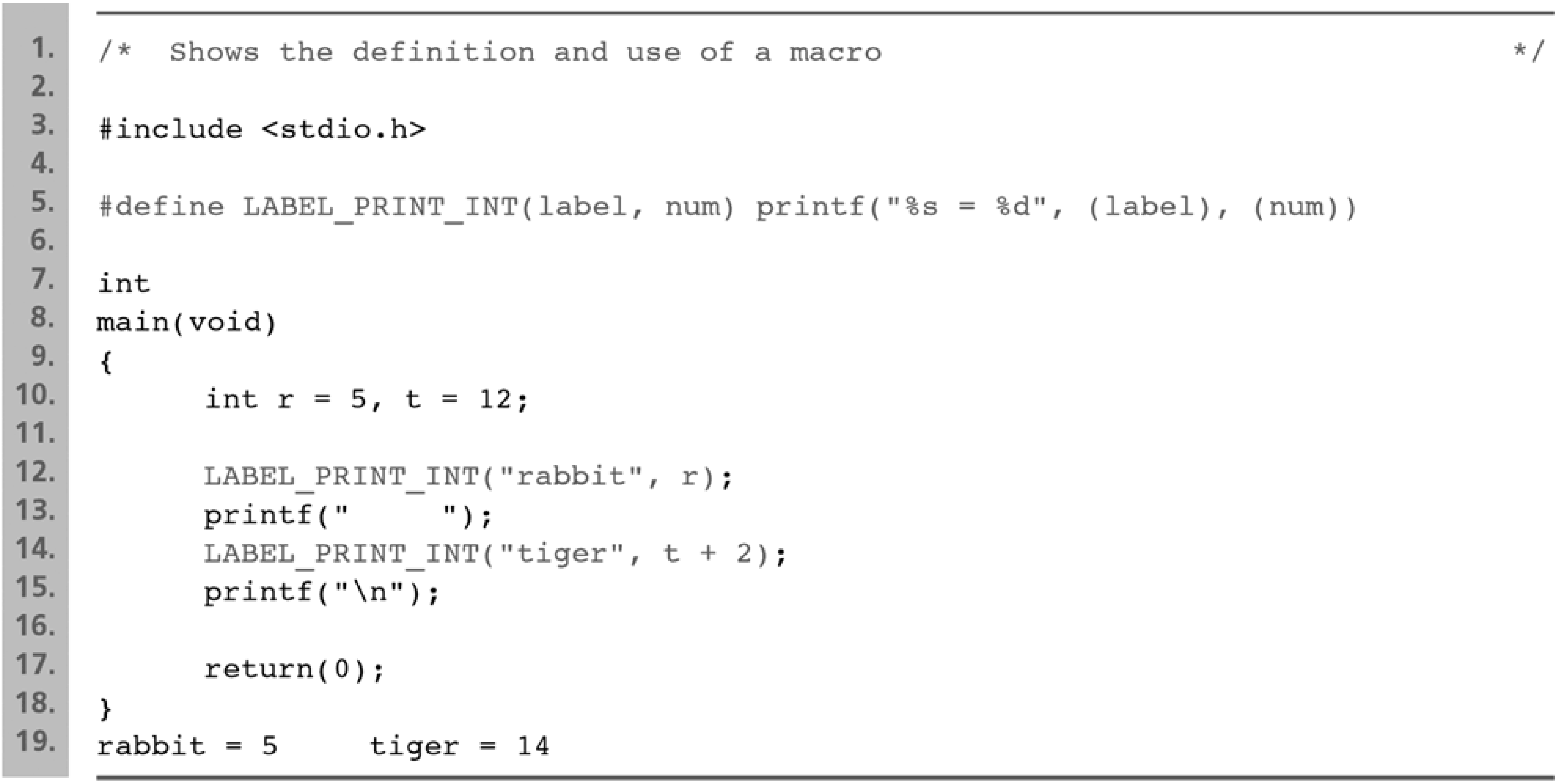
```c
/*
 *  Makes a backup of the file whose name is the first command line argument.
 *  The second command line argument is the name of the new file.
 */
#include <stdio.h>
#include <stdlib.h>

int
main(int   argc,    /* input - argument count (including program name) */
     char *argv[])  /* input - argument vector                          */
{
      FILE *inp,    /* file pointers for input     */
           *outp;   /*     and backup files        */
      char  ch;     /* one character of input file */

      /* Open input and backup files if possible                               */
      inp = fopen(argv[1], "r");
      if (inp == NULL) {
            printf("\nCannot open file %s for input\n", argv[1]);
            exit(1);
      }

      outp = fopen(argv[2], "w");
      if (outp == NULL) {
            printf("\nCannot open file %s for output\n", argv[2]);
            exit(1);
      }

      /* Make backup copy one character at a time                              */
      for (ch = getc(inp);  ch != EOF;  ch = getc(inp))
         putc(ch, outp);

      /*  Close files and notify user of backup completion                     */
      fclose(inp);
      fclose(outp);
      printf("\nCopied %s to %s\n", argv[1], argv[2]);

      return(0);
}
```

Macros

- Constant macros: defines symbolic names
- Macros can have formal parameters
  - Gives a name to frequently used operation
  - No overhead of function calls

```
#define macro_name(parameter_list) macro_body

#define SQUARE(x) ((x)*(x))

#define ROOT(a,b,c) ((-(b)+sqrt((b)*(b)-4*(a)*(c)))/(2*(a)))
```

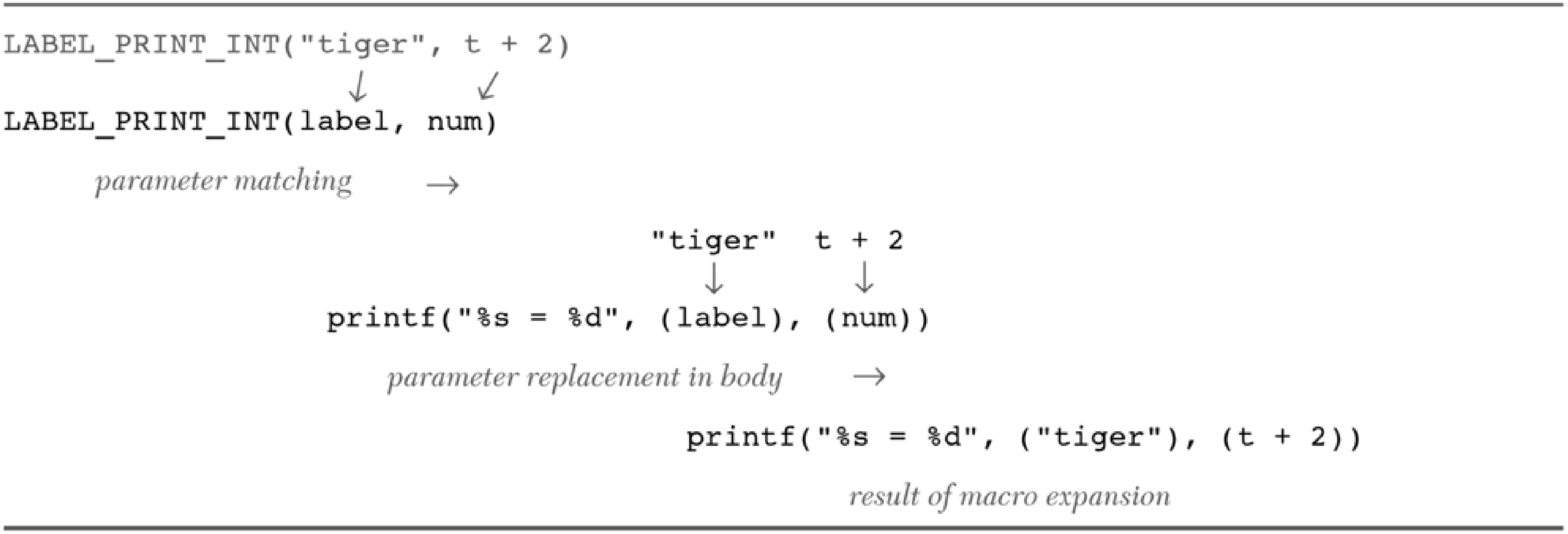
Macro with Formal Parameters



```c
/*  Shows the definition and use of a macro                                  */

#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);
}
```

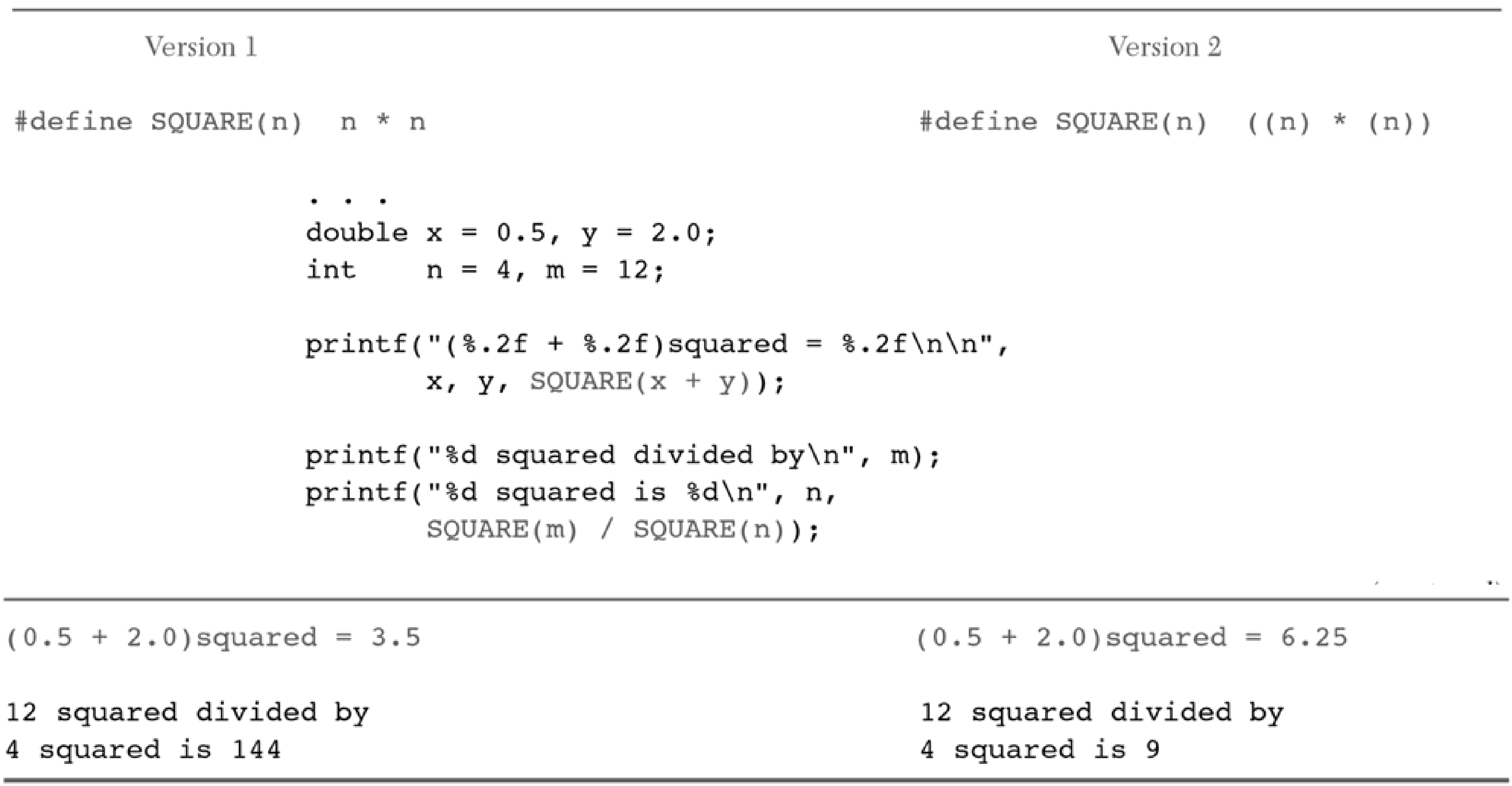
```
rabbit = 5     tiger = 14
```

Macro Expansion of Macro Call



```
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
```

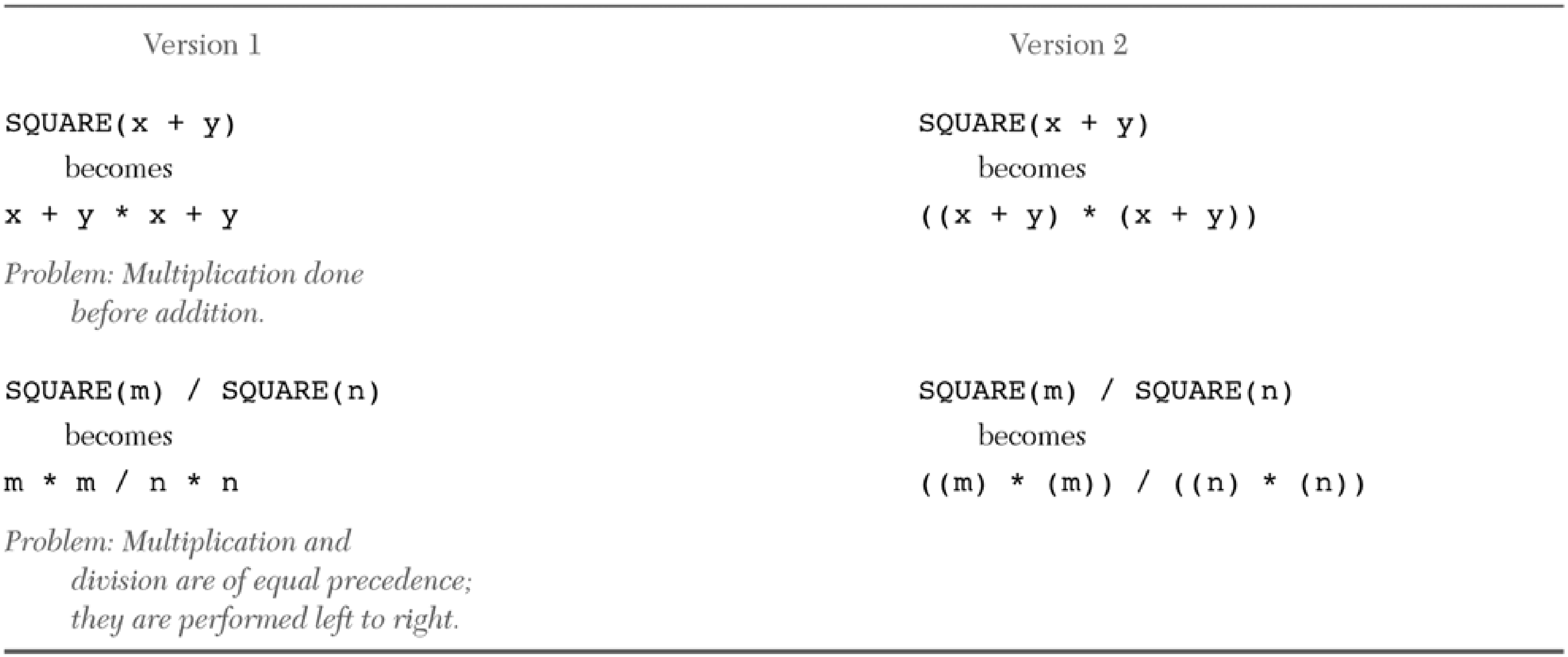
Importance of Parentheses in Macro Body



```
        Version 1                                                   Version 2

#define SQUARE(n)  n * n                                  #define SQUARE(n)  ((n) * (n))

                 . . .
                 double x = 0.5, y = 2.0;
                 int    n = 4, m = 12;

                 printf("(%.2f + %.2f)squared = %.2f\n\n",
                        x, y, SQUARE(x + y));

                 printf("%d squared divided by\n", m);
                 printf("%d squared is %d\n", n,
                        SQUARE(m) / SQUARE(n));
```

```
(0.5 + 2.0)squared = 3.5                                  (0.5 + 2.0)squared = 6.25

12 squared divided by                                     12 squared divided by
4 squared is 144                                          4 squared is 9
```

Macro Expansions of Macro Calls



Version 1	Version 2

`SQUARE(x + y)`
becomes
`x + y * x + y`
	`SQUARE(x + y)`
	becomes
	`((x + y) * (x + y))`

Problem: Multiplication done before addition.	

`SQUARE(m) / SQUARE(n)`
becomes
`m * m / n * n`
	`SQUARE(m) / SQUARE(n)`
	becomes
	`((m) * (m)) / ((n) * (n))`

Problem: Multiplication and division are of equal precedence; they are performed left to right.	

Macros

- Notes:
  - No space between macro name and (
  - Do not use semicolon at the end of the macro
  - Use parenthesis for each formal parameter
  - Avoid using operators with side effects in expressions as arguments in a macro call

```
#define ROOT(a,b,c) ((-(b)+sqrt((b)*(b)-4*(a)*(c)))/(2*(a)))
r = ROOT(++n1, n2, n3);
r = ((-(n2)+sqrt((n2)*(n2)-4*(++n1)*(n3)))/(2*(++n1)));
```

  - Macro with more than one lines is possible
    - Use \ at the end of the line

```
#define INDEXED_FOR(ct, st, end) \
    for ((ct)=(st); (ct) < (end); ++(ct))
```