17

Introduction to C99

There is nothing like returning to a place that remains unchanged to find the ways in which you yourself have altered.

—Nelson Mandela

Everything you can imagine is real.

—Pablo Picasso

...life is too full of rich possibilities to have restrictions placed upon it.

—Gloria Swanson

The return from your work must be the satisfaction which that work brings you and the world's need of that work.

—William E.B. Du Bois

Just never forget to be dexterous and deft
And never mix up your right foot with your left.

—Dr. Seuss

OBJECTIVES

In this chapter you will learn:

- Many of the new features of the C99 standard.
- Some key C99 features in the context of complete working programs.
- To use `//` comments.
- To mix declarations and executable code and to declare variables in `for` statement headers.
- To initialize arrays and `structs` with designated initializers.
- To use data type `bool` to create boolean variables whose data values can be `true` or `false`.
- To create variable-length arrays and pass them to functions.
- To perform arithmetic operations on complex variables.

Outline

17.1 Introduction

- **C99**
  - **C99 is a revised standard for the C programming language**
    - **Refines and expands the capabilities of C89 (ANSI C)**
  - **C99 has not been widely adopted**
    - **Not all compilers support it**
  - **In this chapter, we discuss many of C99's key features**

17.2 Support for C99

- **Microsoft Visual C++ does not support C99**
- **Instead, we use Bloodshed Software's Dev-C++**
  - **Free compiler**
  - **Available at `www.bloodshed.net/dev/devcpp.html`**
    - **To enable C99 support in Dev-C++, open the "Compiler Options" window in the "Tools" menu**
    - **Check the box next to "Add the following commands when calling compiler"**
    - **Add `-std=c99` to the text box underneath**

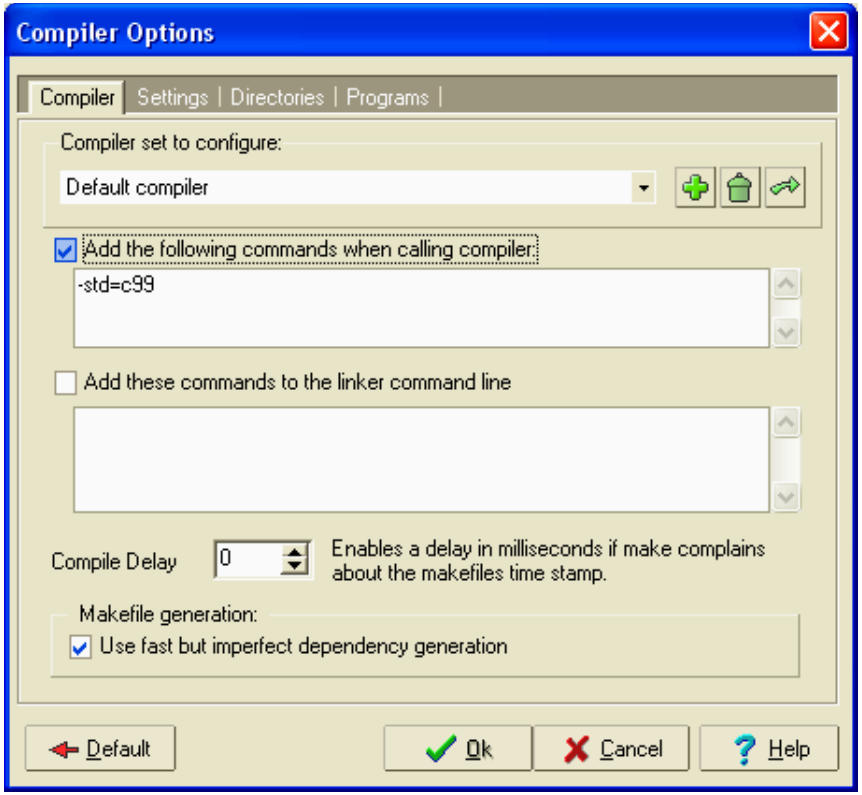

Fig. 17.1 | Compiler Options window in Dev-C++ 4.9.9.2.

17.3 New C99 Headers

- **C99 adds several headers for use with its new features**
  - **List of new headers on next slide**

Standard library	Explanation
`<complex.h>`	Contains macros and function prototypes for supporting complex numbers (see Section 17.10). [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 17.8). [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 17.12). [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. 17.2 | Standard library headers added in C99 and C95.

17.4 // Comments

- **// Comments**
  - **In C89, only way to declare comments is with /* and */**
  - **C99 allows the use of // for comments**
  - **Any text following // is considered a comment**
  - **However, // comments end at end of line**

17.5 Mixing Declarations and Executable Code

- **Block scope variable declarations**
  - **In C89, all variables with block scope must be declared at start of block**
  - **In C99, variables can be declared anywhere within a block**
  - **However, must make sure to declare variables before they are used**

Outline

fig17_03.c

```
1  // Fig 17.3: fig17_03.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
13    return 0; // indicates successful termination
14 } // end main
```

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

y is defined in middle of block without error

Outline

fig17_04.c

```
1  // Fig 17.4: fig17_04.c
2  // Attempting to declare a variable after its use in C99
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int x = 1; // declare x
8
9     printf( "The value of x is %d\n", x ); // output x
10    printf( "The value of y is %d\n", y ); // output y (error)
11
12    int y = 2; // declare y
13
14    return 0; // indicates successful termination
15 } // end main
```

y has not yet been declared at this point in the code, so this line causes an error

```
../examples/ch17/fig17_03.c: In function `main':
../examples/ch17/fig17_03.c:10: error: `y' undeclared
   (first use in this function)
../examples/ch17/fig17_03.c:10: error:
   (Each undeclared identifier is reported only once
../examples/ch17/fig17_03.c:10: error: for each function it appears in.)
```

17.6 Declaring a Variable in a `for` Statement Header

- **Variable declarations**
  - **In C89, all variables with block scope must be declared at start of block, so `for` loop counters must be defined before the loop itself**
  - **C99's capability to mix declarations and executable code allows programmers to define `for` loop counters inside the loop header**
  - **Variable declared in a `for` loop header has scope only for duration of loop**

Outline

fig17_05.c

```
1  // Fig 17.5: fig17_05.c
2  // Declaring a loop counter before a for statement in C89
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int x; // declare loop counter
8
9     // output values 1 through 5
10    printf( "Values of x\n" );
11    for ( x = 1; x <= 5; x++ ) // initialize loop counter
12       printf( "%d\n", x );
13
14    printf( "Value of x is %d\n", x ); // x is still in scope
15
16    return 0; // indicates successful termination
17 } // end main
```

In C89, **for** loop counters must be declared before **for** loop actually starts

```
Values of x
1
2
3
4
5
Value of x is 6
```

Outline

fig17_06.c

```
1  // Fig 17.6: fig17_06.c
2  // Declaring a variable in a for statement header in C99
3  #include <stdio.h>
4
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
13    return 0; // indicates successful termination
14 } // end main
```

In C99, loop counters can be declared in **`for`** loop header

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

Outline

fig17_07.c

```
1  // Fig 17.7: fig17_07.c
2  // Accessing a for statement variable after the for statement in C99
3  #include <stdio.h>
4
5  int main( void )
6  {
7     printf( "Values of x:\n" );
8
9     // declare variable in for statement header
10    for ( int x = 1; x <= 5; x++ )
11       printf( "%d\n", x );
12
13    printf( "Value of x is: %d\n", x ); // x is out of scope
14
15    return 0; // indicates successful termination
16 } // end main
```

```
../examples/ch17/fig17_07.c: In function `main':
../examples/ch17/fig17_07.c:13: error: `x' undeclared
   (first use in this function)
../examples/ch17/fig17_07.c:13: error:
   (Each undeclared identifier is reported only once
../examples/ch17/fig17_07.c:13: error: for each function it appears in.)
```

Variables defined in a **`for`** loop header have scope for only the loop, so this line causes an error

17.7 Designated Initializers and Compound Literals

- **Designated initializers**
  - **In C89, cannot initialize only selected elements of an array, `struct`, or `union`**
    - **Can initialize all elements, or first few elements, but cannot initialize, for example, only last element**
  - **C99's designated initializers allow initialization of only selected elements**

Outline

fig17_08.c

```
1  // Fig 17.8: fig17_08.c
2  // Assigning elements of an array in C89
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
17     // output array contents
18    printf( "The array is \n" );
19    for ( i = 0; i < 5; i++ )
20       printf( "%d\n", a[ i ] );
21
22    return 0; // indicates successful termination
23 } // end main
```

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

In C89, we can initialize the first element of an array in the declaration, but cannot initialize any one of the others without initializing all the elements before it

Must explicitly assign value to last element after array declaration

Outline

fig17_09.c

```
1  // Fig 17.9: fig17_09.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    for ( int i = 0; i < 5; i++ )
17       printf( "%d\n", a[ i ] );
18
19    return 0; // indicates successful termination
20 } // end main
```

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

Separate designated initializers with commas

C99's designated initializers allow only specific elements to be initialized

17.7 Designated Initializers and Compound Literals

- **Compound literals**
  - **Designated initializers can be used to create what are called “compound literals”**
  - **Allows programmer to create unnamed temporary arrays, `structs`, and `unions`**
  - **If, for example, programmer wants to pass a temporary array to function `demoFunction`, he uses the code**

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

Outline

fig17_10.c

(1 of 2)

```c
// Fig. 17.10: fig17_10.c
// Using designated initializers to initialize an array of structs in C99
#include <stdio.h>

struct twoInt // declare a struct of two integers
{
   int x;
   int y;
}; // end struct twoInt

int main( void )
{
   // explicitly initialize elements of array a
   // then explicitly initialize members of each struct element
   struct twoInt a[ 5 ] =
   {
      [ 0 ] = { .x = 1, .y = 2 },
      [ 4 ] = { .x = 10, .y = 20 }
   };
```


Designated initializers can also be used to initialize specific elements of **`struct`**s, like this

```
20
21    // output array contents
22    printf( "\nx\ty\n" );
23    for ( int i = 0; i <= 4; i++ )
24       printf( "%d\t%d\n", a[ i ].x, a[ i ].y );
25
26    return 0; // indicates successful termination
27 } //end main
```

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

Outline

fig17_10.c

(2 of 2)

17.8 Type `bool`

- **Boolean variables**
  - **"True or false" variables**
  - **Can hold only value of 0 or 1**
  - **C89 has no Boolean variable type—considers 0 to be "false" and any other value to be "true"**
  - **C99 provides Boolean variable type `_Bool`**
  - **Also provides `stdbool.h` header that defines three symbolic constants**
    - **`bool`, which evaluates to `_Bool`**
    - **`true`, which evaluates to `1`**
    - **`false`, which evalutates to `0`**

Outline

```c
// Fig 17.11: fig17_11.c
// Using the boolean type and the values true and false in C99.
#include <stdio.h>
#include <stdbool.h> // allows the use of bool, true, and false

bool isEven( int number ); // function prototype

int main( void )
{
   int input; // value entered by user
   bool valueIsEven; // stores result of function isEven

   // loop for 2 inputs
   for ( int i = 0; i < 2; i++ ) {
      printf( "Enter an integer: " );
      scanf( "%d", &input );

      valueIsEven = isEven( input ); // determine whether input is even

      // determine whether input is even
      if ( valueIsEven ) {
         printf( "%d is even \n\n", input );
      } // end if
      else {
         printf( "%d is odd \n\n", input );
      } // end else
   } // end for

   return 0;
} // end main
```

Include **`stdbool.h`** header to use **`bool`**, **`true`**, and **`false`**

`fig17_11.c`

(1 of 2)

`bool` type variable can hold only 0 or 1

Remember that C considers 0 to be false and non-zero to be true

Outline

fig17_11.c

(2 of 2)

```
31
32 // even returns true if number is even
33 bool isEven( int number )
34 {
35    if ( number % 2 == 0 ) { // is number divisible by 2?
36       return true;
37    }
38    else {
39       return false;
40    }
41 } // end function isEven
```

true symbolic constant evaluates to 1

false symbolic constant evaluates to 0

```
Enter an integer: 34
34 is even

Enter an integer: 23
23 is odd
```

17.9 Implicit `int` in Function Declarations

- **Implicit return types**
  - **In C89, if a function does not have a specified return type or one of its arguments does not have a type, they are implicitly given a type of `int`**
  - **In C99 this is not allowed; a function must be given a return type and its arguments must all have specified types or a compiler warning will be issued**

Outline

fig17_12.c

(1 of 2)

```
1  // Fig 17.12: fig17_12.c
2  // Using implicit int in C89
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
13    // assign data of unspecified return type to int
14    x = returnImplicitInt();
15
16    // pass in an int to a function with an unspecified type
17    y = demoImplicitInt(82);
18
19
20    printf( "x is %d\n", x );
21    printf( "y is %d\n", y );
22    return 0; // indicates successful termination
23
24 } // end main
```

In C89 functions with no return type default to type **int**

Likewise, arguments without a type default to type **int**

Outline

```
25
26 returnImplicitInt()
27 {
28    return 77; // returning an int when return type is not specified
29 } // end function returnImplicitInt
30
31 int demoImplicitInt(x)
32 {
33    return x;
34 } // end function demoImplicitInt
```

fig17_12.c

(2 of 2)

Outline

```
..\examples\ch17\fig17_12.c:5: warning: type defaults to `int' in declaration
   of `returnImplicitInt'
..\examples\ch17\fig17_12.c:5: warning: data definition has no type or
   storage class
..\examples\ch17\fig17_12.c:6: warning: parameter names (without types) in
   function declaration
..\examples\ch17\fig17_12.c:27: warning: return type defaults to `int'
..\examples\ch17\fig17_12.c:32: warning: return type defaults to `int'
..\examples\ch17\fig17_12.c: In function `demoImplicitInt':
..\examples\ch17\fig17_12.c:32: warning: type of "x" defaults to "int"
```


In C89 using implicit **`int`**s is allowed, but in C99 compiler warnings will occur

17.10 Complex Numbers

- **Complex numbers**
  - **C99 adds support for complex numbers and arithmetic using the `complex.h` header**
  - **Defines symbolic constant `I` that corresponds to *i*, the square root of -1**
  - **Complex variables defined using `complex` type**
  - **Note that `complex` can be combined with other types, e.g. `double complex`**
  - **All arithmetic operations in C with complex numbers are done with same operators as regular numbers**
  - **Exponents done with `cpow` function in `complex.h` header**
  - **Function creal returns real portion of a complex number, and function cimag returns imaginary portion**

Outline

fig17_14.c

(1 of 2)

```
1  // Fig 17.14: fig17_14.c
2  // Using complex numbers in C99
3  #include <stdio.h>
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
```

Include **complex.h** header to use complex numbers

Declare complex variables with **complex** type

cpow function performs exponentiation with complex numbers

Outline

fig17_14.c

(2 of 2)

```
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
23    return 0; // indicates successful termination
24 } // 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
```

creal function returns real portion of a complex number

cimag function returns imaginary portion of a complex number

17.11 Variable-Length Arrays

- **Variable-length arrays**
  - **Does *not* refer to arrays whose size can change after they have been declared**
  - **In C89, the size of an array must be a constant**
  - **C99 allows array sizes to be set by variables**
  - **Functions can also take arrays of variable length**
  - **If function does not need to know the size of an array passed to it, array should be declared in function header as having size ***

Outline

fig17_15.c

(1 of 2)

```
1  // Fig 17.15: fig17_15.c
2  // Using variable-length arrays in C99
3  #include <stdio.h>
4
5  void printArray( int sz, int arr[ sz ] ); // function that accepts a VLA
6
7  int main( void )
8  {
9     int arraySize; // size of array
10    printf( "Enter array size in words: " );
11    scanf( "%d", &arraySize );
12
13    // using a non-constant expression for array's size
14    int array[ arraySize ]; // declare variable-length array
15    // test sizeof operator on VLA
16    int a = sizeof( array );
17    printf( "\nsizeof yields array size of %d bytes\n\n", a );
18
19    // assign elements of VLA
20    for ( int i = 0; i < arraySize; i++ ) {
21       array[ i ] = i * i;
22    } // end for
23
24    printArray( arraySize, array ); // pass VLA to function
25
26    return 0; // indicates successful termination
27 } // end main
```

Note that the array's size is a variable entered by the user

```
28
29 void printArray( int size, int array[ size ] )
30 {
31    // output contents of array
32    for ( int i = 0; i < size; i++ ) {
33       printf( "array[%d] = %d\n", i, array[ i ] );
34    } // end for
35 } // end function printArray
```

```
Enter array size in words: 7

sizeof yields array size of 28 bytes

array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25
array[6] = 36
```

Function takes a variable-length array—note that the value of **`size`** must also be passed

Outline

`fig17_15.c`

(2 of 2)

17.12 Other C99 Features

- **Extended Identifiers**
  - **In C89, identifiers for variable names must be 31 characters or less for variables with internal linkage and 6 characters or less for variables with external linkage**
  - **C99 increases this limit to 63 characters for internally-linked variables and 31 characters for externally-linked variables**
- **The `restrict` keyword**
  - **Declares restricted pointers—gives a portion of a program exclusive access to that location in memory**
  - **Example for restricted pointer to an integer:**

```
int *restrict ptr;
```

17.12 Other C99 Features

- **Reliable Integer Division**
  - In C89, the behavior of integer division when the quotient is negative is implementation-dependent
    - Some compilers round negative quotients toward 0, while others round the other way, toward negative infinity
  - C99 always rounds negative quotients toward 0
- **Flexible Array Members**
  - C99 allows `structs` to contain an array of unspecified length
    - Must be last member of `struct`, and cannot be only member
    - Memory for the array must be dynamically allocated

17.12 Other C99 Features

- `long long int` Type
  - C99 defines type `long long int` for use on 64-bit systems
- Type Generic Math
  - The functions in C89's `math.h` header only take one type
  - C99 contains the `tgmath.h` header which contains math functions for all variable types
  - Function prototypes are the same as those in `math.h`, but they will automatically call the correct version of the function depending on the argument type
    - For example, if user calls `sin(x)`, and `x` is a `float` (`sin` normally takes `double`s), the function will instead call the `tgmath.h` function `sinf`, which takes `float`s

17.12 Other C99 Features

- **Inline Functions**
  - C99 allows inline functions (a feature of C++)
  - Declared by placing `inline` before function declaration
  - If a function is declared inline, the compiler will replace all calls to it with the function's code body
  - Improves performance, but may increase program's size
- **Return Without Expression**
  - In C99, non-`void` functions must return a value
    - Cannot use statement `return;`
    - C89 allowed this but it could result in undefined behavior
  - Likewise, `void` functions cannot return a value

17.12 Other C99 Features

- **Function `snprintf`**
  - **C89 function `sprintf` writes into a buffer, but does not know how large the buffer is**
    - **If text is too long, a "buffer overflow" occurs and information in memory following the buffer could be destroyed**
  - **C99's `snprintf` function takes the buffer size as an argument, preventing buffer overflows**
  - **Most C compilers support this function, even those that are not C99-compliant**