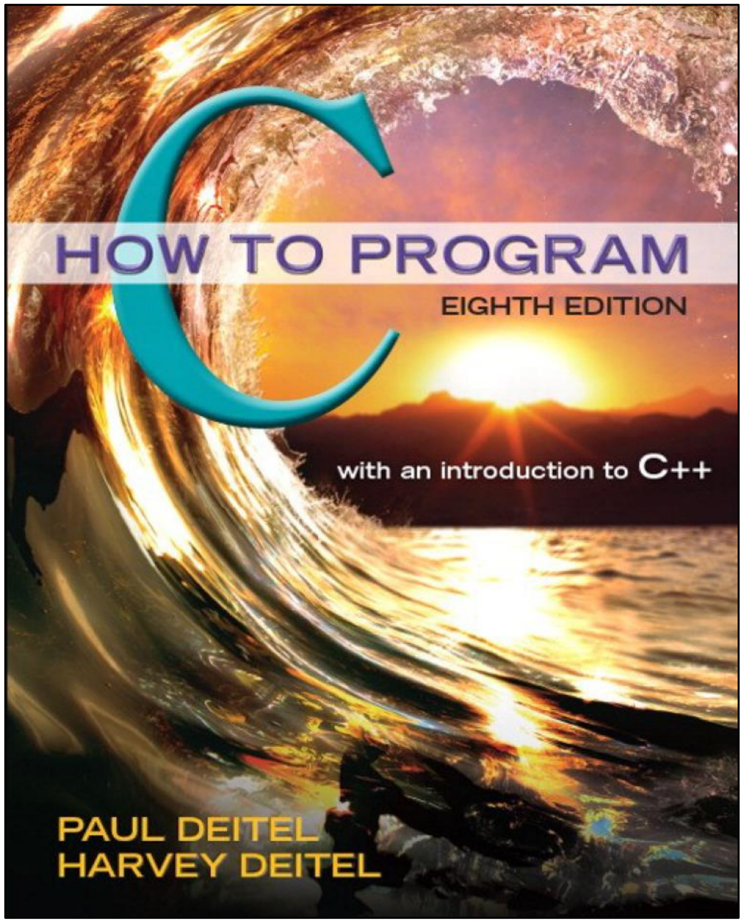
C How to Program

Eighth Edition



Chapter 2

Introduction to C Programming

Learning Objectives

- Write simple C programs.
- Use simple input and output statements.
- Use the fundamental data types.
- Learn computer memory concepts.
- Use arithmetic operators.
- Learn the precedence of arithmetic operators.
- Write simple decisionmaking statements.
- Begin focusing on secure C programming practices.

Outline

2.1 Introduction

2.2 A Simple C Program: Printing a Line of Text

2.3 Another Simple C Program: Adding Two Integers

2.4 Memory Concepts

2.5 Arithmetic in C

2.6 Decision Making: Equality and Relational Operators

2.7 Secure C Programming

2.1 Introduction

- The C language facilitates a structured and disciplined approach to computer program design.
- In this chapter we introduce C programming and present several examples that illustrate many important features of C.
- In Chapters 3 and 4 we present an introduction to structured programming in C.

2.1 The C Programming Language (1 of 3)

- C evolved from two previous languages, BCPL and B.
- BCPL was developed in 1967 by Martin Richards as a language for writing operating-systems software and compilers.
- Ken Thompson modeled many features in his B language after their counterparts in BCPL, and in 1970 he used B to create early versions of the UNIX operating system at Bell Laboratories.

2.1 The C Programming Language (2 of 3)

- The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented in 1972.
- C initially became widely known as the development language of the UNIX operating system.
- Many of todays leading operating systems are written in C and/or C++.
- C is mostly hardware independent.
- With careful design, it's possible to write C programs that are portable to most computers.

2.1 The C Programming Language (3 of 3)

Built for Performance

- C is widely used to develop systems that demand performance, such as operating systems, embedded systems, real-time systems and communications systems.

Some Popular Performance-Oriented C Applications (1 of 2)

Application	Description
Operating systems	C's portability and performance make it desirable for implementing operating systems, such as Linux and portions of Microsoft's Windows and Google's Android. Apple's OS X is built in Objective-C, which was derived from C. We discuss some key popular desktop/notebook operating systems and mobile operating systems in Section 1.11.
Embedded systems	The vast majority of the microprocessors produced each year are embedded in devices other than general-purpose computers. These **embedded systems** include navigation systems, smart home appliances, home security systems, smartphones, tablets, robots, intelligent traffic intersections and more. C is one of the most popular programming languages for developing embedded systems, which typically need to run as fast as possible and conserve memory. For example, a car's antilock brakes must respond immediately to slow or stop the car without skidding; game controllers used for video games should respond instantaneously to prevent any lag between the controller and the action in the game, and to ensure smooth animations.

Some Popular Performance-Oriented C Applications (2 of 2)

Application	Description
Real-time systems	Real-time systems are often used for “mission-critical” applications that require nearly instantaneous and predictable response times. Real-time systems need to work continuously for example, an air-traffic-control system must constantly monitor the positions and velocities of the planes and report that information to air-traffic controllers without delay so that they can alert the planes to change course if there’s a possibility of a collision.
Communications systems	Communications systems need to route massive amounts of data to their destinations quickly to ensure that things such as audio and video are delivered smoothly and without delay.

Portability Tip 1.1

Because C is a hardware-independent, widely available language, applications written in C often can run with little or no modification on a wide range of computer systems.

2.1 C Standard Library (1 of 3)

- As you'll learn later, C programs consist of pieces called **functions**.
- You can program all the functions you need to form a C program, but most C programmers take advantage of the rich collection of existing functions called the **C Standard Library**.
- Visit the following website for the C Standard Library documentation:
  http://www.gnu.org/software/libc/manual/html_mono/libc.html
- This textbook encourages a **building-block approach** to creating programs.

2.1 C Standard Library (2 of 3)

- Avoid reinventing the wheel.
- Instead, use existing pieces—this is called **software reuse**.
- When programming in C you'll typically use the following building blocks:
  - C Standard Library functions
  - Functions you create yourself
  - Functions other people (whom you trust) have created and made available to you

2.1 C Standard Library (3 of 3)

- The advantage of creating your own functions is that you'll know exactly how they work. You'll be able to examine the C code.
- The disadvantage is the time-consuming effort that goes into designing, developing and debugging new functions.

Performance Tip 2.0

Using C Standard Library functions instead of writing your own versions can improve program performance, because these functions are carefully written to perform efficiently.

Portability Tip 2.0+

Using C Standard Library functions instead of writing your own comparable versions can improve program portability, because these functions are used in virtually all Standard C implementations.

2.1 Typical C Program Development Environment (1 of 2)

- C systems generally consist of several parts: a program development environment, the language and the C Standard Library.
- C programs typically go through six phases to be executed
- These are: **edit**, **preprocess**, **compile**, **link**, **load** and **execute**.
- Although **C How to Program, 7/e** is a generic C textbook (written independently of the details of any particular operating system), we concentrate in this section on a typical Linux-based C system**.**

2.1 Typical C Program Development Environment (2 of 2)

- [Note: The programs in this book will run with little or no modification on most current C systems, including Microsoft Windows-based systems.] If you’re not using a Linux system, refer to the manuals for your system or ask your instructor how to accomplish these tasks in your environment.
- Check out our C Resource Center at `www.deitel.com/C` to locate “getting started” tutorials for popular C compilers and development environments.

2.1 Phase 1: Creating a Program

- Phase 1 consists of editing a file.
- This is accomplished with an **editor program**.
- Two editors widely used on Linux systems are `vi` and `emacs`.
- Software packages for the C/C++ integrated program development environments such as Eclipse and Microsoft Visual Studio have editors that are integrated into the programming environment.
- You type a C program with the editor, make corrections if necessary, then store the program on a secondary storage device such as a hard disk.
- C program file names should end with the `.c` extension.

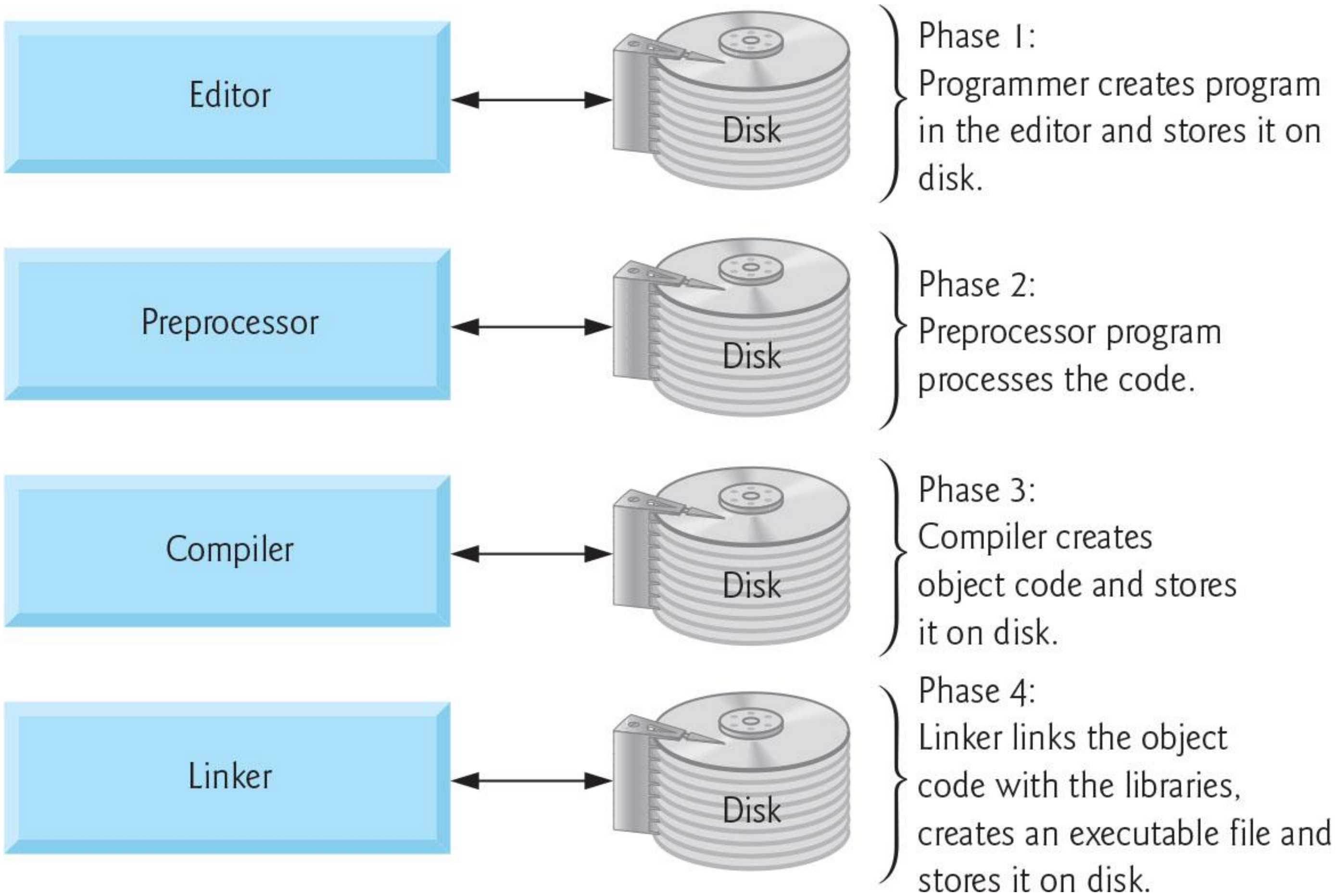
2.1 Phases 2 and 3: Preprocessing and Compiling a C Program (1 of 2)

- In Phase 2, the you give the command to **compile** the program.
- The compiler translates the C program into machine language-code (also referred to as **object code**).
- In a C system, a **preprocessor** program executes automatically before the compiler’s translation phase begins.
- The **C preprocessor** obeys special commands called **preprocessor directives**, which indicate that certain manipulations are to be performed on the program before compilation.

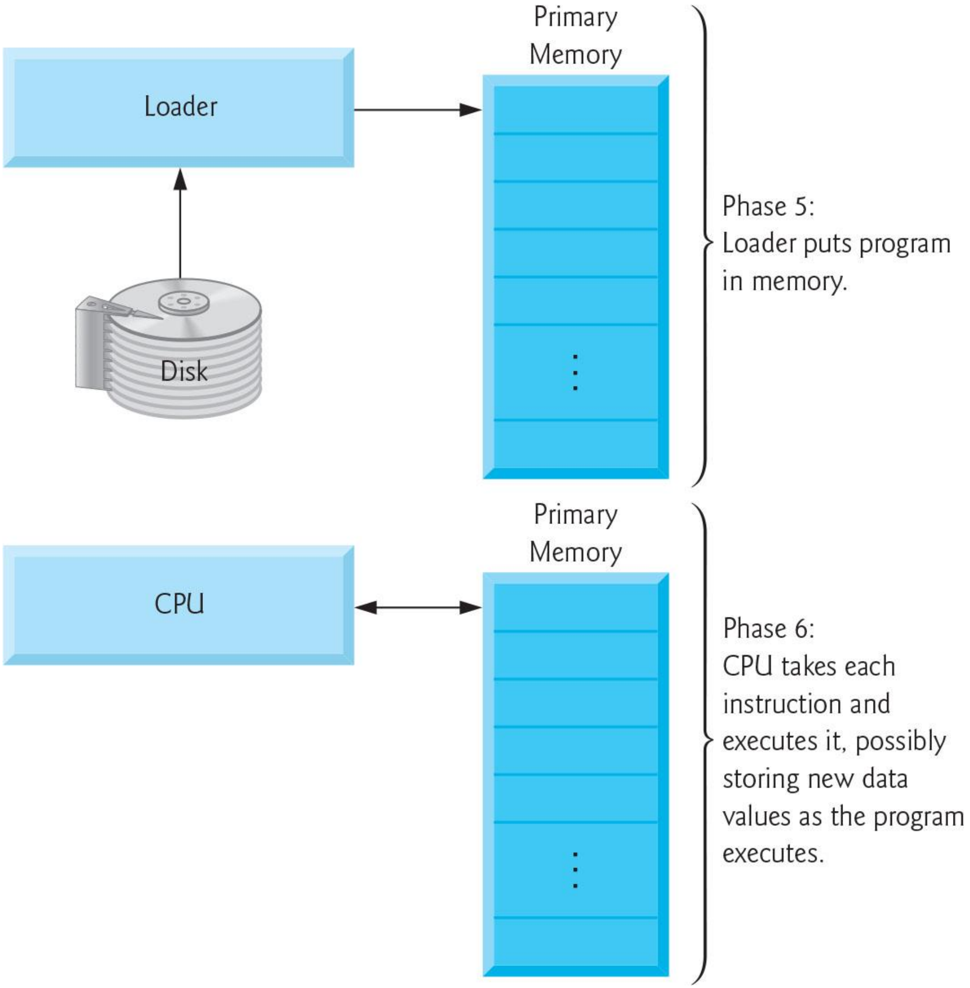
2.1 Phases 2 and 3: Preprocessing and Compiling a C Program (2 of 2)

- These manipulations usually consist of including other files in the file to be compiled and performing various text replacements.
- A detailed discussion of preprocessor features appears in Chapter 13.
- In Phase 3, the compiler translates the C program into machine-language code.
- A **syntax error** occurs when the compiler cannot recognize a statement because it violates the rules of the language.
- Syntax errors are also called **compile errors**, or **compile-time errors**.

Typical C Development Environment (1 of 2)



Typical C Development Environment (2 of 2)



2.1 Phase 4: Linking (1 of 2)

- The next phase is called **linking**.
- C programs typically contain references to functions defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project.
- The object code produced by the C compiler typically contains “holes” due to these missing parts.
- A **linker** links the object code with the code for the missing functions to produce an **executable image** (with no missing pieces).
- On a typical Linux system, the command to compile and link a program is called **`gcc`** (the GNU compiler).

2.1 Phase 4: Linking (2 of 2)

- To compile and link a program named `welcome.c` type
  - `gcc welcome.c`
- at the Linux prompt and press the **Enter** key (or **Return** key).
- [Note: Linux commands are case sensitive; make sure that each c is lowercase and that the letters in the filename are in the appropriate case.]
- If the program compiles and links correctly, a file called `a.out` is produced.
- This is the executable image of our `welcome.c` program.

2.1 Phase 5: Loading

- The next phase is called **loading**.
- Before a program can be executed, the program must first be placed in memory.
- This is done by the **loader**, which takes the executable image from disk and transfers it to memory.
- Additional components from shared libraries that support the program are also loaded.

2.1 Phase 6: Execution

- Finally, the computer, under the control of its CPU, **executes** the program one instruction at a time.
- To load and execute the program on a Linux system, type `./a.out` at the Linux prompt and press **Enter.**

2.2 A Simple C Program: Printing a Line of Text (1 of 17)

- We begin by considering a simple C program.
- Our first example prints a line of text (Figure 2.1).

```c
// Fig. 2.1: fig02_01.c
// A first program in C.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   printf( "Welcome to C!\n" );
} // end function main
```

```
Welcome to C!
```

2.2 A Simple C Program: Printing a Line of Text (2 of 17)

- ```
  // Fig. 2.1: fig02_01.c
  // A first program in C
  ```
  - begin with `//`, indicating that these two lines are **comments.**
  - Comments **document programs** and improve program readability.
  - Comments do not cause the computer to perform any action when the program is run.
  ```

2.2 A Simple C Program: Printing a Line of Text (3 of 17)

- Comments are ignored by the C compiler and do not cause any machine-language object code to be generated.
- Comments also help other people read and understand your program.

2.2 A Simple C Program: Printing a Line of Text (4 of 17)

- You can also use **/*…*/ multi-line comments** in which everything from /* on the first line to */ at the end of the line is a comment.
- We prefer `//` comments because they're shorter and they eliminate the common programming errors that occur with /*…*/ comments, especially when the closing `*/` is omitted.

`#include` Preprocessor Directive

- `#include <stdio.h>`
  – is a directive to the **C preprocessor**.

2.2 A Simple C Program: Printing a Line of Text (5 of 17)

- Lines beginning with # are processed by the preprocessor before compilation.
- Line 3 tells the preprocessor to include the contents of the **standard input/output header** (**<stdio.h>**) in the program.
- This header contains information used by the compiler when compiling calls to standard input/output library functions such as `printf`.

Blank Lines and White Space

- You use blank lines, space characters and tab characters (i.e., “tabs”) to make programs easier to read.
- Together, these characters are known as **white space.** White-space characters are normally ignored by the compiler.

2.2 A Simple C Program: Printing a Line of Text (6 of 17)

The `main` Function

- `int main( void )`
  - is a part of every C program.
  - The parentheses after main indicate that main is a program building block called a **function.**

2.2 A Simple C Program: Printing a Line of Text (7 of 17)

- C programs contain one or more functions, one of which must be `main`.
- Every program in C begins executing at the function `main`.
- The keyword `int` to the left of `main` indicates that `main` “returns” an integer (whole number) value.
- We’ll explain what it means for a function to “return a value” when we demonstrate how to create your own functions in Chapter 5.

2.2 A Simple C Program: Printing a Line of Text (8 of 17)

- For now, simply include the keyword `int` to the left of `main` in each of your programs.
- Functions also can receive information when they're called upon to execute.
- The `void` in parentheses here means that `main` does not receive any information.

Good Programming Practice 2.1

Every function should be preceded by a comment describing the function's purpose.

2.2 A Simple C Program: Printing a Line of Text (9 of 17)

- A left brace, {, begins the **body** of every function
- A corresponding **right brace** ends each function
- This pair of braces and the portion of the program between the braces is called a block.

An Output Statement

- `printf( "Welcome to C!\n" );`
  - instructs the computer to perform an **action,** namely to print on the screen the **string** of characters marked by the quotation marks.
  - A string is sometimes called a **character string,** a **message** or a **literal.**

2.2 A Simple C Program: Printing a Line of Text (10 of 17)

- The entire line, including the `printf` function (the “f” stands for “formatted”), its **argument** within the parentheses and the semicolon (;), is called a **statement.**
- Every statement must end with a semicolon (also known as the **statement terminator**).
- When the preceding `printf` statement is executed, it prints the message `Welcome to C (Hello World!)!` on the screen.
- The characters normally print exactly as they appear between the double quotes in the `printf` statement.

Escape Sequences

- Notice that the characters `\n` were not printed on the screen.
- The backslash (\) is called an **escape character.**
- It indicates that `printf` is supposed to do something out of the ordinary.

2.2 A Simple C Program: Printing a Line of Text (11 of 17)

- When encountering a backslash in a string, the compiler looks ahead at the next character and combines it with the backslash to form an **escape sequence.**
- The escape sequence **\n** means **newline.**
- When a newline appears in the string output by a `printf`, the newline causes the cursor to position to the beginning of the next line on the screen.
- Some common escape sequences are listed in Figure 2.2.

Figure 2.2 Some Common Escape Sequences

Escape sequence	Description
`\n`	Newline. Position the cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the cursor to the next tab stop.
`\a`	Alert. Produces a sound or visible alert without changing the current cursor position.
`\\`	Backslash. Insert a backslash character in a string.
`\”`	Double quote. Insert a double-quote character in a string.

2.2 A Simple C Program: Printing a Line of Text (12 of 17)

- Because the backslash has special meaning in a string, i.e., the compiler recognizes it as an escape character, we use a double backslash (`\\`) to place a single backslash in a string.
- Printing a double quote also presents a problem because double quotes mark the boundaries of a string—such quotes are not printed.
- By using the escape sequence `\"` in a string to be output by `printf`, we indicate that `printf` should display a double quote.
- The right brace, `}`, indicates that the end of `main` has been reached.

Good Programming Practice 2.2

Add a comment to the line containing the right brace, `}`, that closes every function, including `main`.

2.2 A Simple C Program: Printing a Line of Text (13 of 17)

- We said that `printf` causes the computer to perform an action.
- As any program executes, it performs a variety of actions and makes **decisions.**
- Chapter 3 discusses this **action/decision model** of programming in depth.

Common Programming Error 2.1

Mistyping the name of the output function printf as print in a program.

2.2 A Simple C Program: Printing a Line of Text (14 of 17)

The Linker and Executables

- Standard library functions like `printf` and `scanf` are not part of the C programming language.
- For example, the compiler cannot find a spelling error in `printf` or `scanf`.
- When the compiler compiles a `printf` statement, it merely provides space in the object program for a "call" to the library function.
- But the compiler does not know where the library functions are—the linker does.
- When the linker runs, it locates the library functions and inserts the proper calls to these library functions in the object program.

2.2 A Simple C Program: Printing a Line of Text (15 of 17)

- Now the object program is complete and ready to be executed.
- For this reason, the linked program is called an **executable.**
- If the function name is misspelled, it's the linker that will spot the error, because it will not be able to match the name in the C program with the name of any known function in the libraries.

Good Programming Practice 2.3

Indent the entire body of each function one level of indentation (we recommend three spaces) within the braces that define the body of the function. This indentation emphasizes the functional structure of programs and helps make them easier to read.

Good Programming Practice 2.4

Set a convention for the indent size you prefer and then uniformly apply that convention. The tab key may be used to create indents, but tab stops can vary. Professional style guides often recommend using spaces rather than tabs.

2.2 A Simple C Program: Printing a Line of Text (16 of 17)

Using Multiple `printf`s

- The `printf` function can print `Welcome to C!` several different ways.
- For example, the program of Figure 2.3 produces the same output as the program of Figure 2.1 (see slide 6).
- This works because each `printf` resumes printing where the previous `printf` stopped printing.
- The first `printf` prints `Welcome` followed by a space and the second `printf` begins printing on the same line immediately following the space.

Figure 2.3 Printing One Line with Two `Printf` Statements

```c
// Fig. 2.3: fig02_03.c
// Printing on one line with two printf statements.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   printf( "Welcome " );
   printf( "to C!\n" );
} // end function main
```

```
Welcome to C!
```

2.2 A Simple C Program: Printing a Line of Text (17 of 17)

- One `printf` can print **several** lines by using additional newline characters as in Figure 2.4.
- Each time the `\n` (newline) escape sequence is encountered, output continues at the beginning of the next line.

Figure 2.4 Printing Multiple Lines with a Single `Printf`

```
// Fig. 2.4: fig02_04.c
// Printing multiple lines with a single printf.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   printf( "Welcome\nto\nC!\n" );
} // end function main
```

```
Welcome
to
C!
```

2.3 Another Simple C Program: Adding Two Integers (1 of 13)

- Our next program (Figure 2.5) uses the Standard Library function `scanf` to obtain two integers typed by a user at the keyboard, computes the sum of these values and prints the result using `printf.`
- [In the input/output dialog of Figure 2.5, we emphasize the numbers entered by the user in **bold.**]

Figure 2.5 Addition Program (1 of 2)

```c
// Fig. 2.5: fig02_05.c
// Addition program.
#include <stdio.h>

// function main begins program execution
int main( void )
{
    int integer1; // first number to be entered by user
    int integer2; // second number to be entered by user

    printf( "Enter first integer\n" ); // prompt
    scanf( "%d", &integer1 ); // read an integer

    printf( "Enter second integer\n" ); // prompt
    scanf( "%d", &integer2 ); // read an integer

    int sum; // variable in which sum will be stored
    sum = integer1 + integer2; // assign total to sum

    printf( "Sum is %d\n", sum ); // print sum
} // end function main
```

Figure 2.5 Addition Program (2 of 2)

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

2.3 Another Simple C Program: Adding Two Integers (2 of 13)

Variables and Variable Definitions

- ```c
  int integer1; // first number to be entered by user
  int integer2; // second number to be entered by user
  int sum; // variable in which sum will be stored
  ```
  are **definitions**.
- The names `integer1`, `integer2` and `sum` are the names of **variables**—locations in memory where values can be stored for use by a program.
- These definitions specify that the variables `integer1`, `integer2` and `sum` are of type `int`, which means that they'll hold **integer** values, i.e., whole numbers such as 7, –11, 0, 31914 and the like.
  ```

2.3 Another Simple C Program: Adding Two Integers (3 of 13)

- All variables must be defined with a name and a data type before they can be used in a program.
- The preceding definitions could have been combined into a single definition statement as follows:
  - `int integer1, integer2, sum;`

  but that would have made it difficult to describe the variables with corresponding comments

2.3 Another Simple C Program: Adding Two Integers (4 of 13)

Identifiers and Case Sensitivity

- A variable name in C is any valid **identifier.**
- An identifier is a series of characters consisting of letters, digits and underscores (_) that does **not** begin with a digit.
- C is **case sensitive**—uppercase and lowercase letters are different in C, so `a1` and `A1` are different identifiers.

Common Programming Error 2.2

Using a capital letter where a lowercase letter should be used (for example, typing Main instead of `main`).

Error-Prevention Tip 2.1

Avoid starting identifiers with the underscore character (_) to prevent conflicts with compiler-generated identifiers and standard library identifiers.

Good Programming Practice 2.5

Choosing meaningful variable names helps make a program self-documenting—that is, fewer comments are needed.

Good Programming Practice 2.6

The first letter of an identifier used as a simple variable name should be a lowercase letter. Later in the text we'll assign special significance to identifiers that begin with a capital letter and to identifiers that use all capital letters.

Good Programming Practice 2.7

Multiple-word variable names can help make a program more readable. Separate the words with underscores as in `total_commissions`, or, if you run the words together, begin each word after the first with a capital letter as in `totalCommissions`. The latter style—often called camel casing because the pattern of uppercase and lowercase letters resembles the silhouette of a camel—is preferred.

2.3 Another Simple C Program: Adding Two Integers (5 of 13)

Prompting Messages

- `printf( "Enter first integer\n" ); // prompt`
  - displays the literal “`Enter first integer`” and positions the cursor to the beginning of the next line.
  - This message is called a **prompt** because it tells the user to take a specific action.

2.3 Another Simple C Program: Adding Two Integers (6 of 13)

The scanf Function and Formatted Inputs

- The next statement
  - `scanf( "%d", &integer1 ); // read an integer`

  uses **`scanf`** to obtain a value from the user.
- The `scanf` function reads from the standard input, which is usually the keyboard.

2.3 Another Simple C Program: Adding Two Integers (7 of 13)

- This `scanf` has two arguments, `"%d"` and `&integer1`.
- The first, the **format control string,** indicates the type of data that should be input by the user.
- The **%d conversion specifier** indicates that the data should be an integer (the letter `d` stands for "decimal integer").
- The `%` in this context is treated by `scanf` (and `printf` as we'll see) as a special character that begins a conversion specifier.
- The second argument of `scanf` begins with an ampersand (&)—called the **address operator** in C — followed by the variable name.

2.3 Another Simple C Program: Adding Two Integers (8 of 13)

- The &, when combined with the variable name, tells `scanf` the location (or address) in memory at which the variable `integer1` is stored.
- The computer then stores the value that the user enters for `integer1` at that location.
- The use of ampersand `(&)` is often confusing to novice programmers or to people who have programmed in other languages that do not require this notation.
- For now, just remember to precede each variable in every call to `scanf` with an ampersand.

Good Programming Practice 2.8

Place a space after each comma (,) to make programs more readable.

2.3 Another Simple C Program: Adding Two Integers (9 of 13)

- When the computer executes the preceding `scanf`, it waits for the user to enter a value for variable `integer1`.
- The user responds by typing an integer, then pressing the **Enter key** to send the number to the computer.
- The computer then assigns this number, or value, to the variable `integer1`.
- Any subsequent references to `integer1` in this program will use this same value.
- Functions `printf` and `scanf` facilitate interaction between the user and the computer.
- Because this interaction resembles a dialogue, it's often called **interactive computing.**

2.3 Another Simple C Program: Adding Two Integers (10 of 13)

- `printf( "Enter second integer\n" ); // prompt`
  - displays the message `Enter second integer` on the screen, then positions the cursor to the beginning of the next line.
  - This `printf` also prompts the user to take action.
- `scanf( "%d", &integer2 ); // read an integer`
  - obtains a value for variable `integer2` from the user.

2.3 Another Simple C Program: Adding Two Integers (11 of 13)

Assignment Statement

- The **assignment statement**
  - `sum = integer1 + integer2; // assign total to sum`

  calculates the total of variables `integer1` and `integer2` and assigns the result to variable `sum` using the assignment operator `=`.
- The statement is read as, "`sum` **gets** the value of `integer1 + integer2`." Most calculations are performed in assignments.
- The `=` operator and the `+` operator are called binary operators because each has two **operands**.
- The `+` operator's two operands are `integer1` and `integer2`.
- The `=` operator's two operands are `sum` and the value of the expression `integer1 + integer2`.

Good Programming Practice 2.9

Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.

Common Programming Error 2.3

A calculation in an assignment statement must be on the right side of the = operator. It’s a compilation error to place a calculation on the left side of an assignment operator.

2.3 Another Simple C Program: Adding Two Integers (12 of 13)

Printing with a Format Control String

- `printf( "Sum is %d\n", sum ); // print sum`
  - calls function `printf` to print the literal `Sum` is followed by the numerical value of variable `sum` on the screen.
  - This `printf` has two arguments, `"Sum is %d\n"` and sum.
  - The first argument is the format control string.
  - It contains some literal characters to be displayed, and it contains the conversion specifier `%d` indicating that an integer will be printed.
  - The second argument specifies the value to be printed.
  - Notice that the conversion specifier for an integer is the same in both `printf` and `scanf`.

2.3 Another Simple C Program: Adding Two Integers (13 of 13)

Calculations in `printf` Statements

- We could have combined the previous two statements into the statement
  - `printf( "Sum is %d\n", integer1 + integer2 );`
- The right brace, `}`, at line 21 indicates that the end of function `main` has been reached.

Common Programming Error 2.4

Forgetting to precede a variable in a scanf statement with an ampersand (&) when that variable should, in fact, be preceded by an ampersand results in an execution-time error. On many systems, this causes a "segmentation fault" or "access violation." Such an error occurs when a user's program attempts to access a part of the computer's memory to which it does not have access privileges. The precise cause of this error will be explained in Chapter 7.

Common Programming Error 2.5

Preceding a variable included in a `printf` statement with an ampersand when, in fact, that variable should not be preceded by an ampersand.

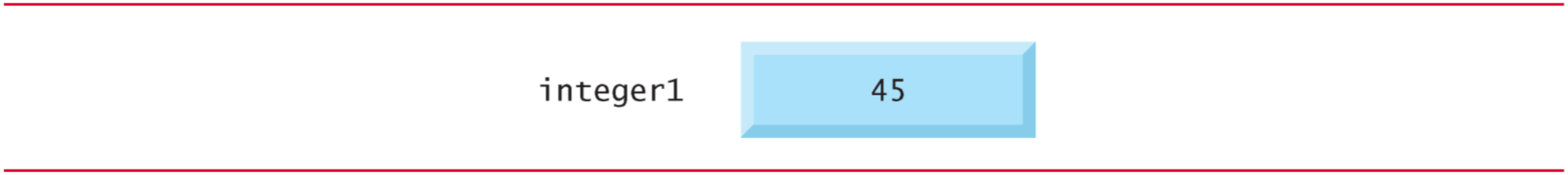
2.4 Memory Concepts (1 of 5)

- Variable names such as `integer1, integer2` and `sum` actually correspond to locations in the computer's memory.
- Every variable has a name, a **type** and a **value.**
- In the addition program of Figure 2.5, when the statement
  –

```
scanf( "%d", &integer1 ); // read an integer
```

- is executed, the value entered by the user is placed into a memory location to which the name `integer1` has been assigned.
- Suppose the user enters the number `45` as the value for `integer1`.
- The computer will place `45` into location `integer1` as shown in Figure 2.6.

Figure 2.6 Memory Location Showing the Name and Value of a Variable

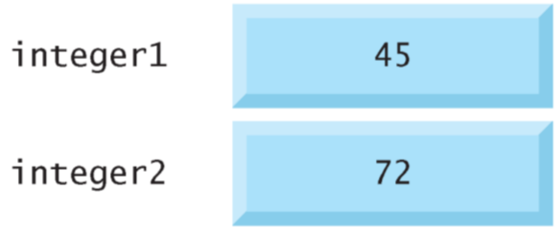


2.4 Memory Concepts (2 of 5)

- Whenever a value is placed in a memory location, the value replaces the previous value in that location; thus, this process is said to be **destructive.**
- When the statement
  - `scanf( "%d", &integer2 ); // read an integer`

  executes, suppose the user enters the value `72`.
- This value is placed into location `integer2`, and memory appears as in Figure 2.7.
- These locations are not necessarily adjacent in memory.

Figure 2.7 Memory Locations After Both Variables Are Input



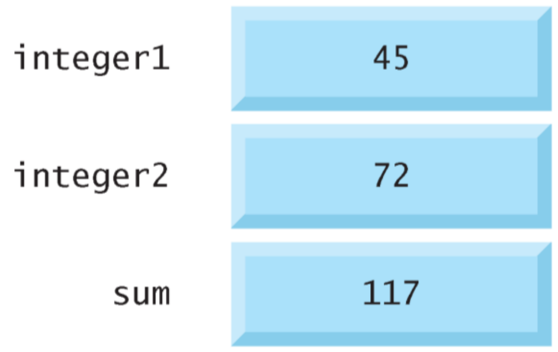
2.4 Memory Concepts (3 of 5)

- Once the program has obtained values for `integer1` and `integer2`, it adds these values and places the total into variable `sum`.
- `sum = integer1 + integer2; // assign total to sum`
  - replaces whatever value was stored in `sum`.

2.4 Memory Concepts (4 of 5)

- This occurs when the calculated total of `integer1` and `integer2` is placed into location `sum` (destroying the value already in `sum`).
- After `sum` is calculated, memory appears as in Figure 2.8.
- The values of `integer1` and `integer2` appear exactly as they did before they were used in the calculation.

Figure 2.8 Memory Locations After a Calculation



2.4 Memory Concepts (5 of 5)

- They were used, but not destroyed, as the computer performed the calculation.
- Thus, when a value is read from a memory location, the process is said to be **nondestructive.**

2.5 Arithmetic in C (1 of 9)

- Most C programs perform calculations using the C **arithmetic operators** (Figure 2.9).
- The **asterisk** (*****) indicates multiplication and the **percent sign** (**%**) denotes the remainder operator, which is introduced below.
- In algebra, to multiply a times b, we simply place these single-letter variable names side by side as in ab.
- In C, however, if we were to do this, `ab` would be interpreted as a single, two-letter name (or identifier).
- Therefore, C requires that multiplication be explicitly denoted by using the `*` operator as in `a * b`.
- The arithmetic operators are all binary operators.
- For example, the expression `3 + 7` contains the binary operator `+` and the operands `3` and `7`.

Figure 2.9 Arithmetic Operators

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	$f + 7$	f + 7
Subtraction	–	$p - c$	p - c
Multiplication	*	bm	b * m
Division	/	x/y or $\frac{x}{y}$ or $x \div y$	x / y
Remainder	%	$r \bmod s$	r % s

2.5 Arithmetic in C (2 of 9)

Integer Division and the Remainder Operator

- **Integer division** yields an integer result
- For example, the expression $\frac{7}{4}$ evaluates to 1 and the expression $\frac{17}{5}$ evaluates to 3
- C provides the **remainder operator, %,** which yields the remainder after integer division
- Can be used only with integer operands
- The expression x % y yields the remainder after x is divided by y
- Thus, 7 % 4 yields 3 and 17 % 5 yields 2

Common Programming Error 2.6

An attempt to divide by zero is normally undefined on computer systems and generally results in a fatal error that causes the program to terminate immediately without having successfully performed its job. **Nonfatal errors** allow programs to run to completion, often producing incorrect results.

2.5 Arithmetic in C (3 of 9)

Arithmetic Expressions in Straight-Line Form

- Arithmetic expressions in C must be written in **straight-line form** to facilitate entering programs into the computer.
- Thus, expressions such as "**a** divided by **b**" must be written as `a/b` so that all operators and operands appear in a straight line.
- The algebraic notation $\frac{a}{b}$ is generally not acceptable to compilers, although some special-purpose software packages do support more natural notation for complex mathematical expressions.

2.5 Arithmetic in C (4 of 9)

Parentheses for Grouping Subexpressions

- Parentheses are used in C expressions in the same manner as in algebraic expressions.
- For example, to multiply a times the quantity b + c we write a * (b + c).

2.5 Arithmetic in C (5 of 9)

Rules of Operator Precedence

- C applies the operators in arithmetic expressions in a precise sequence determined by the following **rules of operator precedence,** which are generally the same as those in algebra:
  - Operators in expressions contained within pairs of parentheses are evaluated first. Parentheses are said to be at the "highest level of precedence." In cases of **nested,** or **embedded, parentheses,** such as
    - ((a + b) + c)

the operators in the innermost pair of parentheses are applied first.

2.5 Arithmetic in C (6 of 9)

- Multiplication, division and remainder operations are applied next. If an expression contains several multiplication, division and remainder operations, evaluation proceeds from left to right. Multiplication, division and remainder are said to be on the same level of precedence.
- Addition and subtraction operations are evaluated next. If an expression contains several addition and subtraction operations, evaluation proceeds from left to right. Addition and subtraction also have the same level of precedence, which is lower than the precedence of the multiplication, division and remainder operations.
- The assignment operator (=) is evaluated last.

2.5 Arithmetic in C (7 of 9)

- The rules of operator precedence specify the order C uses to evaluate expressions. When we say evaluation proceeds from left to right, we’re referring to the **associativity** of the operators.
- We’ll see that some operators associate from right to left.
- Figure 2.10 summarizes these rules of operator precedence for the operators we’ve seen so far.

Figure 2.10 Precedence of Arithmetic Operators

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the **innermost** pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they’re evaluated left to right.
* / %	Multiplication Division Remainder	Evaluated second. If there are several, they’re evaluated left to right.
+ −	Addition Subtraction	Evaluated third. If there are several, they’re evaluated left to right.
=	Assignment	Evaluated last.

2.5 Arithmetic in C (8 of 9)

- Figure 2.11 illustrates the order in which the operators are applied.

Figure 2.11 Order in Which a Second-Degree Polynomial is Evaluated

Step 1.	`y = 2 * 5 * 5 + 3 * 5 + 7;`	*(Leftmost multiplication)*
	`2 * 5 is` $\boxed{10}$	
Step 2.	`y = 10 * 5 + 3 * 5 + 7;`	*(Leftmost multiplication)*
	`10 * 5 is` $\boxed{50}$	
Step 3.	`y = 50 + 3 * 5 + 7;`	*(Multiplication before addition)*
	`3 * 5 is` $\boxed{15}$	
Step 4.	`y = 50 + 15 + 7;`	*(Leftmost addition)*
	`50 + 15 is` $\boxed{65}$	
Step 5.	`y = 65 + 7;`	*(Last addition)*
	`65 + 7 is` $\boxed{72}$	
Step 6.	`y = 72`	*(Last operation—place `72` in `y`)*

2.5 Arithmetic in C (9 of 9)

- As in algebra, it’s acceptable to place unnecessary parentheses in an expression to make the expression clearer.
- These are called **redundant parentheses.**

2.6 Decision Making: Equality and Relational Operators (1 of 9)

- Executable C statements either perform actions (such as calculations or input or output of data) or make **decisions** (we'll soon see several examples of these).
- We might make a decision in a program, for example, to determine whether a person's grade on an exam is greater than or equal to 60 and whether the program should print the message "Congratulations! You passed."
- This section introduces a simple version of C's **if statement** that allows a program to make a decision based on the truth or falsity of a statement of fact called a **condition.**

2.6 Decision Making: Equality and Relational Operators (2 of 9)

- If the condition is **true** (i.e., the condition is met) the statement in the body of the `if` statement is executed.
- If the condition is **false** (i.e., the condition isn’t met) the body statement is not executed.
- Whether the body statement is executed or not, after the `if` statement completes, execution proceeds with the next statement after the `if` statement.
- Conditions in `if` statements are formed by using the **equality operators** and **relational operators** summarized in Figure 2.12.

2.6 Decision Making: Equality and Relational Operators (3 of 9)

- The relational operators all have the same level of precedence and they associate left to right.
- The equality operators have a lower level of precedence than the relational operators and they also associate left to right.
- In C, a condition may actually be **any expression that generates a zero (false) or nonzero (true) value**.

Figure 2.12 Equality and Relational Operators (1 of 2)

Relational operators

Algebraic equality or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Figure 2.12 Equality and Relational Operators (2 of 2)

Equality operators

Algebraic equality or relational operator	**C equality or relational operator**	**Example of C condition**	**Meaning of C condition**
=	==	x==y	x is equal to y
≠	!=	x!=y	x is not equal to y

Common Programming Error 2.7

A syntax error occurs if the two symbols in any of the operators $==$, $!=$, $>=$ and $<=$ are separated by spaces.

Common Programming Error 2.8

Confusing the equality operator == with the assignment operator. To avoid this confusion, the equality operator should be read “double equals” and the assignment operator should be read “gets” or “is assigned the value of.” As you’ll see, confusing these operators may not cause an easy-to-recognize compilation error, but may cause extremely subtle logic errors.

2.6 Decision Making: Equality and Relational Operators (4 of 9)

- Figure 2.13 uses six `if` statements to compare two numbers entered by the user.
- If the condition in any of these `if` statements is true, the `printf` statement associated with that `if` executes.

Figure 2.13 Using `If` Statements, Relational Operators, and Equality Operators (1 of 3)

```c
// Fig. 2.13: fig02_13.c
// Using if statements, relational
// operators, and equality operators.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   printf( "Enter two integers, and I will tell you\n" );
   printf( "the relationships they satisfy: " );

   int num1; // first number to be read from user
   int num2; // second number to be read from user

   scanf( "%d %d", &num1, &num2 ); // read two integers

   if ( num1 == num2 ) {
      printf( "%d is equal to %d\n", num1, num2 );
   } // end if

```

Figure 2.13 Using If Statements, Relational Operators, and Equality Operators (2 of 3)

```
    if ( num1 != num2 ) {
        printf( "%d is not equal to %d\n", num1, num2 );
    } // end if

    if ( num1 < num2 ) {
        printf( "%d is less than %d\n", num1, num2 );
    } // end if

    if ( num1 > num2 ) {
        printf( "%d is greater than %d\n", num1, num2 );
    } // end if

    if ( num1 <= num2 ) {
        printf( "%d is less than or equal to %d\n", num1, num2 );
    } // end if

    if ( num1 >= num2 ) {
        printf( "%d is greater than or equal to %d\n", num1, num2 );
    } // end if
} // end function main
```

Figure 2.13 Using If Statements, Relational Operators, and Equality Operators (3 of 3)

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

2.6 Decision Making: Equality and Relational Operators (5 of 9)

- The program uses `scanf` to input two numbers.
- Each conversion specifier has a corresponding argument in which a value will be stored.
- The first `%d` converts a value to be stored in the variable `num1`, and the second `%d` converts a value to be stored in variable `num2`.

Good Programming Practice 2.10

Although it’s allowed, there should be no more than one statement per line in a program.

Common Programming Error 2.9

Placing commas (when none are needed) between conversion specifiers in the format control string of a `scanf` statement.

2.6 Decision Making: Equality and Relational Operators (6 of 9)

Comparing Numbers

- The if statement

```
if ( num1 == num2 ) {
   printf( "%d is equal to %d\n", num1, num2 );
}
```

compares the values of variables `num1` and `num2` to test for equality.

- If the conditions are `true` in one or more of the `if` statements, the corresponding body statement displays an appropriate line of text.
- Indenting the body of each `if` statement and placing blank lines above and below each `if` statement enhances program readability.

Common Programming Error 2.10

Placing a semicolon immediately to the right of the right parenthesis after the condition in an `if` statement.

2.6 Decision Making: Equality and Relational Operators (7 of 9)

- A left brace, `{`, begins the body of each `if` statement
- A corresponding right brace, `}`, ends each `if` statement's body
- Any number of statements can be placed in the body of an `if` statement.

Good Programming Practice 2.11

A lengthy statement may be spread over several lines. If a statement must be split across lines, choose breaking points that make sense (such as after a comma in a comma-separated list). If a statement is split across two or more lines, indent all subsequent lines. It’s not correct to split identifiers.

2.6 Decision Making: Equality and Relational Operators (8 of 9)

- Figure 2.14 (see slide 118) lists from highest to lowest the precedence of the operators introduced in this chapter.
- Operators are shown top to bottom in decreasing order of precedence.
- The equals sign is also an operator.
- All these operators, with the exception of the assignment operator =, associate from left to right.
- The assignment operator (=) associates from right to left.

Good Programming Practice 2.12

Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are applied in the proper order. If you’re uncertain about the order of evaluation in a complex expression, use parentheses to group expressions or break the statement into several simpler statements. Be sure to observe that some of C’s operators such as the assignment operator (=) associate from right to left rather than from left to right.

Figure 2.14 Precedence and Associativity of the Operators Discussed So Far

Operators	Associativity
()	left to right
* / %	left to right
+ −	left to right
< <= > >=	left to right
== !=	left to right
=	right to left

2.6 Decision Making: Equality and Relational Operators (9 of 9)

- Some of the words we've used in the C programs in this chapter—in particular `int` and `if`—are **keywords** or reserved words of the language.
- Figure 2.15 contains the C keywords.
- These words have special meaning to the C compiler, so you must be careful not to use these as identifiers such as variable names.

Figure 2.15 C’s Keywords

Keywords

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
Char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Keywords added in C99 standard

`_Bool   _Complex   _Imaginary   inline   restrict`

Keywords added in C11 standard

`_Alignas   _Alignof   _Atomic   _Generic   _Noreturn   _Static_assert   _Thread_local`

Copyright

This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.