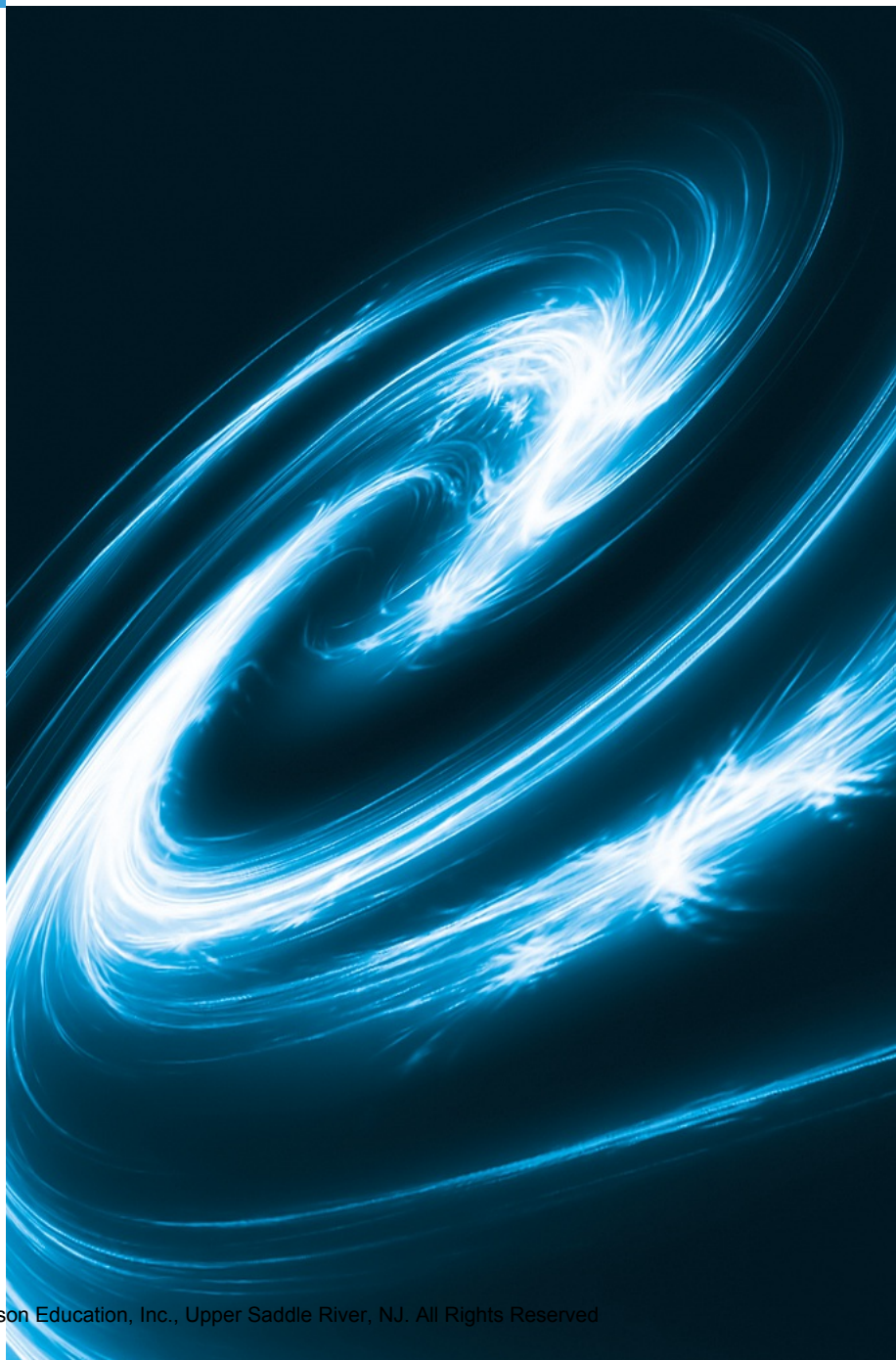
F

Additional Features of the C99 and C11 Standards



Objectives

In this appendix, you'll learn:

- Many additional key features of the C99 and C11 standards.
- To mix declarations and executable code and to declare variables in `for` statement headers.
- To initialize arrays and `struct`s with designated initializers.
- To use data type `bool` to create boolean variables whose data values can be `true` or `false`.
- To manipulate variable-length arrays.
- To perform arithmetic operations on complex variables.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Outline

F.1 Introduction

C99 (1999) and C11 (2011) are revised standards for the C programming language that refine and expand the capabilities of Standard C. C99 introduced a larger collection of changes than C11. As you read, keep in mind that not every compiler implements every C99 feature. Also, C11 is still sufficiently new that even the compilers that intend to implement the new features may not have done so yet. Before using the capabilities shown here, check that your compiler supports them. Our goal is simply to introduce these capabilities and to provide resources for further reading.

We discuss compiler support and include links to several free compilers and IDEs that provide various levels of C99 and C11 support. We explain with complete working code examples and code snippets some of these key features that were not discussed in the main text, including mixing declarations and executable code, declarations in `for` statements, designated initializers, compound literals, type `bool`, implicit `int` return type in function prototypes and function definitions (not allowed in C11), complex numbers and variable-length arrays. We provide brief explanations for additional key C99 features, including extended identifiers, restricted pointers, reliable integer division, flexible array members, generic math, `inline` functions and `return` without expression. Another significant C99 feature is the addition of `float` and `long double` versions of most of the math functions in `<math.h>`. We discuss capabilities of the recent C11 standard, including multithreading, improved unicode support, the `_Noreturn` function specifier, type-generic expressions, the `quick_exit` function, anonymous structures and unions, memory alignment control, static assertions, analyzability and floating-point types. Many of these capabilities have been designated as optional. We include an extensive list of Internet and web resources to help you locate appropriate C11 compilers and IDEs, and dig deeper into the technical details of the language.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

F.2 Support for C99

Most C and C++ compilers did not support C99 when it was released. Support has grown in recent years, and many compilers are close to being C99 compliant. The Microsoft Visual C++ 2012 Express Edition Software included with this book does not support C99. For more information about this, visit `blogs.msdn.com/vcblog/archive/2007/11/05/iso-c-standard-update.aspx`.

In this appendix, we run GNU GCC 4.3 on Linux, which supports most C99 features. To specify that the C99 standard should be used in compilation, you need to include the command-line argument "`-std=c99`" when you compile your programs. On Windows, you can install GCC to run C99 programs by downloading either Cygwin (`www.cygwin.com`) or MinGW (`sourceforge.net/projects/mingw`). Cygwin is a complete Linux-style environment for Windows, while MinGW (Minimalist GNU for Windows) is a native Windows port of the compiler and related tools. Additionally, both GCC and Microsoft Visual C++ can use the Dinkum C99 Standard Library from Dinkumware (`www.dinkumware.com/c99.aspx`). This library provides all of the new required headers in C99 and C95.

F.3 New C99 Headers

Figure F.1 lists alphabetically the standard library headers added in C99 (three of these were added in C95). All of these remain available in C11. We'll discuss the new C11 headers later in this appendix.

Standard library header	Explanation
`<complex.h>`	Contains macros and function prototypes for supporting *complex numbers* (see Section F.9). [C99 feature.]
`<fenv.h>`	Provides information about the C implementation's *floating-point environment and capabilities*. [C99 feature.]
`<inttypes.h>`	Defines several new *portable integral types* and provides *format specifiers for defined types*. [C99 feature.]
`<iso646.h>`	Defines *macros* that represent the equality, relational and bitwise operators; an *alternative to trigraphs*. [C95 feature.]
`<stdbool.h>`	Contains macros defining `bool`, `true` and `false`, used for *boolean variables* (see Section F.7). [C99 feature.]
`<stdint.h>`	Defines *extended integer types and related macros*. [C99 feature.]
`<tgmath.h>`	Provides *type-generic macros* that allow functions from `<math.h>` to be used with a variety of parameter types (see Section F.12). [C99 feature.]
`<wchar.h>`	Along with `<wctype.h>`, provides *multibyte and wide-character input and output support*. [C95 feature.]
`<wctype.h>`	Along with `<wchar.h>`, provides *wide-character library support*. [C95 feature.]

Fig. F.1 | Standard library headers added in C99 and C95.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

F.4 Mixing Declarations and Executable Code

[**This section can be taught after Section 2.3.**]
Prior to C99 *all* variables with block scope had to be declared at the *start* of a block. C99 allows **mixing declarations and executable code**. A variable can be declared anywhere in a block prior to its usage. Consider the C99 program of Fig. F.2.

```
1  // Fig. F.2: figF_02.c
2  // Mixing declarations and executable code in C99
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int x = 1; // declare variable at beginning of block
8     printf( "x is %d\n", x );
9
10    int y = 2; // declare variable in middle of executable code
11    printf( "y is %d\n", y );
12 } // end main
```

```
x is 1
y is 2
```

Fig. F.2 | Mixing declarations and executable code in C99.

In this program, we call `printf` (executable code) in line 8, yet declare the `int` variable `y` in line 10. In C99 you can declare variables close to their first use, even if those declarations appear *after* executable code in a block. We don't declare `int y` (line 10) until just before we use it (line 11). Although this can improve program readability and reduce the possibility of unintended references, some programmers still prefer to group their variable declarations together at the beginnings of blocks. A variable cannot be declared *after* code that uses the variable.

F.5 Declaring a Variable in a `for` Statement Header

[**This section can be taught after Section 4.4.**]
As you may recall, a `for` statement consists of an initialization, a loop-continuation condition, an increment and a loop body.

C99 allows the initialization clause of the `for` statement to include a declaration. Rather than using an existing variable as a loop counter, we can create a new loop-counter variable in the `for` statement header whose scope is limited to the `for` statement. The program of Fig. F.3 declares a variable in a `for` statement header.

```
1  // Fig. F.3: figF_03.c
2  // Declaring a variable in a for statement header
3  #include <stdio.h>
4
```

Fig. F.3 | Declaring a variable in a `for` statement header in C99. (Part 1 of 2.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```
5  int main( void )
6  {
7     printf( "Values of x\n" );
8
9     // declare a variable in a for statement header
10    for ( int x = 1; x <= 5; ++x ) {
11       printf( "%d\n", x );
12    } // end for
13 } // end main
```

```
Values of x
1
2
3
4
5
```

Fig. F.3 | Declaring a variable in a `for` statement header in C99. (Part 2 of 2.)

Any variable declared in a `for` statement has the scope of the `for` statement—the variable does *not* exist *outside* the `for` statement and attempting to access such a variable after the statement body is a compilation error.

F.6 Designated Initializers and Compound Literals

[**This section can be taught after Section 10.3.**]
Designated initializers allow you to initialize the elements of an array, `union` or `struct` explicitly by subscript or name. Figure F.4 shows how we might assign the first and last elements of an array.

```
1  /* Fig. F.4: figF_04.c
2     Assigning elements of an array in prior to C99 */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int i; /* declare loop counter */
8     int a[ 5 ]; /* array declaration */
9
10    a[ 0 ] = 1; /* explicitly assign values to array elements... */
11    a[ 4 ] = 2; /* after the declaration of the array */
12
13    /* assign zero to all elements but the first and last */
14    for ( i = 1; i < 4; ++i ) {
15       a[ i ] = 0;
16    } /* end for */
17
18    /* output array contents */
19    printf( "The array is\n" );
20
```

Fig. F.4 | Assigning elements of an array prior to C99. (Part 1 of 2.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```
21      for ( i = 0; i < 5; ++i ) {
22         printf( "%d\n", a[ i ] );
23      } /* end for */
24   } /* end main */
```

```
The array is
1
0
0
0
2
```

Fig. F.4 | Assigning elements of an array prior to C99. (Part 2 of 2.)

In Fig. F.5 we show the program again, but rather than *assigning* values to the first and last elements of the array, we *initialize* them explicitly by subscript, using designated initializers.

```
1   // Fig. F.5: figF_05.c
2   // Using designated initializers
3   // to initialize the elements of an array in C99
4   #include <stdio.h>
5
6   int main( void )
7   {
8      int a[ 5 ] =
9      {
10        [ 0 ] = 1, // initialize elements with designated initializers...
11        [ 4 ] = 2 // within the declaration of the array
12     }; // semicolon is required
13
14     // output array contents
15     printf( "The array is \n" );
16
17     for ( int i = 0; i < 5; ++i ) {
18        printf( "%d\n", a[ i ] );
19     } // end for
20  } // end main
```

```
The array is
1
0
0
0
2
```

Fig. F.5 | Using designated initializers to initialize the elements of an array in C99.

Lines 8–12 declare the array and initialize the specified elements within the braces. Note the syntax. Each initializer in the initializer list (lines 10–11) is separated from the next by a comma, and the end brace is followed by a semicolon. Elements that are not

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

explicitly initialized are *implicitly* initialized to zero (of the correct type). This syntax was not allowed prior to C99.

In addition to using an initializer list to declare a variable, you can also use an initializer list to create an unnamed array, `struct` or `union`. This is known as a **compound literal**. For example, if you want to pass an array equivalent to `a` in Fig. F.5 to a function without having to declare it beforehand, you could use

```
demoFunction( ( int [ 5 ] ) { [ 0 ] = 1, [ 4 ] = 2 } );
```

Consider the more elaborate example in Fig. F.6, where we use designated initializers for an array of `structs`.

```
1  // Fig. F.6: figF_06.c
2  // Using designated initializers to initialize an array of structs in C99
3  #include <stdio.h>
4
5  struct twoInt // declare a struct of two integers
6  {
7     int x;
8     int y;
9  }; // end struct twoInt
10
11 int main( void )
12 {
13    // explicitly initialize elements of array a
14    // then explicitly initialize members of each struct element
15    struct twoInt a[ 5 ] =
16    {
17       [ 0 ] = { .x = 1, .y = 2 },
18       [ 4 ] = { .x = 10, .y = 20 }
19    };
20
21    // output array contents
22    printf( "x\ty\n" );
23
24    for ( int i = 0; i < 5; ++i ) {
25       printf( "%d\t%d\n", a[ i ].x, a[ i ].y );
26    } // end for
27 } //end main
```

```
x       y
1       2
0       0
0       0
0       0
10      20
```

Fig. F.6 | Using designated initializers to initialize an array of `structs` in C99.

Line 17 uses a *designated initializer* to explicitly initialize a `struct` element of the array. Then, within that initialization, we use another level of designated initializer, explic-

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

itly initializing the x and y members of the `struct`. To initialize `struct` or `union` members we list each member's name preceded by a *period*.

Compare lines 15–19 of Fig. F.6, which use designated initializers to the following executable code, which does not use designated initializers:

```
struct twoInt a[ 5 ];

a[ 0 ].x = 1;
a[ 0 ].y = 2;
a[ 4 ].x = 10;
a[ 4 ].y = 20;
```

F.7 Type bool

[This section can be taught after Section 3.6.]

The C99 boolean type is `_Bool`, which can hold only the values 0 or 1. Recall C's convention of using *zero* and *nonzero* values to represent *false* and *true*—the value 0 in a condition evaluates to *false*, while *any* nonzero value in a condition evaluates to *true*. Assigning *any* non-zero value to a `_Bool` sets it to 1. C99 provides the `<stdbool.h>` header file which defines macros representing the type `bool` and its values (`true` and `false`). These macros replace `true` with 1, `false` with 0 and `bool` with the C99 keyword `_Bool`. Figure F.7 uses a function named `isEven` (lines 29–37) that returns a `bool` value of `true` if the number is even and `false` if it is odd.

```
1  // Fig. F.7: figF_07.c
2  // Using the boolean type and the values true and false in C99.
3  #include <stdio.h>
4  #include <stdbool.h> // allows the use of bool, true, and false
5
6  bool isEven( int number ); // function prototype
7
8  int main( void )
9  {
10    // loop for 2 inputs
11    for ( int i = 0; i < 2; ++i ) {
12       int input; // value entered by user
13       printf( "Enter an integer: " );
14       scanf( "%d", &input );
15
16       bool valueIsEven = isEven( input ); // determine if input is even
17
18       // determine whether input is even
19       if ( valueIsEven ) {
20          printf( "%d is even \n\n", input );
21       } // end if
22       else {
23          printf( "%d is odd \n\n", input );
24       } // end else
25    } // end for
26 } // end main
```

Fig. F.7 | Using the type `bool` and the values `true` and `false` in C99. (Part 1 of 2.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```
27
28  // even returns true if number is even
29  bool isEven( int number )
30  {
31     if ( number % 2 == 0 ) { // is number divisible by 2?
32        return true;
33     }
34     else {
35        return false;
36     }
37  } // end function isEven
```

```
Enter an integer: 34
34 is even

Enter an integer: 23
23 is odd
```

Fig. F.7 | Using the type `bool` and the values `true` and `false` in C99. (Part 2 of 2.)

Line 16 declares a `bool` variable named `valueIsEven`. Lines 13–14 in the loop prompt for and obtain the next integer. Line 16 passes the `input` to function `isEven` (lines 29–37). Function `isEven` returns a value of type `bool`. Line 31 determines whether the argument is divisible by 2. If so, line 32 returns `true` (i.e., the number is *even*); otherwise, line 32 returns `false` (i.e., the number is odd). The result is assigned to `bool` variable `valueIsEven` in line 16. If `valueIsEven` is `true`, line 20 displays a string indicating that the value is *even*. If `valueIsEven` is `false`, line 23 displays a string indicating that the value is *odd*.

F.8 Implicit `int` in Function Declarations

[**This section can be taught after Section 5.5.**]
Prior to C99, if a function does not have an *explicit* return type, it *implicitly* returns an `int`. In addition, if a function does not specify a parameter type, that type implicitly becomes `int`. Consider the program in Fig. F.8.

```
1  /* Fig. F.8: figF_08.c
2     Using implicit int prior to C99 */
3  #include <stdio.h>
4
5  returnImplicitInt(); /* prototype with unspecified return type */
6  int demoImplicitInt( x ); /* prototype with unspecified parameter type */
7
8  int main( void )
9  {
10    int x;
11    int y;
12
```

Fig. F.8 | Using implicit `int` prior to C99. (Part 1 of 2.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```
13      /* assign data of unspecified return type to int */
14      x = returnImplicitInt();
15
16      /* pass an int to a function with an unspecified type */
17      y = demoImplicitInt( 82 );
18
19      printf( "x is %d\n", x );
20      printf( "y is %d\n", y );
21   } /* end main */
22
23   returnImplicitInt()
24   {
25      return 77; /* returning an int when return type is not specified */
26   } /* end function returnImplicitInt */
27
28   int demoImplicitInt( x )
29   {
30      return x;
31   } /* end function demoImplicitInt */
```

Fig. F.8 | Using implicit `int` prior to C99. (Part 2 of 2.)

When this program is run in Microsoft's Visual C++ 2008 Express Edition, which is not C99 compliant, no compilation errors or warning messages occur and the program executes correctly. C99 *disallows* the use of the implicit `int`, requiring that C99-compliant compilers issue either a warning or an error. When we run the same program using GCC 4.7, we get the warning messages shown in Fig. F.9.

```
figF_11.c:5:1: warning: data definition has no type or storage class
   [enabled by default]
figF_11.c:5:1: warning: type defaults to 'int' in declaration of
   'returnImplicitInt' [enabled by default]
figF_11.c:6:1: warning: parameter names (without types) in function
   declaration [enabled by default]
figF_11.c:23:1: warning: return type defaults to 'int' [enabled by default]
figF_11.c: In function 'demoImplicitInt':
figF_11.c:28:5: warning: type of 'x' defaults to 'int' [enabled by default]
```

Fig. F.9 | Warning messages for implicit `int` produced by GCC 4.3.

F.9 Complex Numbers

[**This section can be taught after Section 5.3.**]
The C99 standard introduces support for complex numbers and complex arithmetic. The program of Fig. F.10 performs basic operations with complex numbers.

```
1  // Fig. F.10: figF_10.c
2  // Using complex numbers in C99
3  #include <stdio.h>
```

Fig. F.10 | Using complex numbers in C99. (Part 1 of 2.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```
4  #include <complex.h> // for complex type and math functions
5
6  int main( void )
7  {
8      double complex a = 32.123 + 24.456 * I; // a is 32.123 + 24.456i
9      double complex b = 23.789 + 42.987 * I; // b is 23.789 + 42.987i
10     double complex c = 3.0 + 2.0 * I;
11
12     double complex sum = a + b; // perform complex addition
13     double complex pwr = cpow( a, c ); // perform complex exponentiation
14
15     printf( "a is %f + %fi\n", creal( a ), cimag( a ));
16     printf( "b is %f + %fi\n", creal( b ), cimag( b ));
17     printf( "a + b is: %f + %fi\n", creal( sum ), cimag( sum ));
18     printf( "a - b is: %f + %fi\n", creal( a - b ), cimag( a - b ));
19     printf( "a * b is: %f + %fi\n", creal( a * b ), cimag( a * b ));
20     printf( "a / b is: %f + %fi\n", creal( a / b ), cimag( a / b ));
21     printf( "a ^ b is: %f + %fi\n", creal( pwr ), cimag( pwr ));
22 } // end main
```

```
a is 32.123000 + 24.456000i
b is 23.789000 + 42.987000i
a + b is: 55.912000 + 67.443000i
a - b is: 8.334000 + -18.531000i
a * b is: -287.116025 + 1962.655185i
a / b is: 0.752119 + -0.331050i
a ^ b is: -17857.051995 + 1365.613958i
```

Fig. F.10 | Using complex numbers in C99. (Part 2 of 2.)

For C99 to recognize `complex`, we include the `<complex.h>` header (line 4). This will expand the macro `complex` to the keyword `_Complex`—a type that reserves an array of exactly two elements, corresponding to the complex number's *real part* and *imaginary part*.

Having included the header file in line 4, we can define variables as in lines 8–10 and 12–13. We define each of the variables `a`, `b`, `c`, `sum` and `pwr` as type `double complex`. We also could have used `float complex` or `long double complex`.

The arithmetic operators also work with complex numbers. The `<complex.h>` header also defines several math functions, for example, `cpow` in line 13. You can also use the operators `!`, `++`, `--`, `&&`, `||`, `==`, `!=` and unary `&` with complex numbers.

Lines 17–21 output the results of various arithmetic operations. The *real part* and the *imaginary part* of a complex number can be accessed with functions `creal` and `cimag`, respectively, as shown in lines 15–21. In the output string of line 21, we use the symbol `^` to indicate exponentiation.

F.10 Variable-Length Arrays

[This section can be taught after Section 6.9.]

Prior to C99, arrays were of *constant size*. But what if you don't know an array's size at compilation time? To handle this, you'd have to use *dynamic memory allocation* with `malloc` and related functions. C99 allows you to handle arrays of unknown size using *variable-*

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

length arrays (VLAs). A **variable-length array** is an array whose length, or size, is defined in terms of an expression evaluated at *execution time*. The program of Fig. F.11 declares and prints several VLAs.

```
1  // Fig. F.11: figF_11.c
2  // Using variable-length arrays in C99
3  #include <stdio.h>
4
5  // function prototypes
6  void print1DArray( int size, int arr[ size ] );
7  void print2DArray( int row, int col, int arr[ row ][ col ] );
8
9  int main( void )
10 {
11    int arraySize; // size of 1-D array
12    int row1, col1, row2, col2; // number of rows and columns in 2-D arrays
13
14    printf( "Enter size of a one-dimensional array: " );
15    scanf( "%d", &arraySize );
16
17    printf( "Enter number of rows and columns in a 2-D array: " );
18    scanf( "%d %d", &row1, &col1 );
19
20    printf( "Enter number of rows and columns in another 2-D array: " );
21    scanf( "%d %d", &row2, &col2 );
22
23    int array[ arraySize ]; // declare 1-D variable-length array
24    int array2D1[ row1 ][ col1 ]; // declare 2-D variable-length array
25    int array2D2[ row2 ][ col2 ]; // declare 2-D variable-length array
26
27    // test sizeof operator on VLA
28    printf( "\nsizeof(array) yields array size of %d bytes\n",
29       sizeof( array ) );
30
31    // assign elements of 1-D VLA
32    for ( int i = 0; i < arraySize; ++i ) {
33       array[ i ] = i * i;
34    } // end for
35
36    // assign elements of first 2-D VLA
37    for ( int i = 0; i < row1; ++i ) {
38       for ( int j = 0; j < col1; ++j ) {
39          array2D1[ i ][ j ] = i + j;
40       } // end for
41    } // end for
42
43    // assign elements of second 2-D VLA
44    for ( int i = 0; i < row2; ++i ) {
45       for ( int j = 0; j < col2; ++j ) {
46          array2D2[ i ][ j ] = i + j;
47       } // end for
48    } // end for
```

Fig. F.11 | Using variable-length arrays in C99. (Part 1 of 2.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```
49
50     printf( "\nOne-dimensional array:\n" );
51     print1DArray( arraySize, array ); // pass 1-D VLA to function
52
53     printf( "\nFirst two-dimensional array:\n" );
54     print2DArray( row1, col1, array2D1 ); // pass 2-D VLA to function
55
56     printf( "\nSecond two-dimensional array:\n" );
57     print2DArray( row2, col2, array2D2 ); // pass other 2-D VLA to function
58  } // end main
59
60  void print1DArray( int size, int array[ size ] )
61  {
62     // output contents of array
63     for ( int i = 0; i < size; ++i ) {
64        printf( "array[%d] = %d\n", i, array[ i ] );
65     } // end for
66  } // end function print1DArray
67
68  void print2DArray( int row, int col, int arr[ row ][ col ] )
69  {
70     // output contents of array
71     for ( int i = 0; i < row; ++i ) {
72        for ( int j = 0; j < col; ++j ) {
73           printf( "%5d", arr[ i ][ j ] );
74        } // end for
75
76        printf( "\n" );
77     } // end for
78  } // end function print2DArray
```

```
Enter size of a one-dimensional array: 6
Enter number of rows and columns in a 2-D array: 2 5
Enter number of rows and columns in another 2-D array: 4 3

sizeof(array) yields array size of 24 bytes

One-dimensional array:
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25

First two-dimensional array:
    0    1    2    3    4
    1    2    3    4    5

Second two-dimensional array:
    0    1    2
    1    2    3
    2    3    4
    3    4    5
```

Fig. F.11 | Using variable-length arrays in C99. (Part 2 of 2.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

First, we prompt the user for the desired sizes for a one-dimensional array and two two-dimensional arrays (lines 14–21). Lines 23–25 then declare VLAs of the appropriate size. This used to lead to a compilation error but is valid in C99, as long as the variables representing the array sizes are of an integral type.

After declaring the arrays, we use the `sizeof` operator (line 29) to make sure that our VLA is of the proper length. Prior to C99 `sizeof` was a *compile-time-only* operation, but when applied to a VLA in C99, `sizeof` operates at *runtime*. The output window shows that the `sizeof` operator returns a size of 24 bytes—four times that of the number we entered because the size of an `int` on our machine is 4 bytes.

Next we assign values to the VLA elements (lines 32–48). We use `i < arraySize` as our loop-continuation condition when filling the one-dimensional array. As with fixed-length arrays, there's *no protection against stepping outside the array bounds*.

Lines 60–66 define function `print1DArray` that takes a one-dimensional VLA. The syntax for passing VLAs as parameters to functions is the same as with normal, fixed-length arrays. We use the variable `size` in the declaration of the `array` parameter (`int array[size]`), but no checking is performed other than the variable being defined and of integral type—it's purely *documentation* for the programmer.

Function `print2DArray` (lines 68–78) takes a variable-length two-dimensional array and displays it to the screen. Recall from Section 6.9 that prior to C99, all but the first subscript of a multi-dimensional array must be specified when declaring a multidimensional-array function parameter. The same restriction holds true for C99, except that for VLAs the sizes can be specified by variables. The initial value of `col` passed to the function is used to convert from two-dimensional indices to offsets into the contiguous memory the array is stored in, just as with a fixed-size array. Changing the value of `col` inside the function will *not* cause any changes to the indexing, but passing an incorrect value to the function will.

F.11 Additions to the Preprocessor

[This section can be taught after Chapter 13.]

C99 adds features to the C preprocessor. The first is the `_Pragma` operator, which functions like the `#pragma` directive introduced in Section 13.6. `_Pragma ( "tokens" )` has the same effect as `#pragma tokens`, but is more flexible because it can be used inside a macro definition. Therefore, instead of surrounding each usage of a compiler-specific pragma by an `#if` directive, you can simply define a macro using the `_Pragma` operator once and use it anywhere in your program.

Second, C99 specifies three standard pragmas that deal with the behavior of floating-point operations. The first token in these standard pragmas is always `STDC`, the second is one of `FENV_ACCESS`, `FP_CONTRACT` or `CX_LIMITED_RANGE`, and the third is `ON`, `OFF` or `DEFAULT` to indicate whether the given pragma should be *enabled*, *disabled*, or set to its *default value*, respectively. The `FENV_ACCESS` pragma is used to inform the compiler which portions of code will use functions in the C99 `<fenv.h>` header. On modern desktop systems, floating-point processing is done with 80-bit floating-point values. If `FP_CONTRACT` is enabled, the compiler may perform a sequence of operations at this precision and store the final result into a lower-precision `float` or `double` instead of reducing the precision after each operation. Finally, if `CX_LIMITED_RANGE` is enabled, the compiler is allowed to use the standard mathematical formulas for complex operations such as multiplying or

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

dividing. Because floating-point numbers are *not* stored exactly, using the normal mathematical definitions can result in *overflows* where the numbers get larger than the floating-point type can represent, even if the operands and result are below this limit.

Third, the C99 preprocessor allows passing *empty arguments* to a macro call—in the previous version, the behavior of an empty argument was *undefined*, though GCC acts according to the C99 standard even in C89 mode. In many cases, it results in a syntax error, but in some cases it can be useful. For instance, consider a macro `PTR(type, cv, name)` defined to be `type * cv name`. In some cases, there is no `const` or `volatile` declaration on the pointer, so the second argument will be empty. When an empty macro argument is used with the `#` or `##` operator (Section 13.7), the result is the empty string or the identifier the argument was concatenated with, respectively.

A key preprocessor addition is *variable-length argument lists for macros*. This allows for macro wrappers around functions like `printf`—for example, to automatically add the name of the current file to a debug statement, you can define a macro as follows:

```
#define DEBUG( ... ) printf( __FILE__ ": " __VA_ARGS__ )
```

The `DEBUG` macro takes a variable number of arguments, as indicated by the `...` in the argument list. As with functions, the `...` must be the *last* argument; unlike functions, it may be the *only* argument. The identifier `__VA_ARGS__`, which begins and ends with *two* underscores, is a *placeholder* for the variable-length argument list. When a call such as

```
DEBUG( "x = %d, y = %d\n", x, y );
```

is preprocessed, it's replaced with

```
printf( "file.c" ": " "x = %d, y = %d\n", x, y );
```

As mentioned in Section 13.7, strings separated by white space are *concatenated* during preprocessing, so the three string literals will be combined to form the first argument to `printf`.

F.12 Other C99 Features

Here we provide brief overviews of some additional features of C99.

F.12.1 Compiler Minimum Resource Limits

[This section can be taught after Section 14.5.]
Prior to C99 the standard required implementations of the language to support identifiers of no less than 31 characters for identifiers with *internal linkage* (valid only within the file being compiled) and no less than six characters for identifiers with *external linkage* (also valid in other files). For more information on internal and external linkage, see Section 14.5. The C99 standard increases these limits to 63 characters for identifiers with internal linkage and to 31 characters for identifiers with external linkage. These are just *lower* limits. Compilers are free to support identifiers with *more* characters than these limits. Identifiers are now allowed to contain national language characters via Universal Character Names (C99 Standard, Section 6.4.3) and, if the implementation chooses, directly (C99 Standard, Section 6.4.2.1). [For more information, see C99 Standard Section 5.2.4.1.]

In addition to increasing the identifier length that compilers are required to support, the C99 standard sets *minimum* limits on many language features. For example, compilers are required to support at least 1023 members in a `struct`, `enum` or `union`, and at least 127

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

parameters to a function. For more information on other limits set by the C99 Standard, see C99 Standard Section 5.2.4.1.

F.12.2 The `restrict` Keyword

[This section can be taught after Section 7.5.]

The keyword `restrict` is used to declare *restricted pointers*. We declare a **restricted pointer** when that pointer should have *exclusive* access to a region of memory. Objects accessed through a restricted pointer cannot be accessed by other pointers except when the value of those pointers was derived from the value of the restricted pointer. We can declare a restricted pointer to an `int` as:

```
int *restrict ptr;
```

Restricted pointers allow the compiler to optimize the way the program accesses memory. For example, the standard library function `memcpy` is defined in the C99 standard as follows:

```
void *memcpy( void *restrict s1, const void *restrict s2, size_t n );
```

The specification of the `memcpy` function states that it cannot be used to copy between *overlapping* regions of memory. Using restricted pointers allows the compiler to see that requirement, and it can *optimize* the copy by copying multiple bytes at a time, which is more efficient. Incorrectly declaring a pointer as restricted when another pointer points to the same region of memory can result in *undefined behavior*. [For more information, see C99 Standard Section 6.7.3.1.]

F.12.3 Reliable Integer Division

[This section can be taught after Section 2.5.]

In compilers prior to C99, the behavior of integer division varies across implementations. Some implementations *round a negative quotient toward negative infinity*, while others *round toward zero*. When one of the integer operands is negative, this can result in different answers. Consider dividing –28 by 5. The exact answer is –5.6. If we round the quotient toward zero, we get the integer result of –5. If we round –5.6 toward negative infinity, we get an integer result of –6. C99 removes the ambiguity and *always* performs integer division (and integer modulus) by *rounding the quotient toward zero*. This makes integer division reliable—C99-compliant platforms all treat integer division in the same way. [For more information, see C99 Standard Section 6.5.5.]

F.12.4 Flexible Array Members

[This section can be taught after Section 10.3.]

C99 allows us to declare an *array of unspecified length* as the *last* member of a `struct`. Consider the following

```
struct s {
   int arraySize;
   int array[];
}; // end struct s
```

A **flexible array member** is declared by specifying empty square brackets (`[]`). To allocate a `struct` with a flexible array member, use code such as

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```
int desiredSize = 5;
struct s *ptr;
ptr = malloc( sizeof( struct s ) + sizeof( int ) * desiredSize );
```

The `sizeof` operator ignores flexible array members. The `sizeof( struct s )` phrase is evaluated as the size of all the members in a `struct s` *except* for the flexible array. The extra space we allocate with `sizeof( int ) * desiredSize` is the size of our flexible array.

There are many restrictions on the use of flexible array members. A flexible array member can be declared only as the *last* member of a `struct`—each `struct` can contain at most *one* flexible array member. Also, a flexible array cannot be the only member of a `struct`. The `struct` must also have *one or more* fixed members. Furthermore, any `struct` containing a flexible array member *cannot* be a member of another `struct`. Finally, a `struct` with a flexible array member cannot be *statically* initialized—it must be allocated *dynamically*. You cannot fix the size of the flexible array member at compile time. [For more information, see C99 Standard Section 6.7.2.1.]

F.12.5 Relaxed Constraints on Aggregate Initialization

[**This section can be taught after Section 10.3.**]
In C99, it's no longer required that aggregates such as arrays, structs, and unions be initialized by constant expressions. This enables the use of more concise initializer lists instead of using many separate statements to initialize members of an aggregate.

F.12.6 Type Generic Math

[**This section can be taught after Section 5.3.**]
The `<tgmath.h>` header is new in C99. It provides type-generic macros for many math functions in `<math.h>`. For example, after including `<tgmath.h>`, if `x` is a `float`, the expression `sin(x)` will call `sinf` (the `float` version of `sin`); if `x` is a `double`, `sin(x)` will call `sin` (which takes a `double` argument); if `x` is a `long double`, `sin(x)` will call `sinl` (the `long double` version of `sin`); and if `x` is a complex number, `sin(x)` will call the appropriate version of the `sin` function for that complex type (`csin`, `csinf` or `csinl`). C11 includes additional generics capabilities which we mention later in this appendix.

F.12.7 Inline Functions

[**This section can be taught after Section 5.5.**]
C99 allows the declaration of *inline functions* (as C++ does) by placing the keyword `inline` before the function declaration, as in:

```
inline void randomFunction();
```

This has *no effect* on the logic of the program from the user's perspective, but it can *improve performance*. Function calls take time. When we declare a function as `inline`, the program no longer calls that function. Instead, the compiler replaces every call to an inline function with a copy of the code body of that function. This improves the runtime performance but it may increase the program's size. Declare functions as `inline` *only* if they are short and called frequently. The `inline` declaration is only *advice* to the compiler, which can decide to ignore it. [For more information, see C99 Standard Section 6.7.4.]

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

F.12.8 Return Without Expression

[**This section can be taught after Section 5.5.**]
C99 adds tighter restrictions on returning from functions. In functions that return a non-void value, we are no longer permitted to use the statement

```
return;
```

In compilers prior to C99 this is allowed but results in *undefined behavior* if the caller tries to use the returned value of the function. Similarly, in functions that do not return a value, we are no longer permitted to return a value. Statements such as:

```
void returnInt() { return 1; }
```

are no longer allowed. C99 requires that compatible compilers produce warning messages or compilation errors in each of the preceding cases. [For more information, see C99 Standard Section 6.8.6.4.]

F.12.9 __func__ Predefined Identifier

[**This section can be taught after Section 13.5.**]
The `__func__` predefined identifier is similar to the `__FILE__` and `__LINE__` preprocessor macros—it's a string that holds the *name of the current function*. Unlike `__FILE__`, it's not a string literal but a real variable, so it cannot be concatenated with other literals. This is because string literal concatenation is performed during preprocessing, and the preprocessor has no knowledge of the semantics of the C language proper.

F.12.10 va_copy Macro

[**This section can be taught after Section 14.3.**]
Section 14.3 introduced the `<stdarg.h>` header and facilities for working with variable-length argument lists. C99 adds the `va_copy` macro, which takes two `va_list`s and copies its second argument into its first argument. This allows for multiple passes over a variable-length argument list without starting from the beginning each time.

F.13 New Features in the C11 Standard

C11 was approved as this book went to publication. C11 refines and expands the capabilities of C. At the time of this writing, most C compilers that support C11 implement only a *subset* of the new features. In addition, various new features are considered *optional* by the C11 standard. Microsoft Visual C++ does not support most features that were added in C99 and C11. Figure F.12 lists C compilers that have incorporated various C11 features.

Compiler	URL
GNU GCC	`gcc.gnu.org/gcc-4.7/changes.html`
LLVM	`clang.llvm.org/docs/ReleaseNotes.html#cchanges`
IBM XL C	`www-01.ibm.com/software/awdtools/xlcpp/`
Pelles C	`www.smorgasbordet.com/pellesc/`

Fig. F.12 | C11 compliant compilers.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

A pre-final draft of the standard document can be found at

```
www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
```

and the final standard document can be purchased at

```
webstore.ansi.org/RecordDetail.aspx?sku=
   INCITS%2FISO%2FIEC+9899-2012
```

F.13.1 New C11 Headers

Figure F.13 lists the new C11 standard library headers.

Standard library header	Explanation
`<stdalign.h>`	Provides type alignment controls.
`<stdatomic.h>`	Provides uninterruptible access to objects, used in multithreading.
`<stdnoreturn.h>`	Non-returning functions
`<threads.h>`	Thread library
`<uchar.h>`	UTF-16 and UTF-32 character utilities

Fig. F.13 | New C11 Standard Library header files

F.13.2 Multithreading Support

Multithreading is one of the most significant improvements in the C11 standard. Though multithreading has been around for decades, interest in it is rising quickly due to the proliferation of multicore systems—even smartphones and tablets are typically multicore now. The most common level of multicore processor today is dual core, though quad core processors are becoming popular. The number of cores will continue to grow. In multicore systems, the hardware can put multiple processors to work on different parts of your task, thereby enabling the tasks (and the program) to complete faster. To take the fullest advantage of multicore architecture you need to write multithreaded applications. When a program splits tasks into separate threads, a multicore system can run those threads in parallel.

Standard Multithreading Implementation

Previously, C multithreading libraries were non-standard, platform-specific extensions. C programmers often want their code to be portable across platforms. This is a key benefit of standardized multithreading. C11's `<threads.h>` header declares the new (optional) multithreading capabilities that, when implemented, will enable you to write more portable multithreaded C code. At the time of this writing, very few C compilers provide C11 multithreading support. For the examples in this section, we used the Pelles C compiler (Windows only), which you can download from `www.smorgasbordet.com/pellesc/`. In this section, we introduce the basic multithreading features that enable you to create and execute threads. At the end of the section we introduce several other multithreading features that C11 supports.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Running Multithreaded Programs

When you run any program on a modern computer system, your program's tasks compete for the attention of the processor(s) with the operating system, other programs and other activities that the operating system is running on your behalf. All kinds of tasks are typically running in the background on your system. When you execute the examples in this section, the time to perform each calculation will vary based on your computer's processor speed, number of processor cores and what's running on your computer. It's not unlike a drive to the supermarket. The time it takes you to drive there can vary based on traffic conditions, weather and other factors. Some days the drive might take 10 minutes, but during rush hour or bad whether it could take longer. The same is true for executing applications on computer systems.

There is also overhead inherent in multithreading itself. Simply dividing a task into two threads and running it on a dual core system does not run it twice as fast, though it will typically run faster than performing the thread's tasks in sequence. As you'll see, executing a multithreaded application on a single-core processor can actually take longer than simply performing the thread's tasks in sequence.

Overview of This Section's Examples

To provide a convincing demonstration of multithreading on a multicore system, this section presents two programs:

- One performs two compute-intensive calculations sequentially.
- These other executes the same compute-intensive calculations in parallel threads.

We executed each program on single-core *and* dual-core Windows 7 computers to demonstrate the performance of each program in each scenario. We timed each calculation and the total calculation time in both programs. The program outputs show the time improvements when the multithreaded program executes on a multicore system.

Example: Sequential Execution of Two Compute-Intensive Tasks

Figure F.14 uses the recursive `fibonacci` function (lines 37–46) that we introduced in Section 5.15. Recall that, for larger Fibonacci values, the recursive implementation can require significant computation time. The example sequentially performs the calculations `fibonacci(50)` (line 16) and `fibonacci(49)` (line 25). Before and after each `fibonacci` call, we capture the time so that we can calculate the total time required for the calculation. We also use this to calculate the total time required for both calculations. Lines 21, 30 and 33 use function `difftime` (from header `<time.h>`) to calculate the number of seconds between two times.

The first output shows the results of executing the program on a dual-core Windows 7 computer on which every execution produced the same results. The second and third outputs show the results of executing the program on a single-core Windows 7 computer on which the results varied, but always took longer to execute, because the processor was being shared between this program and all the others that happened to be executing on the computer at the same time.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```c
// Fig. F.14: fibonacci.c
// Fibonacci calculations performed sequentially
#include <stdio.h>
#include <time.h>

unsigned long long int fibonacci( unsigned int n ); // function prototype

// function main begins program execution
int main( void )
{
   puts( "Sequential calls to fibonacci(50) and fibonacci(49)" );

   // calculate fibonacci value for number input by user
   time_t startTime1 = time( NULL );
   puts( "Calculating fibonacci( 50 )" );
   unsigned long long int result1 = fibonacci( 50 );
   time_t endTime1 = time( NULL );

   printf( "fibonacci( %u ) = %llu\n", 50, result1 );
   printf( "Calculation time = %f minutes\n\n",
      difftime( endTime1, startTime1 ) / 60.0 );

   time_t startTime2 = time( NULL );
   puts( "Calculating fibonacci( 49 )" );
   unsigned long long int result2 = fibonacci( 49 );
   time_t endTime2 = time( NULL );

   printf( "fibonacci( %u ) = %llu\n", 49, result2 );
   printf( "Calculation time = %f minutes\n\n",
      difftime( endTime2, startTime2 ) / 60.0 );

   printf( "Total calculation time = %f minutes\n",
      difftime( endTime2, startTime1 ) / 60.0 );
} // end main

// Recursively calculates fibonacci numbers
unsigned long long int fibonacci( unsigned int n )
{
   // base case
   if ( 0 == n || 1 == n ) {
      return n;
   } // end if
   else { // recursive step
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
   } // end else
} // end function fibonacci
```

Fig. F.14 | Fibonacci calculations performed sequentially. (Part 1 of 2.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

a) Output on a Dual Core Windows 7 Computer

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci( 50 )
fibonacci( 50 ) = 12586269025
Calculation time = 1.550000 minutes

Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 0.966667 minutes

Total calculation time = 2.516667 minutes
```

b) Output on a Single Core Windows 7 Computer

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci( 50 )
fibonacci( 50 ) = 12586269025
Calculation time = 1.600000 minutes

Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 0.950000 minutes

Total calculation time = 2.550000 minutes
```

c) Output on a Single Core Windows 7 Computer

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci( 50 )
fibonacci( 50 ) = 12586269025
Calculation time = 1.550000 minutes

Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 1.200000 minutes

Total calculation time = 2.750000 minutes
```

Fig. F.14 | Fibonacci calculations performed sequentially. (Part 2 of 2.)

Example: Multithreaded Execution of Two Compute-Intensive Tasks

Figure F.15 also uses the recursive `fibonacci` function, but executes each call to `fibonacci` in a separate thread. The first two outputs show the multithreaded Fibonacci example executing on a dual-core computer. Though execution times varied, the total time to perform both Fibonacci calculations (in our tests) was always less than sequential execution in Fig. F.14. The last two outputs show the example executing on a single-core computer. Again, times varied for each execution, but the total time was *more* than the sequential execution due to the overhead of sharing *one* processor among all the program's threads and the other programs executing on the computer at the same time.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```c
// Fig. F.15: ThreadedFibonacci.c
// Fibonacci calculations performed in separate threads
#include <stdio.h>
#include <threads.h>
#include <time.h>

#define NUMBER_OF_THREADS 2

int startFibonacci( void *nPtr );
unsigned long long int fibonacci( unsigned int n );

typedef struct ThreadData {
   time_t startTime; // time thread starts processing
   time_t endTime; // time thread finishes processing
   unsigned int number; // fibonacci number to calculate
} ThreadData; // end struct ThreadData

int main( void )
{
   // data passed to the threads; uses designated initializers
   ThreadData data[ NUMBER_OF_THREADS ] =
      { [ 0 ] = { .number = 50 },
        [ 1 ] = { .number = 49 } };

   // each thread needs a thread identifier of type thrd_t
   thrd_t threads[ NUMBER_OF_THREADS ];

   puts( "fibonacci(50) and fibonacci(49) in separate threads" );

   // create and start the threads
   for ( unsigned int i = 0; i < NUMBER_OF_THREADS; ++i ) {
      printf( "Starting thread to calculate fibonacci( %d )\n",
         data[ i ].number );

      // create a thread and check whether creation was successful
      if ( thrd_create( &threads[ i ], startFibonacci, &data[ i ] ) !=
         thrd_success ) {

         puts( "Failed to create thread" );
      } // end if
   } // end for

   // wait for each of the calculations to complete
   for ( int i = 0; i < NUMBER_OF_THREADS; ++i )
      thrd_join( threads[ i ], NULL );

   // determine time that first thread started
   time_t startTime = ( data[ 0 ].startTime > data[ 1 ].startTime ) ?
      data[ 0 ].startTime : data[ 1 ].startTime;

   // determine time that last thread terminated
   time_t endTime = ( data[ 0 ].endTime > data[ 1 ].endTime ) ?
      data[ 0 ].endTime : data[ 1 ].endTime;
```

Fig. F.15 | Fibonacci calculations performed in separate threads. (Part 1 of 3.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

```
54
55       // display total time for calculations
56       printf( "Total calculation time = %f minutes\n",
57          difftime( endTime, startTime ) / 60.0 );
58    } // end main
59
60    // Called by a thread to begin recursive Fibonacci calculation
61    int startFibonacci( void *ptr )
62    {
63       // cast ptr to ThreadData * so we can access arguments
64       ThreadData *dataPtr = (ThreadData *) ptr;
65
66       dataPtr->startTime = time( NULL ); // time before calculation
67
68       printf( "Calculating fibonacci( %d )\n", dataPtr->number );
69       printf( "fibonacci( %d ) = %lld\n",
70          dataPtr->number, fibonacci( dataPtr->number ) );
71
72       dataPtr->endTime = time( NULL ); // time after calculation
73
74       printf( "Calculation time = %f minutes\n\n",
75          difftime( dataPtr->endTime, dataPtr->startTime ) / 60.0 );
76       return thrd_success;
77    } // end function startFibonacci
78
79    // Recursively calculates fibonacci numbers
80    unsigned long long int fibonacci( unsigned int n )
81    {
82       // base case
83       if ( 0 == n || 1 == n ) {
84          return n;
85       } // end if
86       else { // recursive step
87          return fibonacci( n - 1 ) + fibonacci( n - 2 );
88       } // end else
89    } // end function fibonacci
```

a) Output on a Dual Core Windows 7 Computer

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci( 50 )
Starting thread to calculate fibonacci( 49 )
Calculating fibonacci( 50 )
Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 1.066667 minutes

fibonacci( 50 ) = 12586269025
Calculation time = 1.700000 minutes

Total calculation time = 1.700000 minutes
```

Fig. F.15 | Fibonacci calculations performed in separate threads. (Part 2 of 3.)

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

b) Output on a Dual Core Windows 7 Computer

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci( 50 )
Starting thread to calculate fibonacci( 49 )
Calculating fibonacci( 50 )
Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 0.683333 minutes

fibonacci( 50 ) = 12586269025
Calculation time = 1.666667 minutes

Total calculation time = 1.666667 minutes
```

c) Output on a Single Core Windows 7 Computer

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci( 50 )
Starting thread to calculate fibonacci( 49 )
Calculating fibonacci( 50 )
Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 2.116667 minutes

fibonacci( 50 ) = 12586269025
Calculation time = 2.766667 minutes

Total calculation time = 2.766667 minutes
```

d) Output on a Single Core Windows 7 Computer

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci( 50 )
Starting thread to calculate fibonacci( 49 )
Calculating fibonacci( 50 )
Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 2.233333 minutes

fibonacci( 50 ) = 12586269025
Calculation time = 2.950000 minutes

Total calculation time = 2.950000 minutes
```

Fig. F.15 | Fibonacci calculations performed in separate threads. (Part 3 of 3.)

`struct ThreadData`

The function that each thread executes in this example receives a `ThreadData` object as its argument. This object contains the `number` that will be passed to `fibonacci` and two `time_t` members where we store the time before and after each thread's `fibonacci` call. Lines 21–23 create an array of the two `ThreadData` objects and use designated initializers to set their `number` members to 50 and 49, respectively.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

`thrd_t`

Line 26 creates an array of `thrd_t` objects. When you create a thread, the multithreading library creates a *thread ID* and stores it in a `thrd_t` object. The thread's ID can then be used with various multithreading functions.

Creating and Executing a Thread

Lines 31–41 create two threads by calling function `thrd_create` (line 36). The functions three arguments are:

- A `thrd_t` pointer that `thrd_create` uses to store the thread's ID.
- A pointer to a function (`startFibonacci`) that specifies the task to perform in the thread. The function must return an `int` and receive a `void` pointer representing the argument to the function (in this case, a pointer to a `ThreadData` object). The `int` represents the thread's state when it terminates (e.g., `thrd_success` or `thrd_error`).
- A `void` pointer to the argument that should be passed to the function in the second argument.

Function `thrd_create` returns `thrd_success` if the thread is created, `thrd_nomem` if there was not enough memory to allocate the thread or `thrd_error` otherwise. If the thread is created successfully, the function specified as the second argument begins executing in the new thread.

Joining the Threads

To ensure that the program does not terminate until the threads terminate, lines 44–45 call `thrd_join` for each thread that we created. This causes the program to *wait* until the threads complete execution before executing the remaining code in `main`. Function `thrd_join` receives the `thrd_t` representing the ID of the thread to join and an `int` pointer where `thrd_join` can store the status returned by the thread.

Function `startFibonacci`

Function `startFibonacci` (lines 61–77) specifies the task to perform—in this case, to call `fibonacci` to recursively perform a calculation, to time the calculation, to display the calculation's result and to display the time the calculation took (as we did in Fig. F.14). The thread executes until `startFibonacci` returns the thread's status (`thrd_success`; line 76), at which point the thread terminates.

Other C11 Multithreading Features

In addition to the basic multithreading support shown in this section, C11 also includes other features such as `_Atomic` variables and atomic operations, thread local storage, conditions and mutexes. For more information on these topics, see Sections 6.7.2.4, 6.7.3, 7.17 and 7.26 of the standard and the following blog post and article:

```
blog.smartbear.com/software-quality/bid/173187/
   C11-A-New-C-Standard-Aiming-at-Safer-Programming
lwn.net/Articles/508220/
```

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

F.13.3 quick_exit function

In addition to the functions exit (Section 14.6) and abort, C11 now also supports quick_exit for terminating a program—all three functions are declared in the header <stdlib.h>. Like exit, you call quick_exit and pass it an *exit status* as an argument—typically EXIT_SUCCESS or EXIT_FAILURE, but other platform-specific values are possible. The exit status value is returned from the program to the calling environment to indicate whether the program terminated successfully or an error occurred. When called, quick_exit can, in turn, call up to 32 other functions to perform cleanup tasks. You register these functions with the at_quick_exit function (similar to atexit in Section 14.6) and are called in the *reverse* order from which they were registered. Each registered function must return void and have a void parameter list. The motivation for functions quick_exit and at_quick_exit is explained at

```
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1327.htm
```

F.13.4 Unicode® Support

Internationalization and localization is the process of creating software that supports *multiple spoken languages* and *locale-specific requirements*—such as, displaying monetary formats. The Unicode® character set contains characters for many of the world's languages and symbols.

C11 now includes support for both the *16-bit (UTF-16)* and *32-bit (UTF-16)* Unicode character sets, which makes it easier for you to internationalize and localize your apps. Section 6.4.5 in the C11 standard discusses how to create Unicode string literals. Section 7.28 in the standard discusses the features of the new Unicode utilities header (<uchar.h>), which include the new types char16_t and char32_t for UTF-16 and UTF-32 characters, respectively. At the time of this writing, the new Unicode features are *not* widely supported among C compilers.

F.13.5 _Noreturn Function Specifier

The *_Noreturn function specifier* indicates that a function will *not* return to its caller. For example, function exit (Section 14.6) terminates a program, so it does *not* return to its caller. Such functions in the C Standard Library are now declared with _Noreturn. For example, the C11 standard shows function exit's prototype as:

```
_Noreturn void exit(int status);
```

If the compiler knows that a function does *not* return, it can perform various *optimizations*. It can also issue error messages if a _Noreturn function is inadvertently written to return.

F.13.6 Type-Generic Expressions

C11's new _Generic keyword provides a mechanism that you can use to create a macro (Chapter 13) that can invoke different type-specific versions of functions based on the macro's argument type. In C11, this is now used to implement the features of the type-generic math header (<tgmath.h>). Many math functions provide separate versions that take as arguments floats, doubles or long doubles. In such cases, there is a macro that automatically invokes the corresponding type-specific version. For example, the macro ceil invokes the function ceilf when the argument is a float, ceil when the argument

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

is a `double` and `ceill` when the argument is a `long double`. Implementing your own macros using `_Generic` is an advanced concept that's beyond the scope of this book. Section 6.5.1.1 of the C11 standard discusses the details of `_Generic`.

F.13.7 Annex L: Analyzability and Undefined Behavior

The C11 standard document defines the features of the language that compiler vendors must implement. Because of the extraordinary range of hardware and software platforms and other issues, there's a number of places where the standard specifies that the result of an operation is *undefined behavior*. These can raise security and reliability concerns—every time there's an undefined behavior something happens that could leave a system open to attack or failure. We searched the C11 standard document for the term "undefined behavior"—it appears approximately 50 times.

The people from CERT (`cert.org`) who developed C11's optional Annex L on analyzability scrutinized all undefined behaviors and discovered that they fall into two categories—those for which compiler implementers should be able to do something reasonable to avoid serious consequences (known as *bounded undefined behaviors*), and those for which implementers would not be able to do anything reasonable (known as *critical undefined behaviors*). It turned out that most undefined behaviors belong to the first category. David Keaton (a researcher on the CERT Secure Coding Program) explains the categories in the following article:

```
blog.sei.cmu.edu/post.cfm/improving-security-in-the-latest-c-
   programming-language-standard-1
```

The C11 standard's Annex L identifies the critical undefined behaviors. Including this annex as part of the standard provides an opportunity for compiler implementations—a compiler that's Annex L compliant can be depended upon to do something reasonable for most of the undefined behaviors that might have been ignored in earlier implementations. Annex L still does not guarantee reasonable behavior for critical undefined behaviors. A program can determine whether the implementation is Annex L compliant by using conditional compilation directives (Section 13.5) to test whether the macro `__STDC_ANALYZABLE__` is defined.

F.13.8 Anonymous Structures and Unions

Chapter 10 introduced `structs` and `unions`. C11 now supports anonymous `structs` and `unions` that can be nested in named `structs` and `unions`. The members in a nested anonymous `struct` or `union` are considered to be members of the enclosing `struct` or `union` and can be accessed directly through an object of the enclosing type. For example, consider the following `struct` declaration:

```
struct MyStruct {
   int member1;
   int member2;

   struct {
      int nestedMember1;
      int nestedMember2;
   }; // end nested struct
}; // end outer struct
```

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

For a variable `myStruct` of type `struct MyStruct`, you can access the members as:

```
myStruct.member1;
myStruct.member2;
myStruct.nestedMember1;
myStruct.nestedMember2;
```

F.13.9 Memory Alignment Control

In Chapter 10, we discussed the fact that computer platforms have different boundary alignment requirements, which could lead to `struct` objects requiring more memory than the total of their members' sizes. C11 now allows you to specify the boundary alignment requirements of any type using features of the `<stdalign.h> header`. `_Alignas` is used to specify alignment requirements. Operator `alignof` returns the alignment requirement for its argument. Function `aligned_alloc` allows you to dynamically allocate memory for an object and specify its alignment requirements. For more details see Section 6.2.8 of the C11 standard document.

F.13.10 Static Assertions

In Section 13.10, you learned that C's `assert` macro tests the value of an expression at execution time. If the condition's value is false, `assert` prints an error message and calls function `abort` to terminate the program. This is useful for debugging purposes. C11 now provides `_Static_assert` for compile-time assertions that test constant expressions after the preprocessor executes and at a point when the types of expressions are known. For more details see Section 6.7.10 of the C11 standard document.

F.13.11 Floating Point Types

C11 is now compatible with the IEC 60559 floating-point arithmetic standard, though support for this is optional.

F.14 Web Resources

C99 Resources

`www.open-std.org/jtc1/sc22/wg14/`
Official site for the C standards committee. Includes defect reports, working papers, projects and milestones, the rationale for the C99 standard, contacts and more.

`blogs.msdn.com/vcblog/archive/2007/11/05/iso-c-standard-update.aspx`
Blog post of Arjun Bijanki, the test lead for the Visual C++ compiler. Discusses why C99 is not supported in Visual Studio.

`www.ibm.com/developerworks/linux/library/l-c99/index.html`
Article: "Open Source Development Using C99," by Peter Seebach. Discusses C99 library features on Linux and BSD.

`www.informit.com/guides/content.aspx?g=cplusplus&seqNum=215`
Article: "A Tour of C99," by Danny Kalev. Summarizes some of the new features in the C99 standard.

C11 Standard

`webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2FISO%2FIEC+9899-2012`
Purchase the ANSI variant of the C11 standard.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

`www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`
This is the last *free* draft of the C11 standard before it was approved and published.

What's New in C11

`en.wikipedia.org/wiki/C11_(C_standard_revision)`
The Wikipedia page for the new C11 standard describes what's new since C99.

`progopedia.com/dialect/c11/`
This page includes a brief listing of the new features in C11.

`www.informit.com/articles/article.aspx?p=1843894`
The article, "The New Features of C11," by David Chisnall.

`www.drdobbs.com/cpp/c-finally-gets-a-new-standard/232800444`
The article, "C Finally Gets a New Standard," by Thomas Plum. Discusses concurrency, keywords, the `thread_local` storage class, optional threads and more.

`www.drdobbs.com/cpp/cs-new-ease-of-use-and-how-the-language/240001401`
The article, "C's New Ease of Use and How the Language Compares with C++," by Tom Plum. Discusses some of the new C11 features that match features in C++, and a few key differences in C11 that have no corresponding features in C++.

`www.i-programmer.info/news/98-languages/3546-new-iso-c-standard-c1x.html`
The article, "New ISO C standard - C11," by Mike James. Briefly discusses some of the new features.

`www.drdobbs.com/cpp/the-new-c-standard-explored/232901670`
The article, "The New C Standard Explored," by Tom Plum. Discusses the C11 Annex K functions, `fopen()` safety, fixing `tmpnam`, the `%n` formatting vulnerability, security improvements and more.

`m.drdobbs.com/144530/show/871e182fd14035dc243e815651ffaa79&t=qkoob97760e69a0dopqejjech4`
The article, "C's New Ease of Use and How the Language Compares with C++." Topics include alignment, Unicode strings and constants, type-generic macros, ease-of-use features and C++ compatibility.

`www.sdtimes.com/link/36892`
The article, "The thinking behind C11," by John Benito, the convener of the ISO working group for the C programming language standard. The article discusses the C programming language standard committee's guiding principles for the new C11 standard.

Improved Security

`blog.smartbear.com/software-quality/bid/173187/C11-A-New-C-Standard-Aiming-at-Safer-Programming`
The blog, "C11: A New C Standard Aiming at Safer Programming," by Danny Kalev. Discusses the problems with the C99 standards and new hopes with the C11 standard in terms of security.

`www.amazon.com/exec/obidos/ASIN/0321335724/deitelassociatin`
The book, *Secure Coding in C and C++*, by Robert Seacord, discusses the security benefits of the Annex K library.

`blog.sei.cmu.edu/post.cfm/improving-security-in-the-latest-c-programming-language-standard-1`
The blog, "Improving Security in the Latest C Programming Language Standard," by David Keaton of the CERT Secure Coding Program at Carnegie Mellon's Software Engineering Institute. Discusses bounds checking interfaces and analyzability.

`blog.sei.cmu.edu/post.cfm/helping-developers-address-security-with-the-cert-c-secure-coding-standard`
The blog, "Helping Developers Address Security with the CERT C Secure Coding Standard," by David Keaton. Discusses hoe C has handled security issues over the years and the CERT C Secure Coding Rules.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Bounds Checking

`www.securecoding.cert.org/confluence/display/seccode/ERR03-C.+Use+runtime-constraint+handlers+when+calling+the+bounds-checking+interfaces`
Carnegie Mellon's Software Engineering Institute's post, "ERR03-C. Use `runtime-constraint` handlers when calling the bounds-checking interfaces," by David Svoboda. Provides examples of noncompliant and compliant code.

Multithreading

`stackoverflow.com/questions/8876043/multi-threading-support-in-c11`
The forum discussion, "Multi-Threading support in C11." Discusses the improved memory sequencing model in C11 vs C99.

`www.t-dose.org/2012/talks/multithreaded-programming-new-c11-and-c11-standards`
The slide presentation, "Multithreaded Programming with the New C11 and C++11 Standards," by Klass van Gend. Introduces the new features of both the C11 and C++11 languages and discuss how far `gcc` and clang are implementing the new standards.

`www.youtube.com/watch?v=UqTirRXe8vw`
The video, "Multithreading Using Posix in C Language and Ubuntu," with Ahmad Naser.

`supertech.csail.mit.edu/cilk/lecture-2.pdf`
The lecture, "Multithreaded Programming in Cilk," by Charles E. Leiserson.

`fileadmin.cs.lth.se/cs/Education/EDAN25/F06.pdf`
The slide presentation, "Threads in the Next C Standard," by Jonas Skeppstedt.

`www.youtube.com/watch?v=gRe6Zh2M3zs`
A video of Klaas van Gend discussing multithreaded programming with the new C11 and C++11 standards.

`www.experiencefestival.com/wp/videos/c11-concurrency-part-7/4zWbQRE3tWk`
A series of videos on C11 concurrency.

Compiler Support

`www.ibm.com/developerworks/rational/library/support-iso-c11/support-iso-c11-pdf.pdf`
The whitepaper, "Support for ISO C11 added to IBM XL C/C++ compilers: New features introduced in Phase 1." Provides an overview of the new features supported by the compiler including complex value initialization, static assertions and functions that do not return.

© 2013 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved