Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

# CHAPTER 1

# Introduction to Computers

## Learning Objectives

**When you complete this chapter, you will be able to:**

- **1.1** Describe basic computer system concepts
- **1.2** Identify the different computing environments and their components
- **1.3** List and describe the classifications of computer languages
- **1.4** Identify the steps in the development of a computer program
- **1.5** Describe the system development life cycle (SDLC)

## 1.1 Computer Systems

Welcome to computer science! You are about to explore a wonderful and exciting world—a world that offers many challenging and exciting careers. This chapter introduces basic computer science concepts, with a focus on how these concepts pertain to computer programming. You will learn about the hardware and software components of a computer system. You will also learn about the evolution of computer programming languages and how the C language fits into the picture. Finally, you will be introduced to the tools and steps involved in writing a computer program, and review the system development methodology.

Computer systems are found everywhere. They are an essential part of daily life, facilitating interactions with family and friends, governments, and small and large businesses. But what is a computer? A computer is a system made of two major components: hardware and software. The computer hardware is the physical equipment. The software is the collection of programs (instructions) that allow the hardware to do a specific job. **Figure 1-1** represents a computer system.

### Computer Hardware

The hardware component of a computer system consists of five parts: input devices, central processing unit (CPU), primary storage, output devices, and secondary storage devices such as internal and external hard drives, USB (Universal Serial Bus) devices, and external backup tapes (**Figure 1-2**).



Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 1-1** A computer system

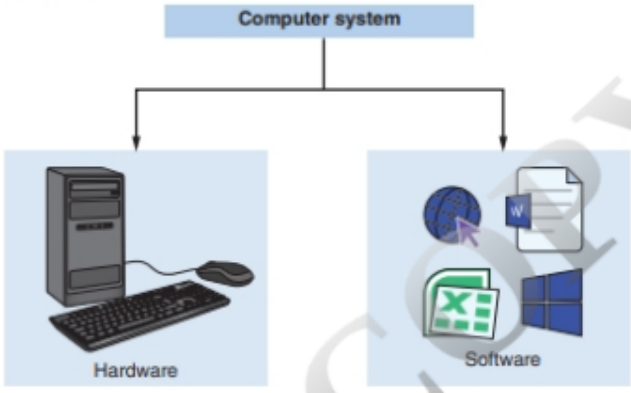
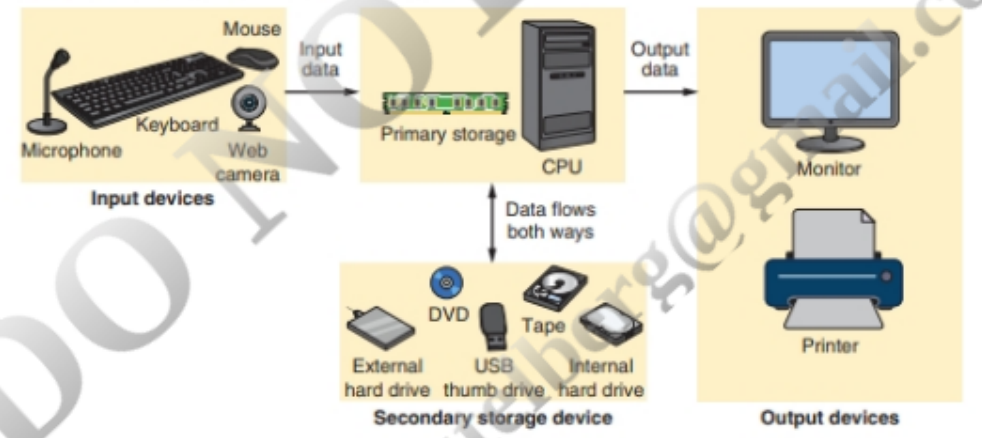



**Figure 1-2** Basic hardware components




The most common input device is a keyboard, where programs and data are entered into the computer. Examples of other input devices include a mouse, a pen or stylus, a touch screen, a camera, and a microphone or other audio input unit. The term input refers to data flowing into the computer system.

The central processing unit (CPU) is responsible for executing instructions such as arithmetic calculations, comparisons among data, and movement of data inside the system. Today's computers may have one, two, or more CPUs. Primary storage, also known as main memory, is composed of Read Only Memory (ROM) and Random Access Memory (RAM). Programs and data are stored temporarily in main memory during processing. The data in primary storage are erased when the personal computer is turned off or when a user is logged off from a time-sharing computer.

The output device, usually a monitor or a printer, allows the user to see the results, or output, of a computer program. If the output is shown on the monitor, it is referred to as a soft copy. If it is printed on the printer, it is referred to as a hard copy.

Secondary storage, is used for both input and output. It is the place where the programs and data are stored permanently. In a PC, a hard drive is typically used as secondary storage. When the computer is turned off, programs and data remain in the secondary storage, ready for the next use.

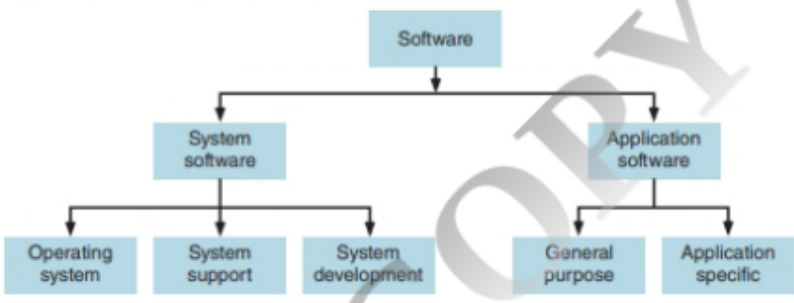
Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Computer Software

Computer software is divided into two broad categories: system software and application software. System software manages the computer resources. Application software, on the other hand, is directly responsible for helping users solve their problems. **Figure 1-3** shows this breakdown of computer software.

**Figure 1-3** Types of software




## System Software

System software consists of programs that manage a computer's hardware resources and perform information-processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The operating system provides a user interface (that is the screen the user interacts with, including dialog boxes, buttons to click, and textboxes in which to enter information), as well as access to files and databases. The operating system also interacts with communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consist of programs that provide performance diagnosis and statistics.

The last system software category, development software, includes the language translators that convert programs into machine language for execution, debugging tools to ensure that the programs are error-free, and computer-assisted software engineering (CASE) systems that are beyond the scope of this book.

## Application Software

We divide application software into two classes: general-purpose software and application-specific software. The first, general-purpose software is purchased from a software developer and can be used for more than one application. Examples of general-purpose software include word processors, database management systems, and computer-aided design systems. They are labeled general purpose because they can solve a variety of user computing problems.

The second type, application-specific software, can be used only for its intended purpose. A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed; they cannot be used for other generalized tasks.

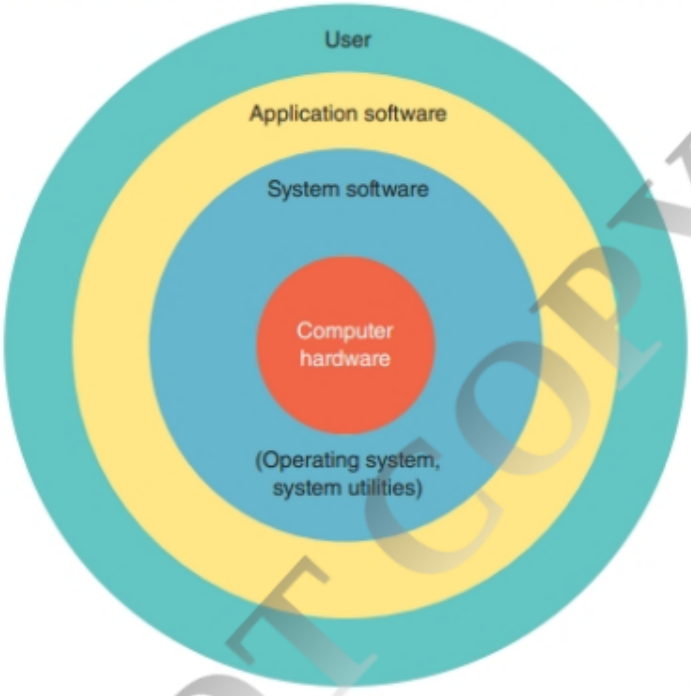
The relationship between system and application software is shown in **Figure 1-4**. In this figure, each circle represents an interface point. The inner core is the hardware. The user is represented by the outer layer. To work with the system, the typical user uses some form of application software. The application software in turn interacts with the operating system, which is a part of the system software layer. The system software provides the direct interaction with the hardware.

If users cannot buy software that supports their needs, then a custom-developed application must be built. In today's computing environment, one of the tools used to develop software is the C language that you will be studying in this text.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 1-4** Relationship between system and application software




## 1.2 Computing Environments

In the early days of computers, there was only one environment: the mainframe computer hidden in a central computing department. With the advent of minicomputers and personal computers, the environment changed, resulting in computers on virtually every desktop. In this section we describe several different environments.

### Personal Computing Environment

In 1971, Marcian E. Hoff, working for Intel, combined the basic elements of the central processing unit into the microprocessor. This first computer on a chip was the Intel 4004 and was the grandparent many times removed of the chips now used in computers around the world. The rapid development of computer chips ultimately led to the transition from large main frame computers to a smaller, self-contained device known as a personal computer in the 1970s. The market soon diverged into two main types of personal computers. The first type ran the Microsoft DOS operating system and then, later, Microsoft Windows, and were manufactured by many different companies. The term PC (short for "personal computer") is now used to refer primarily to computers that run Microsoft operating systems. The second type were computers manufactured exclusively by Apple. The early versions of these computers were designed to sit on a desktop. Now, personal computing devices take several forms, including smart phones, tablets, laptops, desktops setups with monitors and towers, and all-in-one desktop computers, which combine the CPU and the monitor into one device. **Figure 1-5** shows an array of modern personal computing devices.

### Time-Sharing Environment

Although, it is not as common as it used to be, some organizations still employ what is known as a time-sharing environment. In the time-sharing environment, many users are connected to one or more computers. These computers may be minicomputers (nowadays known as servers) or central mainframes. The terminals they use are often nonprogrammable, although today we see more and more microcomputers being used to simulate terminals. Also, in the time-sharing environment, the output devices (such as printers) and secondary storage devices (such as disks) are shared by all of the users. A typical college lab in which a minicomputer is shared by many students is shown in **Figure 1-6**.

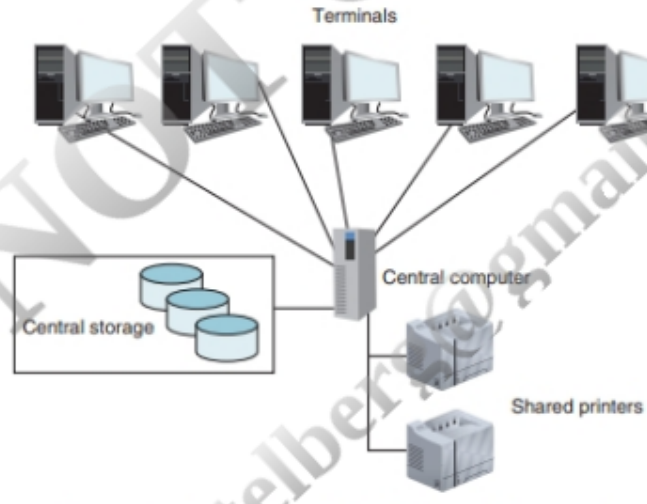
Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 1-5** Modern personal computing devices


**Figure 1-6** Time-sharing environment




In a time-sharing environment, all computing must be done by the central computer. In other words, the central computer has many duties: It must control the shared resources; it must manage the shared data and printing; and it must do the computing. All of this work tends to keep the computer busy. In fact, it is sometimes so busy that the user becomes frustrated by the computer's slow responses.

## Client/Server Environment

A client/server computing environment splits the computing function between a central computer and users' computers. The users are given personal computers or workstations so that some of the computation responsibility can be moved from the central computer and assigned to the workstations. In the client/server environment, the users' workstations are called the client. The central computer, which may be a powerful computer, minicomputer, or central mainframe system, is known as the server. Because the work is now shared between the users' computers and the central computer, response time and monitor display are faster and the users are more productive. **Figure 1-7** shows a typical client/server environment.

## Distributed Computing

A distributed computing environment, introduced in the 1990s, is a large network of servers and clients scattered geographically to provide a seamless integration of computing functions. In a distributed computing environment (illustrated in **Figure 1-8**), all resources of the servers, such as CPU and memory, are pooled together to provide

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 1-7** The client/server environment

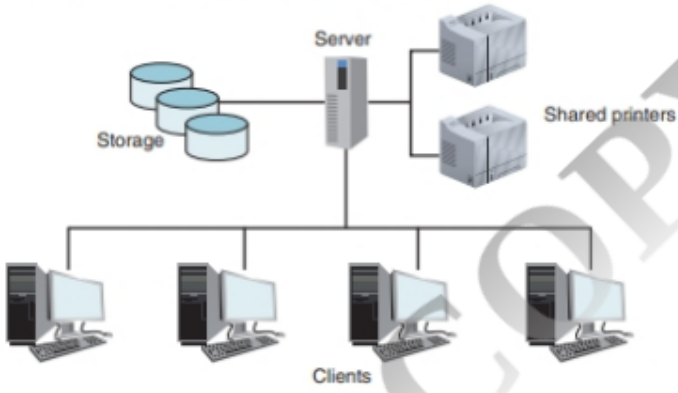
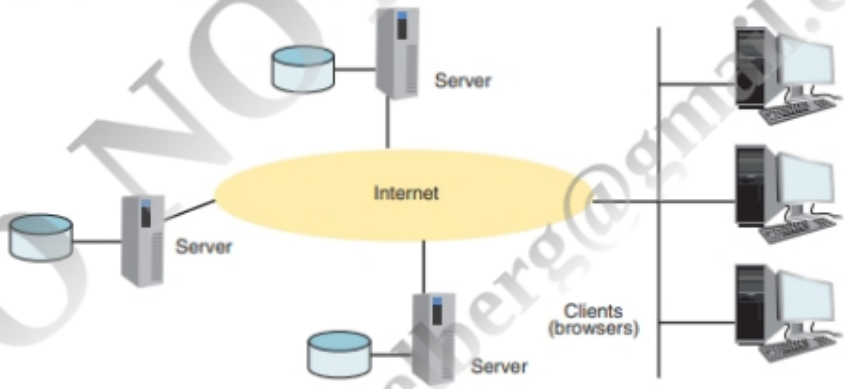



**Figure 1-8** Distributed computing




high processing power for applications. This environment provides a reliable, scalable, and highly available network. Large organizations like Amazon use many servers housed in many data centers to provide its shopping services for online consumers.

### Cloud Computing

In the early 2000s, cloud computing (illustrated in **Figure 1-9**) started to take shape, eventually becoming an essential computing technology. In a cloud computing environment, servers and storage devices are spread out across multiple geographic areas and connected via the Internet. Cloud computing environments provide services such as Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). For example, Microsoft offers Office 365 software as SaaS, which means users can use the software without having to download it and install it on their own computers. Many high-tech companies, such as Amazon, IBM, Google, Microsoft, and Oracle, offer cloud services.

## 1.3 Computer Languages

To write a program for a computer, you must use a computer language. Over the years, computer languages have evolved from machine languages to natural languages. A summary of computer languages is provided in **Figure 1-10**.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 1-9** Cloud computing

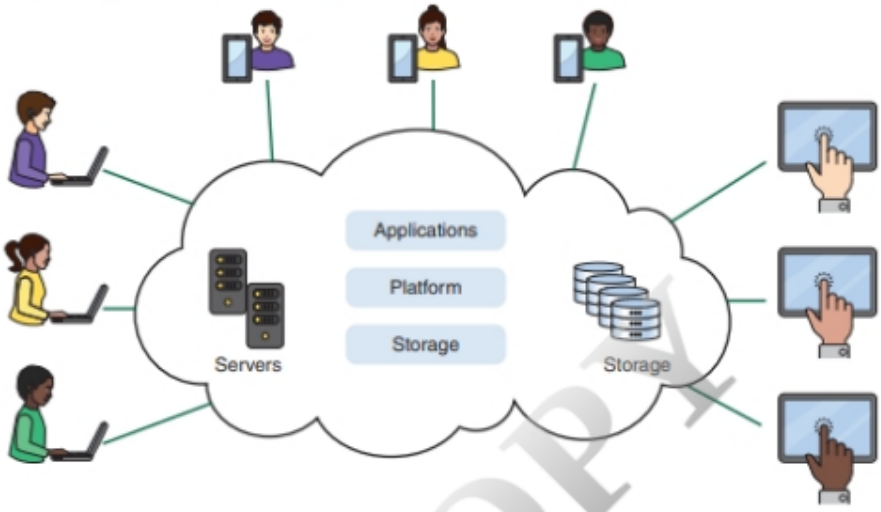
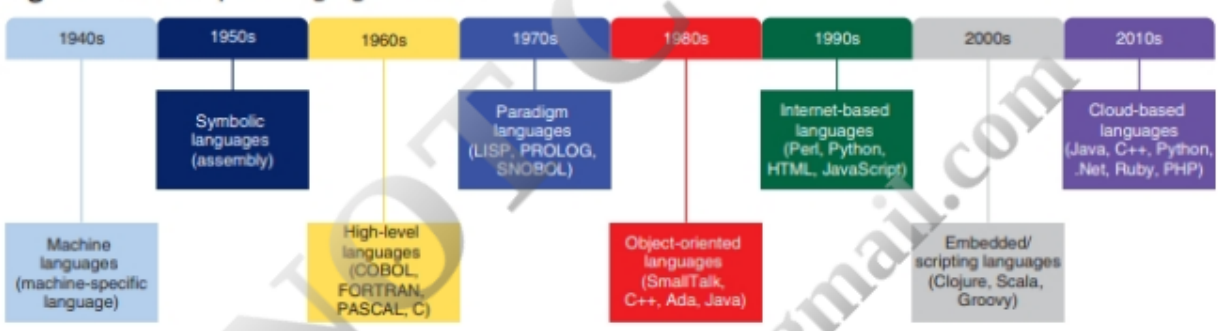



**Figure 1-10** Computer language evolution




## Machine Languages

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language, which is made of streams of 0s and 1s. **Program 1-1** shows an example of a machine language. This program multiplies two numbers and prints the results.

**Program 1-1** Multiplication program in machine language

```
1   00000000    00000100    0000000000000000
2   01011110    00001100    11000010    0000000000000010
3   11101111    00010110    0000000000000101
4   11101111    10011110    0000000000001011
5   11111000    10101101    11011111    0000000000010010
6   01100010    11011111    0000000000010101
```

*(continue)*

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Program 1-1 Multiplication program in machine language *(continued)*

```
7   11101111    00000010    11111011    0000000000010111
8   11110100    10101101    11011111    0000000000011110
9   00000011    10100010    11011111    0000000000100001
10  11101111    00000010    11111011    0000000000100100
11  01111110    11110100    10101101
12  11111000    10101110    11000101    0000000000101011
13  00000110    10100010    11111011    0000000000110001
14  11101111    00000010    11111011    0000000000110100
15  01010000    11010100    0000000000111011
16  00000100    0000000000111101
```

The instructions in machine language consist of streams of 0s and 1s because the internal circuits of a computer are made of switches, transistors, and other electronic devices that can be in one of two states: off or on. The off state is represented by 0; the on state is represented by 1. Although more advanced computer languages have been developed since the early days of computing, the hardware of any computer can still only understand machine language. So to actually cause a computer to do something, all programs must ultimately be translated into machine language.

## Symbolic Languages

It became obvious that few programs would be written if programmers had to work in machine language, which is very difficult. In the early 1950s, Admiral Grace Hopper, a mathematician and naval officer, developed the concept of a special computer program that would convert programs into machine language. These early programming languages simply mirrored the machine languages using symbols, or mnemonics, to represent the various machine language instructions. Because they used symbols, these languages were known as symbolic languages. **Program 1-2** shows the

**Program 1-2** The multiplication program in symbolic language

```
1       entry   main,<r2>
2       subl2   #12, sp
3       jsb     C$MAIN_ARGS
4       movab   $CHAR_STRING_CON
5       pushal  -8(fp)
6       pushal  (r2)
7       calls   #2,SCANF
8       pushal  -12(fp)
9       pushal  3(r2)
10      calls   #2,SCANF
11      mull3   -8(fp),-12(fp),-
12      pushal   6(r2)
13      calls   #2,PRINTF
14      clrl    r0
15      ret
```

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

multiplication program in a symbolic language. Note that symbolic language code is usually laid out in three columns, one containing labels, one operators, and one operands. For the sake of simplicity, the code in Program 1-2 contains only operators and operands. The numbers on the left are line numbers included to make the code easier to discuss.

Because a computer does not understand symbolic language, programs written in symbolic languages must be translated to machine language. A special program called an assembler translates symbolic code into machine language. Because symbolic languages had to be assembled into machine language, these languages became known as assembly languages. Assembly languages use symbols, or mnemonics, to represent the various machine language instructions. This name is still used today for symbolic languages that closely represent the machine language of a computer's hardware platform.

## High-Level Languages

Although symbolic languages greatly improved programming efficiency, they still required programmers to write code specifically designed for the hardware of a particular computer. Working with symbolic languages was also very tedious because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of high-level languages.

High-level languages are portable to many different computers, allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer. High-level languages are designed to relieve the programmer from the details of the assembly language. High-level languages share one thing with symbolic languages, however: They must be converted to machine language. The process of converting them is known as compilation.

The first widely used high-level language, FORTRAN, was created by John Backus and an IBM team in 1957; it is still widely used today in scientific and engineering applications. (FORTRAN is an acronym for FORmula TRANslation.) Following soon after FORTRAN was the language COBOL. (COBOL is an acronym for COmmon Business-Oriented Language.) Admiral Hopper was again a key figure in the development of the COBOL business language.

C is a high-level language used for system software and new application code. **Program 1-3** shows the multiplication program as it would appear in the C language. In this program, colors are used to indicate different parts of the code.

### Program 1-3 | The multiplication program in C

```
1  /* This program reads two integers from the keyboard
2     and prints their product.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7
8  int main (void)
9  {
10 // Local Definitions
11    int number1;
12    int number2;
13    int result;
14
15 // Statements
```

*(continue)*

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Program 1-3 The multiplication program in C *(continued)*

```
16     scanf ("%d", &number1);
17     scanf ("%d", &number2);
18     result = number1 * number2;
19     printf ("%d", result);
20     return 0;
21 } // main
```

Depending on what program you use to write C code, you might see different colors, or you might see your code in simple black and white. You'll learn more about the meaning of the colors used for C code as you become a more experienced programmer. As in Program 1-2, the numbers on the left are line numbers included to make the code easier to discuss.

## 1.4 Creating and Running Programs

As you learned in the previous section, computer hardware can only understand machine language. In this section, we explain the procedure for turning a program written in C into machine language. The process is presented in a straightforward, linear fashion, but you should recognize that these steps are repeated many times during development to correct errors and make improvements to the code.

It is the job of the programmer to write and test the program. There are four steps in this process: (1) writing and editing the program, (2) compiling the program, (3) linking the program with the required library modules, and (4) executing the program. These steps are shown in **Figure 1-11**.

### Writing and Editing Programs

The software used to write programs is known as a text editor. In a text editor, you can enter, change, and store character data. It's also possible to use a word processor program, which you would normally use to write letters or reports. But typically, programmers use text editors, included with the compiler, which are specially designed for writing programs. These text editors include features designed to make writing code easier, including applying colors to specific parts of a program.

Some of the features you should look for in a text editor are search commands for locating and replacing statements, copy and paste commands for copying or moving statements from one part of a program to another, and formatting commands that allow you to set tabs to align statements.

After completing a program, you need to save the file so it can be input to the compiler. This saved file is known as a source file.

### Compiling Programs

The code in a source file must be translated into machine language. This is the job of the compiler. The C compiler is actually two separate programs: the preprocessor and the translator.

The preprocessor reads the source code and prepares it for the translator. While preparing the code, it scans for special instructions known as preprocessor commands. These commands tell the preprocessor to look for special code libraries, make substitutions in the code, and in other ways prepare the code for translation into machine language. The result of preprocessing is called the translation unit.

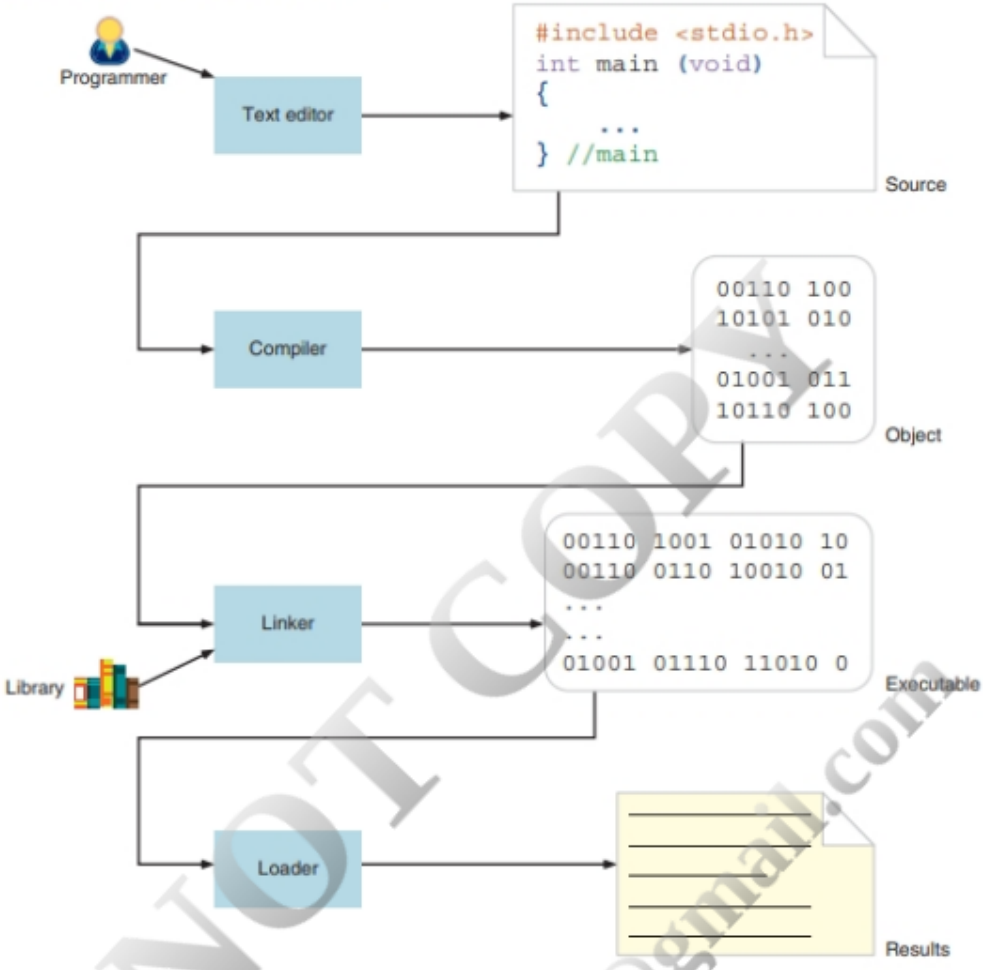
After the preprocessor has prepared the code for compilation, the translator does the actual work of converting the program into machine language. The translator reads the translation unit and writes the resulting object module to a

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 1-11** Building a C program




file that can then be combined with other precompiled units to form the final program. An object module is the code in machine language. Even though the output of the compiler is machine language code, it is not yet ready to run; that is, it is not yet executable because it does not have the required C and other functions included.

## Linking Programs

As you become a more experienced programmer, you will see that a C program is made up of many functions. When you write a C program, you might actually write some of the functions yourself. However, some functions, such as input/output processes and mathematical library functions, exist elsewhere and must be attached to the program. The linker assembles all of these functions, as well as the system's, into a final executable program.

## Executing Programs

After your program has been linked, it is ready for execution. To execute a program, you use an operating system command, such as `run`, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the loader. It locates the executable program and reads it into memory. When everything is loaded, the program takes control and begins execution. In today's integrated development environments, these steps are combined under one mouse click or pull-down window.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

In a typical program execution, the program reads data for processing, either from the user or from a file. After the program processes the data, it prepares the output. Data can be output to the user's monitor or to a file. When the program has finished its job, it tells the operating system, which then removes the program from memory.

## 1.5 System Development

You've now seen the steps that are necessary to build a program. In this section, we discuss *how* we go about developing a program. This critical process determines the overall quality and success of the completed program. If you carefully design each program using well-structured development techniques, your programs will be efficient, error-free, and easy to maintain.

**Note**

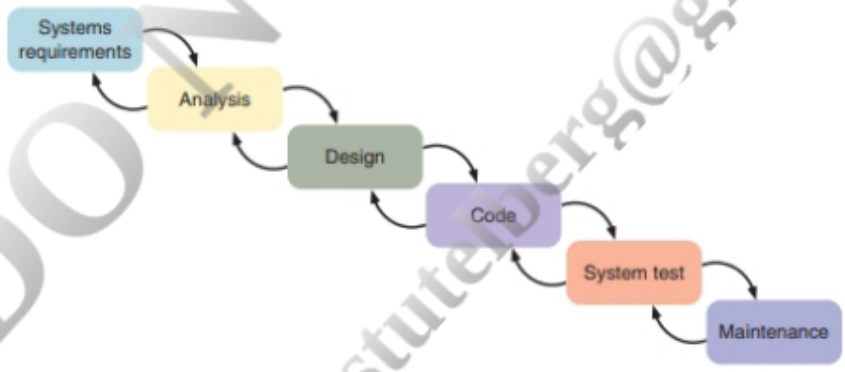
Many computer scientists believe that all programs contain at least one bug—an undetected error—that is just waiting to cause problems, given the right set of circumstances. Programs have run for years without problems only to fail when an unusual situation occurs. Perhaps the most famous bug was the one known as Y2K because it caused programs to fail on January 1, 2000.

### System Development Life Cycle

Today's large-scale, modern programming projects are built using a series of interrelated phases commonly referred to as the system development life cycle. Although the exact number and names of the phases differ depending on the environment, there is general agreement as to the steps that must be followed. Whatever the methodology, however, today's software engineering concepts require a rigorous and systematic approach to software development.

One very popular development life cycle is the waterfall model. Depending on the company and the type of software being developed, this model consists of between five and seven phases. **Figure 1-12** is one possible variation on the model.

**Figure 1-12** Waterfall model




The waterfall model starts with the systems requirements phase. In this phase, the systems analyst defines requirements that specify what the proposed system is to accomplish. The requirements are usually stated in terms that the user understands. In the analysis phase, the systems analyst looks at different alternatives from a system's point of view. In the design phase, the systems analyst determines how the system will be built. This includes determining the functions of the individual programs that will make up the system and designing the required files and the databases. Finally, in the fourth phase, code phase, the program is written. This is the phase that is explained in this book. After the programs have been written and tested to the programmer's satisfaction, the project proceeds to system test. All of the programs are tested together to make sure the system works as a whole. The final phase, maintenance phase, involves keeping the system working after it has been put into production.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

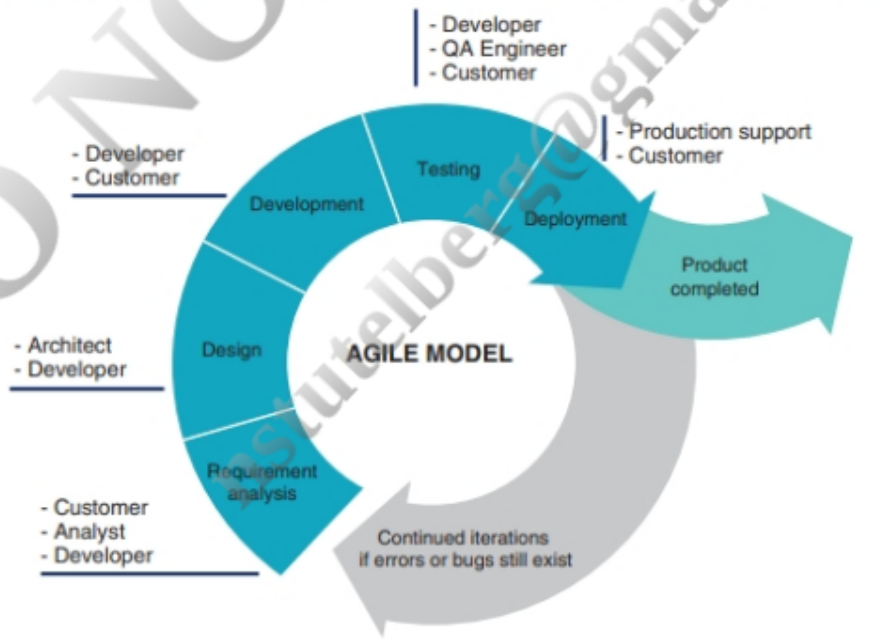
DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Although the implication of the waterfall approach is that the phases flow in a continuous stream from the first to the last, this is not really the case. Note the iteration as indicated by the backward-flowing arrows in Figure 1-12. As each phase is developed, errors and omissions will often be found in the previous work. When this happens, it is necessary to go back to the previous phase to rework it for consistency and to analyze the impact caused by the changes. Hopefully, this is a short rework. However, it often happens that, just as a project reaches the code and test phases, it becomes clear that it actually cannot be implemented and is therefore canceled. Depending on the scope of the project, this can result in losses of millions of dollars and years of development.

Agile, a widely used software development methodology, takes an adaptive, iterative approach to software development, with the goal of expediting the development life cycle and involving the user early in the development process. Customer feedback is solicited regularly and incorporated into program development. Agile advocates speedy development iterations, continuing until the full product is completed and deployed. Agile methodology has gained popularity due to four major advantages: fast development, adaptivity, collaboration of development members, and customer interaction in the development process. **Figure 1-13** illustrates the Agile development model. As you can see, development starts with collecting customer requirements, which the analyst and developer use to begin planning the program. In the design phase, the team creates a design document and product prototype. Coding begins in the development phase. Iterative cycles of customer feedback and development, known as sprints, continue until the product is finally deployed.

**Figure 1-13** Agile takes an adaptive, iterative approach to software development




## Program Development

Program Development involves these major steps: understanding the problem, developing a solution, writing the program, and then testing it. When you are assigned a program to develop, you will be given a program requirements statement and the design of any program interfaces. You should also receive an overview of the complete project so that you can understand how your part fits into the whole. Your job is to determine how to take the inputs you are given and convert them into the specified outputs. The entire process of creating the program is known as program design. To give you an idea of how this process works, let's look at a simple problem: calculating the square footage of a house. How do we go about doing this?

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Understand the Problem

The first step in solving any problem is to make sure you understand it. Start by reading the requirements statement carefully. Then, review your understanding of the problem with the user and the systems analyst. Often this involves asking questions to confirm your understanding.

For example, after reading the requirements statement for the square footage program, you should ask clarifying questions such as:

- What is the definition of square footage?
- How is the square footage going to be used?
  - For calculating a quote for home insurance?
  - For calculating the amount of paint required to paint the inside of the house? Or the outside?
  - For calculating the amount of carpet required to carpet all or part of the house?
- Is the garage included?
- Are closets and hallways included?

Each of the potential uses requires a different measure. If we don't clarify the exact purpose—that is, if we make assumptions about how the output is going to be used—the answer supplied by the program might be wrong.

As this example shows, even the simplest problem statements need clarification. Imagine how many questions must be asked for a programmer to write a program that will contain hundreds or thousands of detailed statements.

## Develop the Solution

After you fully understand the problem and have clarified any questions you may have, the next step is to develop the solution. Three tools will help in this task: (1) structure charts, (2) pseudocode, and (3) flowcharts. Generally, you will use only two of them—a structure chart and either pseudocode or a flowchart.

The structure chart is used to design the whole program. Pseudocode and flowcharts, on the other hand, are used to design the individual parts of the program. These parts are known as modules in pseudocode or functions in the C language.

### Structure Chart

A structure chart, also known as a hierarchy chart, shows the functional flow of the program. Large programs are complex structures consisting of many interrelated parts; thus, they must be carefully laid out. This task is similar to that used by a design engineer who is responsible for the operational design of any complex item. The major difference between the design built by a programmer and the design built by an engineer is that the programmer's product is software that exists only inside the computer, whereas the engineer's product is something that can be seen and touched.

The structure chart shows how the program will be broken into logical steps; each step will be a separate module. The structure chart shows the interaction between all the parts (modules) of the program.

It is important to realize that the design, as represented by the structure chart, is done before the program is written. In this respect, it is like the architect's blueprint. You would not start to build a house without a detailed set of plans. Yet one of the most common errors of both experienced and new programmers alike is to start coding a program before the design is complete and fully documented.

This rush to start is due in part to programmers thinking they fully understand the problem, when in fact some parts of the problem might not be so obvious, or still require clarification. By taking the time to design the program, programmers will continue to ask questions about the problem, and therefore will gain a better understanding of exactly what kind of program they need to write.

**Note** | An old programming proverb: Resist the temptation to code.

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The second reason programmers code before completing the design is just human nature. Programming is a tremendously exciting task. To see your design begin to take shape, to see your program creation working for the first time, brings a form of personal satisfaction that is a natural high. So often programmers jump in and start writing program code before they are actually ready.

In the business world, the completed structure design is reviewed by a committee, which conducts a structured walk-through of the program to determine whether it satisfies the requirements. The committee usually consists of a representative from the user community, one or two peer programmers, the systems/business analyst, and possibly a representative from the testing organization. The committee then offers constructive suggestions as to how to improve the design.

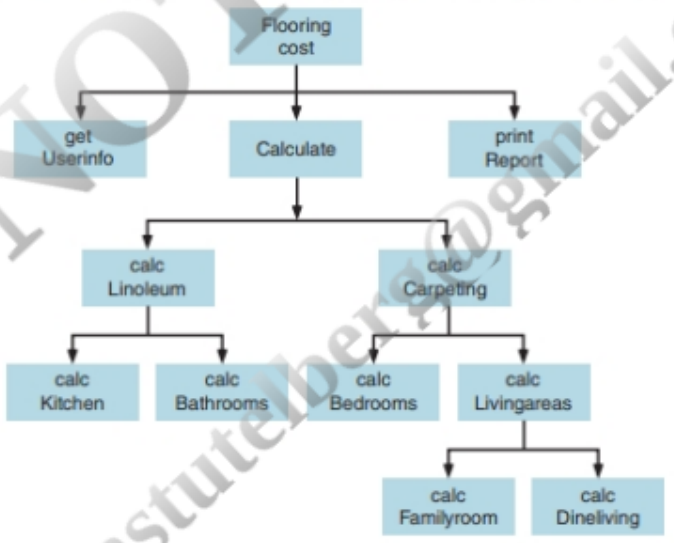
The primary intent of this review is to increase quality and save time. The earlier a mistake is detected, the easier and less expensive it is to fix it. So even if the walk-through eliminates only one or two problems, the time will be well spent. Naturally, in a programming class, you will not be able to convene a full committee and conduct a formal walk-through. What you can do, however, is review your design with some of your classmates and your professor.

Let's return to the problem at hand: calculating the square footage of a house. Let's assume the following answers to the questions raised in the previous section.

1. The purpose of calculating the square footage is to install linoleum and carpeting.
2. The garage and closets will not be considered.
3. The kitchen and bathrooms will be covered with linoleum; the rest of the house is to be carpeted.

With this understanding, you decide to write separate modules for the kitchen, bathroom(s), bedrooms, family room, and living room. You use separate modules because the various rooms may require a different quality of linoleum and carpeting. The structure chart for the design is shown in **Figure 1-14**.

**Figure 1-14** Structure chart for calculating square footage




Whether you use a flowchart or pseudocode to complete the design of your program will depend on your experience, the difficulty of the program you are designing, and the culture and standards of the organization where you work. It's helpful for new programmers to first learn program design by flowcharting because a flowchart is a visual tool that is easier to create than pseudocode. On the other hand, pseudocode is more common among professional programmers.

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

### Pseudocode

Pseudocode is part English, part program logic. Its purpose is to describe, in precise algorithmic detail, what the program is being designed to do. This requires defining the steps to accomplish the task in sufficient detail so that they can be converted into a computer program. Pseudocode excels at this type of precise logic. The pseudocode for determining the total amount of linoleum required for the bathroom is shown in **Algorithm 1-1**.

**Algorithm 1-1** | Pseudocode for Calculate BathRooms

```
1 prompt user to enter (input) linoleum price
2 prompt user and read number of bathrooms
3 set total bath area and baths processed to zero
4 while (baths processed < number of bathrooms)
  4.1 prompt user and read bath length and width
  4.2 total bath area = total bath area + bath length * bath width
  4.3 add 1 to baths processed
5 bath cost = total bath area * linoleum price
6 return bath cost
end Algorithm Calculate BathRooms
```

Most of the statements in pseudocode are easy to understand. A prompt is simply a displayed message telling the user what data are to be entered. The `while` statement is a loop that repeats the three statements that follow it and uses the number of bathrooms read in statement 2 to specify when to stop. Looping is a programming concept that allows us to repeat a block of code. In this case, it allows us to process the information for one or more bathrooms. You will learn more about loops as you become a more experienced programmer.

### Flowchart

A flowchart is a program design tool in which standard graphical symbols are used to represent the logical flow of data through a function. The flowchart in **Figure 1-15** shows the design for calculating the area and cost for the bathrooms. A few points merit comment here. This flowchart is basically the same as the pseudocode. It begins with prompts for the price of the linoleum and the number of bathrooms and reads these two pieces of data. The loop reads the dimensions for each bathroom. Finally, when the total area is known, the price is calculated and results are returned to `calcLinoleum`.

### Write the Program

Now it's time to write the program! But first, let's review the steps that we've used.

1. Understand the problem.
2. Develop a solution.
   a. Design the program—create the structure chart.
   b. Design the algorithms for the program using either flowcharting or pseudocode or both.
3. Write the program.

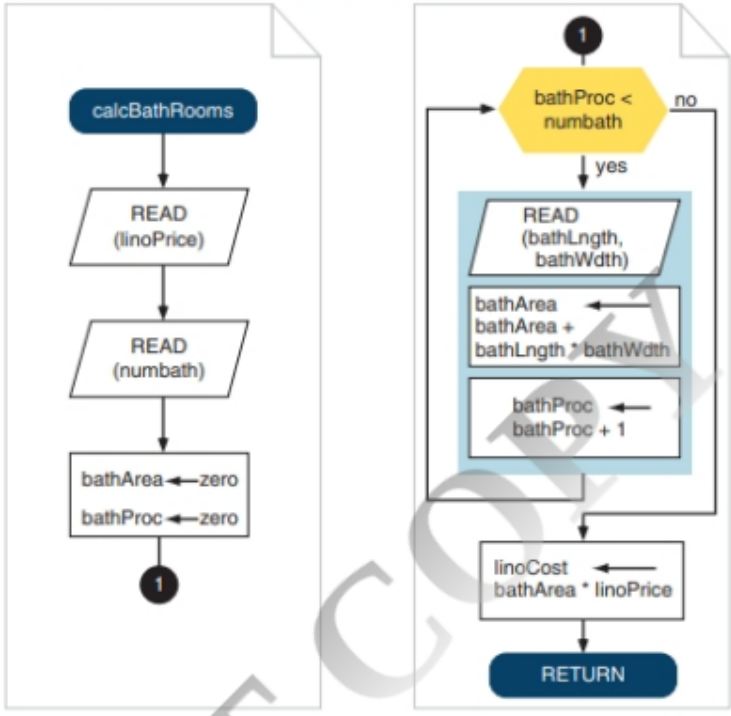
When you write a program, you start with the top box on the structure chart and work your way to the bottom. This is known as top-down implementation. You will find that it is a very easy and natural way to write programs, especially if you have done a solid job on your design.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 1-15** Flowchart for calculating bathrooms




As you begin learning C, your early programs will be so simple that they will require only one module, represented by the top box of the structure chart. Later, however, as you begin to write functions, the structure charts will get larger. At that time you will learn more techniques for writing structured programs. For now, concentrate on writing good pseudocode or flowcharts for the main part of your programs.

## Test the Program

After you write a program, you must test it. Program testing can be a very tedious and time-consuming part of program development. As the programmer, you are responsible for completely testing your program. In large development projects, there are often specialists known as quality assurance engineers who are responsible for testing the system as a whole—that is, for testing to make sure all the programs work together.

There are two types of testing: blackbox and whitebox. Blackbox testing is done by the system test engineer and the user. Whitebox testing is the responsibility of the programmer.

### Blackbox Testing

Blackbox testing gets its name from the concept of testing the program without knowing what is inside it—that is, without knowing how it works. In other words, the program is like a black box that conceals the code inside it.

Blackbox test plans are developed by looking only at the requirements statement. (This is only one reason why it is so important to have a good set of requirements.) The test engineer uses these requirements and his or her knowledge of systems development and the user's working environment to create a test plan that will then be used when the system is tested as a whole. You should always ask to see this test plan before writing the program. The test engineer's plan will help you make sure you fully understand the requirements and also help you create a solid test plan.

### Whitebox Testing

Whereas blackbox testing assumes that the tester knows nothing about the program, whitebox testing assumes that the tester knows everything about the program. In this case, the program is like a glass house in which everything is visible.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Whitebox testing is your responsibility. As the programmer, you know exactly what is going on inside the program. You must make sure that every instruction and every possible situation has been tested. That is not a simple task!

Experience will help you design good test data, but one thing you can do from the start is to get in the habit of writing test plans. You should start the test plan when you are in the design stage. As you build your structure chart, ask yourself what situations, especially unusual situations, you need to test for and make a note of them immediately because you won't remember them an hour later.

When you are writing flowcharts or pseudocode, you need to review them with an eye toward test cases and make additional notes of the cases you may need. Finally, while coding, you need to make notes about test cases that might be needed.

**Note** | Except for the simplest program, one set of test data will not ensure that the program is error- and bug-free.

When it is time to construct test cases, you review your notes and organize them into logical sets. Except for very simple student programs, one set of test data will never completely validate a program. For large-scale development projects, 20, 30, or even more test cases may need to be run to validate a program.

Finally, while you are testing, you must think of more test cases. Again, write them down and incorporate them into your test plan. After your program is finished and in production, you will need the test plan again when you make modifications to the program.

How do you know when a program is completely tested? In reality, there is no way to know for sure. But there are a few things you can do to help the odds. While some of these concepts will not be clear until you have read other chapters, they are included here for completeness.

1. Verify that every line of code has been executed at least once. Fortunately, there are programming tools on the market today that will do this monotonous task.
2. Verify that every conditional statement in your program has executed both the true and false branches, even if one of them is null.
3. For every condition that has a range, make sure the tests include the first and last items in the range, as well as items below the first and above the last—the most common mistakes in array range tests occur at the extremes of the range.
4. If error conditions are being checked, make sure all error logic is tested. This may entail making temporary modifications to the program to force the errors. (For instance, an input/output error usually cannot be created—it must be simulated.)

## Software Engineering

Software engineering is the establishment and use of sound engineering methods and principles to obtain software that is reliable and that works on real machines. This definition, from the first international conference on software engineering in 1969, was proposed 30 years after the first computer was built. During that period, software was more of an art than a science. In fact, one of the most authoritative treatments of programming describes it as an art: *The Art of Computer Programming*. This three-volume series, originally written by Donald E. Knuth in the late 1960s and early 1970s, is considered the most complete discussion of many computer science concepts.

Because the science and engineering base for building reliable software did not yet exist, programs written in the 1950s and 1960s were a maze of complexity known as "spaghetti code." It was not until Edsger Dijkstra wrote a letter to the editor of the *Communications of the ACM* (Association of Computing Machinery) in 1968 that the concept of structured programming began to emerge.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Dijkstra was working to develop algorithms that would mathematically prove program accuracy. He proposed that any program could be written with only three constructs or types of instructions: (1) sequences, (2) the `if...else` selection statement, and (3) the `while` loop. As you will see, language developers have added constructs, such as the `for` loop and the `switch` in C. These additional statements are simply enhancements to Dijkstra's basic constructs that make programming easier. Today, virtually all programming languages offer structured programming capabilities.

Throughout this text you will learn about the concepts of good software engineering. Chief among them is the concept of structured programming and a sound programming style. The "Software Engineering" section in each chapter will include a discussion of these concepts, with specific emphasis on the application of the material in the chapter.

The tools of programming design have also changed over the years. In the first generation of programming, one primary tool was a block diagram. This tool provided boxes, diamonds, and other flowchart symbols to represent different instructions in a program. Each instruction was contained in a separate symbol. This concept allowed programmers to write a program on paper and check its logic flow before they entered it in the computer.

With the advance of symbolic programming, the block diagram gave way to the flowchart. Although the block diagram and flowchart look similar, the flowchart does not contain the detail of the block diagram. Many instructions are implied by the descriptive names put into the boxes; for example, the read statements in **Figure 1-15** imply the prompt. Flowcharts have largely given way to other techniques in program design, but they are still used today by many programmers for working on a difficult logic problem.

Today's programmers are most likely to use a high-level design tool such as tight English or pseudocode. We will use pseudocode throughout the text to describe many of the algorithms you will be developing.

Finally, the last several years have seen the automation of programming through the use of computer-assisted software engineering (CASE) tools. These tools make it possible to determine requirements, design software, and develop and test software in an automated environment using programming workstations. The discussion of the CASE environment is beyond the scope of this text and is left for courses in systems engineering.

## Tips and Common Programming Errors

1. Become familiar with the text editor in your system so you will be able to create and edit your programs efficiently. The time spent learning different techniques and shortcuts in a text editor will save time in the future.
2. Also, become familiar with the compiler commands and keyboard shortcuts. On most computers, a variety of options are available to be used with the compiler. Make yourself familiar with all of these options.
3. Read the compiler's error messages. Becoming familiar with the types of error messages and their meanings will be a big help as you learn C.
4. Remember to save and compile your program each time you make changes or corrections in your source file. When your program has been saved, you won't lose your changes if a program error causes the system to fail during testing.
5. Run your program many times with different sets of data to be sure it does what you want.
6. The most common programming error is not following the old proverb to "resist the urge to code." Make sure you understand the requirements and take the time to design a solution before you start writing code.

DO NOT COPY. nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Summary

- A computer system consists of hardware and software.
- Computer hardware consists of a central processing unit (CPU), primary memory, input devices, output devices, and secondary storage.
- Software consists of two broad categories: system software and application software.
- The components of system software are the operating system, system support, and system development.
- Application software is divided into general-purpose applications and application-specific software.
- Over the years, programming languages have evolved from machine language, to symbolic language, to high-level languages.
- The C language is a high-level language.
- The software used to write programs is known as a text editor.
- The file created by a text editor is known as a source file.
- The code in a source file must be translated into machine language using the C compiler, which is made of two separate programs: the preprocessor and the translator.
- The file created by the compiler is known as an object module.
- An object module is linked to the standard functions necessary for running the program by the linker.
- A linked program is run using a loader.
- The system development life cycle is a series of interrelated steps that provide a rigorous and systematic approach to software development.
- To develop a program, a programmer must complete the following steps:
  1. Understand the problem
  2. Develop a solution using structure charts and either flowcharts or pseudocode
  3. Write the program
  4. Test the program
- The development of a test plan starts with the design of the program and continues through all steps in program development.
- Blackbox testing consists primarily of testing based on user requirements.
- Whitebox testing, executed by the programmer, tests the program with full knowledge of its operational weaknesses.
- Testing is one of the most important parts of your programming task. You are responsible for whitebox testing; the systems analyst and user are responsible for blackbox testing.
- Software engineering is the application of sound engineering methods and principles to the design and development of application programs.

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Key Terms

Agile
analysis phase
application software
application-specific software
assembly languages
blackbox testing
central processing unit (CPU)
client
client/server
code phase
compilation
compiler
computer language
computer system
design phase
development software
distributed computing environment
executable program
flowchart
general-purpose software
hard copy
hardware
high-level languages
input
input device
linker
loader
machine language
main memory
maintenance phase
object module
operating system
output device
PC
personal computer
preprocessor
preprocessor commands
primary storage
program development
pseudocode
secondary storage
server
soft copy
software
source file
structure chart
symbolic languages
system development life cycle
system software
system support software
systems requirements phase
system test
text editor
time-sharing environment
translation unit
translator
waterfall model
whitebox testing

## Review Questions

1. Computer software is divided into two broad categories: system software and operational software.
   **a.** True **b.** False
2. The operating system provides services such as a user interface, file and database access, and interfaces to communications systems.
   **a.** True **b.** False
3. The first step in system development is to create a source program.
   **a.** True **b.** False
4. The programmer design tool used to design the whole program is the flowchart.
   **a.** True **b.** False
5. Blackbox testing gets its name from the concept that the program is being tested without knowing how it works.
   **a.** True **b.** False
6. Which of the following is a component(s) of a computer system?
   **a.** Hardware
   **b.** Software
   **c.** Both hardware and software
   **d.** Pseudocode
   **e.** System test
7. Which of the following is not an example of application software?
   **a.** Database management system
   **b.** Language translator
   **c.** Operating system
   **d.** Accounting system
   **e.** Virus detection
8. Which of the following is not a computer language?
   **a.** Assembly/symbolic language
   **b.** Binary language
   **c.** High-level languages
   **d.** Machine language
   **e.** Natural language

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

9. The computer language that most closely resembles machine language is _____.
   a. assembly/symbolic
   b. COBOL
   c. FORTRAN
   d. high level

10. The tool used by a programmer to convert a source program to a machine language object module is a _____.
   a. compiler
   b. language translator
   c. linker
   d. preprocessor
   e. text editor

11. The _____ contains the programmer's original program code.
   a. application file
   b. executable file
   c. object file
   d. source file
   e. text file

12. The series of interrelated phases that is used to develop computer software is known as _____.
   a. program development
   b. software engineering
   c. system development life cycle
   d. system analysis
   e. system design

13. The _____ is a program design tool that is a visual representation of the logic in a function within a program.
   a. flowchart
   b. program map
   c. pseudocode
   d. structure chart
   e. waterfall model

14. The test that validates a program by ensuring that all of its statements have been executed—that is, by knowing exactly how the program is written—is _____.
   a. blackbox testing
   b. destructive testing
   c. nondestructive testing
   d. system testing
   e. whitebox testing

15. Which of the following is not an advantage of an Agile software development model?
   a. Rapid development
   b. Customer involvement
   c. Very structured
   d. Adaptive
   e. Team collaboration

## Exercises

16. Describe the two major components of a computer system.
17. Computer hardware is made up of five parts. List and describe them.
18. Describe the major differences between a time-sharing and a client/server environment.
19. Describe the two major categories of software.
20. What is the purpose of an operating system?
21. Identify at least two types of system software that you will use when you write programs.
22. Give at least one example of general-purpose and one example of application-specific software.
23. List the levels of computer languages discussed in the text.
24. What are the primary differences between symbolic and high-level languages?
25. What is the difference between a source program and an object module?
26. Describe the basic steps in the system development life cycle.
27. What documentation should a programmer receive to be able to write a program?
28. List and explain the steps that a programmer follows in writing a program.

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**29.** Describe the three tools that a programmer may use to develop a program solution.

**30.** What is meant by the old programming proverb, "Resist the temptation to code"?

**31.** What is the difference between blackbox and whitebox testing?

**32.** What is software engineering?

## Problems

**33.** Write pseudocode for `calcLivingAreas`, based on the structure chart shown in Figure 1-14.

**34.** Create a flowchart for a routine task, such as calling a friend, that you do on a regular basis.

**35.** Write pseudocode for the flowchart you created in Problem 34.

**36.** Create a flowchart to convert Fahrenheit temperature to Celsius and then write pseudocode for the flowchart.

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

# CHAPTER 2

# Introduction to the C Language

## Learning Objectives

**When you complete this chapter, you will be able to:**

- **2.1** Describe the development of the C language
- **2.2** Describe the structure of a C-language program
- **2.3** Create good identifiers for objects in a program
- **2.4** List, describe, and use the C basic data types
- **2.5** Create and use variables in a program
- **2.6** Create and use constants in a program
- **2.7** Use simple input and output statements

## 2.1 Background

In this chapter you will learn about the basics of the C language. You will write your first program, which is traditionally known in C as the "Hello World," or "Greeting," program. Along the way you will learn about the concepts of data types, constants, and variables. Finally, you will learn about two C library functions that read and write data. Because this chapter is just an introduction to C, most of these topics are covered only in sufficient detail to enable you to write your first program. They will be fully developed in future chapters.

We'll start with the evolution of the C language. Computer languages evolved from machine languages to high-level languages like C. Because you are going to spend considerable time working with C, you should have some idea of its origins and evolution.

C is a structured programming language. It is considered a high-level language because it allows the programmer to concentrate on the problem at hand and not worry about the machine the program will be using. While many languages claim to be machine-independent, C is one of the closest to achieving that goal. That is another reason why it is used by software developers whose applications have to run on many different hardware platforms.



Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

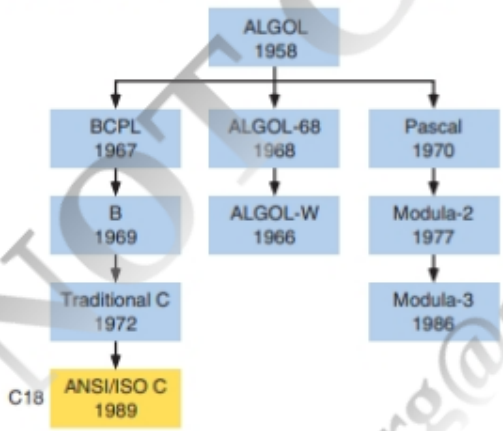
Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

C, like most modern languages, is derived from ALGOL, the first language to use a block structure made up of sequence of code statements. ALGOL never gained wide acceptance in the United States, but it was widely used in Europe.

ALGOL's introduction in the early 1960s paved the way for the development of structured programming concepts. Some of the first work was done by two computer scientists, Corrado Bohm and Giuseppe Jacopini, who published a paper in 1966 that defined the concept of structured programming. Another computer scientist, Edsger Dijkstra, popularized this concept. His letter to the editors of the *Communications of the ACM* (Association of Computing Machinery) brought the structured programming concept to the attention of the computer science community.

Several obscure languages preceded the development of C. In 1967, Martin Richards developed a language he called Basic Combined Programming Language, or BCPL. Ken Thompson followed in 1970 with a similar language he simply called B. B was used to develop the first version of UNIX, one of the popular network operating systems in use today. Finally, in 1972, Dennis Ritchie developed C, which took many concepts from ALGOL and BCPL. This path, along with several others, is shown in **Figure 2-1**.

**Figure 2-1** Taxonomy of the C language




What is known as traditional C is this 1972 version of the language, as documented and popularized in a 1978 book, *The C Programming Language*, by Brian W. Kernighan and Dennis Ritchie. In 1983, the American National Standards Institute (ANSI) began the definition of a standard for C. It was approved in December 1989. In 1990, the International Standards Organization (ISO) adopted the ANSI standard. This version of C is known as C89.

In 1995, minor changes were made to the standard. This version is known as C95. A much more significant update was made in 1999. The changes incorporated into the standard, now known as C99, are summarized in the following list.

1. Extensions to the character type to support non-English characters
2. A Boolean type
3. Extensions to the integer type
4. Inclusion of type definitions in the `for` statement
5. Addition of imaginary and complex types

In 2011 the C standards known as C11 incorporated the C++ style line comment (//). In addition, C11 included new features like multithreading and bounds-checking, and other improvements that allowed for better compatibility with C++. In 2018, a new version of C, known as C18, was published. It contains technical corrections but no new features. This text is based on C18 standards.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

# 2.2 C Programs

It's time to write your first C program! This section will present and demonstrate the basic parts of a C program.

## Structure of a C Program

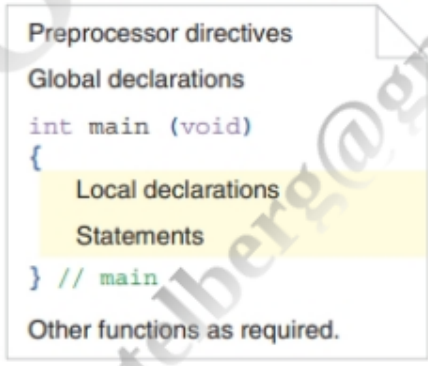
Every C program is made of one or more preprocessor commands, a global declaration section, and one or more functions. The global declaration section comes at the beginning of the program. This will be discussed more later, but the basic idea of global declarations is that they are visible to all parts of the program.

The work of the program is carried out by its functions, blocks of code that accomplish a task within a program. One, and only one, of the functions must be named `main`. The `main` function is the starting point for the program. All functions in a program, including `main`, are divided into two sections: the declaration section and the statement section. The declaration section is at the beginning of the function. It describes the data that you will be using in the function. Declarations in a function are known as local declarations because they are visible only to the function that contains them. (By contrast, global declarations, which you will learn about later, are visible to every function in the program.)

The statement section follows the declaration section. It contains the instructions that cause the computer to do something, such as add two numbers. In C, these instructions are written in the form of statements, which gives the name for the section.

**Figure 2-2** shows the parts of a simple C program. Everything in this program is explained except for the preprocessor commands. They are special instructions to the preprocessor that tell it how to prepare the program for compilation. One of the most important of the preprocessor commands, and one that is used in virtually all programs, is include. The include command tells the preprocessor that the information from selected libraries known as header files is needed. In today's complex programming environments, it is almost impossible to write even the smallest of programs without at least one library function. In your first program, you will use one include command to tell C that you need the input/output library to write data to the monitor.

**Figure 2-2** Structure of a C program




## Your First C Program

Your first C program will be very simple (see **Figure 2-3**). It will have only one preprocessor command, no global declarations, and no local definitions. Its purpose will be simply to print a greeting to the user. Therefore, its statement section will have only two statements: one that prints a greeting and one that stops the program.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 2-3** The greeting program


## Preprocessor Commands

The preprocessor commands come at the beginning of the program. All preprocessor commands start with a pound sign (#); this is just one of the rules of C known as its syntax. Preprocessor commands can start in any column, but they traditionally start in column 1.

The preprocessor command tells the compiler to include the standard input/output library file in the program. You need this library file to print a message to the terminal. Printing is one of the input/output processes identified in this library. The complete syntax for this command is shown below.

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

The syntax of this command must be exact. Because it is a preprocessor command, it starts with the pound sign. There can be no space between the pound sign and the keyword, `include`. Include means just what you would think it does. It tells the preprocessor that you want the library file in the pointed brackets (< >) included in your program. The name of the header file is `stdio.h`. This is an abbreviation for "standard input/output header file."

### main

The executable part of the program begins with the function `main`, which is identified by the function header shown below. For now, all you need to understand is that `int` says that the function will return an integer value to the operating system, that the function's name is `main`, and that it has no parameters (the parameter list is void). Note that there is no punctuation after the function header.

```
int main (void)
```

Within `main` there are two statements: one to print your message and one to terminate the program. The `print` statement uses a library function to do the actual writing to the monitor. To invoke or execute this `print` function, you **call** it—that is, you include a statement that explicitly refers to the function and specifies any parameters may require, which are values the program must pass the function so it can do its work. All function call statements consist of the name of the function, in this case `printf`, followed by a parameter list enclosed in parentheses. For your simple program, the parameter list simply contains what you want displayed, enclosed in two double quote marks (`"..."`). The `\n` at the end of the message tells the computer to advance to the next line in the output.

The last statement in your program, `return 0`, terminates the program and returns control to the operating system. One last thing: The function `main` starts with an open brace ({) and terminates with a close brace (}).

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Comments

Although it is reasonable to expect that a good programmer should be able to read code, sometimes the meaning of a section of code is not entirely clear. This is especially true in C. Thus, it is helpful if the person who writes the code places some explanations, known as comments, in the code to help the reader. Such comments are merely internal program documentation. The compiler ignores these comments when it translates the program into executable code. To identify a comment, C uses two different formats: block comments and line comments.

### Block Comment

A block comment is used when the comment will span several lines. This comment is called a format block comment. It uses opening and closing comment tokens. A token is one or more symbols understood by the compiler to indicate code. Each comment token is made of two characters that, taken together, form the token; there can be no space between them. The opening token is `/*` and the closing token is `*/`. Everything between the opening and closing comment tokens is ignored by the compiler. The tokens can start in any column and they do not have to be on the same line. The only requirement is that the opening token must precede the closing token. **Figure 2-4** shows two examples of block comments.

**Figure 2-4** Examples of block comments

```
/*  This is a block comment that
    covers two lines.                */
/*
**  It is a very common style to put the opening token
**  on a line by itself, followed by the documentation
**  and then the closing token on a separate line. Some
**  programmers also like to put asterisks at the beginning
**  of each line to clearly mark the comment.
*/
```

### Line Comment

The second format, the line comment, uses two slashes (`//`) to identify a comment. This format does not require an end-of-comment token; the end of the line automatically ends the comment. Programmers generally use this format for short comments. The line-comment token can start anywhere on the line. **Figure 2-5** contains two examples of line comments.

**Figure 2-5** Examples of line comments

```
// This is a whole line comment
a = 5; // This is a partial line comment
```

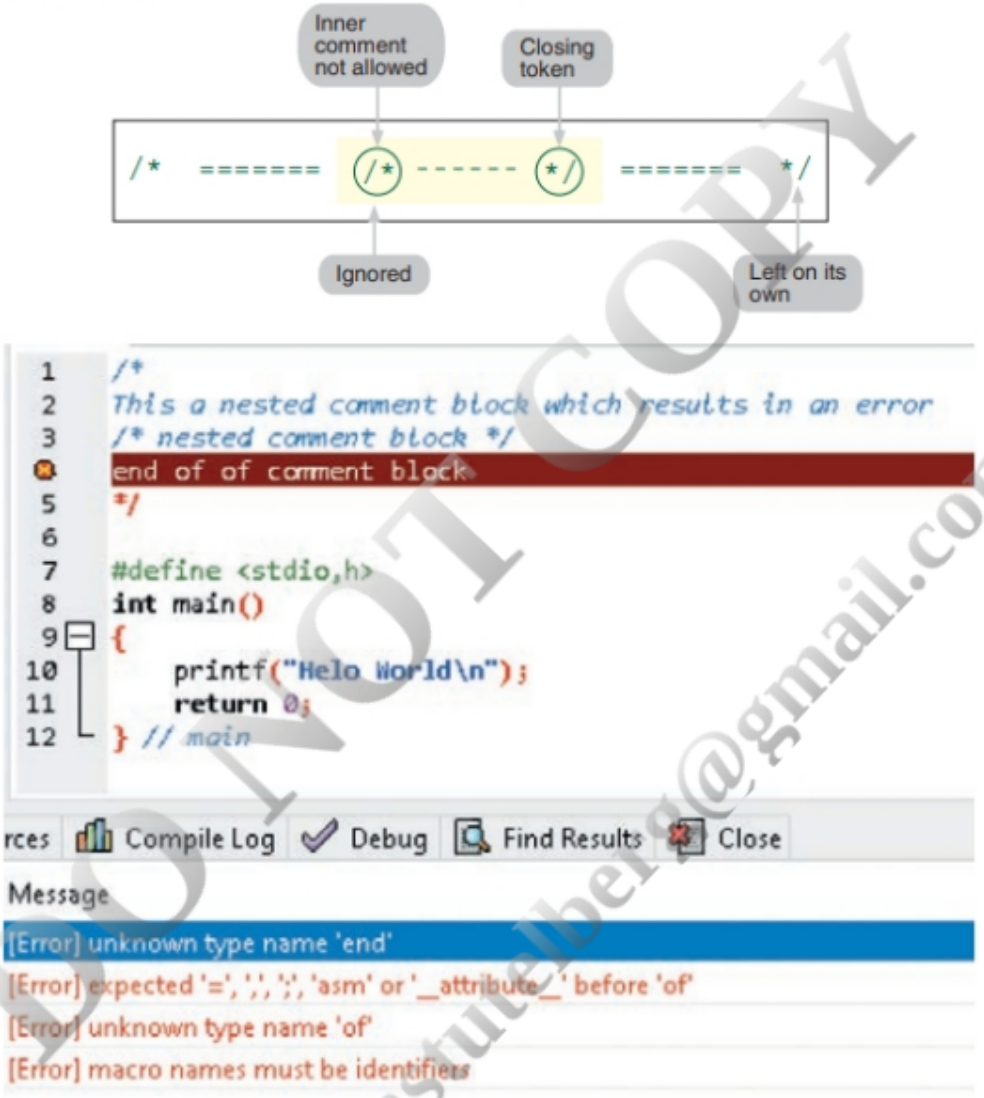
Although they can appear anywhere, comments cannot be nested. In other words, comments inside comments are not allowed. After the compiler sees an opening block-comment token, it ignores everything it sees until it finds the closing token. Therefore, the opening token of the nested comment is not recognized, and the ending token that matches the first opening token is left standing on its own. This error is shown in **Figure 2-6**.

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 2-6** Nested block comments are invalid




## The Greeting Program

**Program 2-1** shows the greeting with comments at the beginning explaining what the program is going to do. It should be emphasized here that each program begins with documentation explaining the purpose of the program. This is considered a best practice according to professional programming style. Program 2-1 also includes comments identifying the declaration and statement sections. The numbers on the left in Program 2-1, and in other programs in the text, are line numbers that you can use as references when discussing the code. They are not part of the program.

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Program 2-1 The greeting program

```
1  /* The greeting program. This program demonstrates
2     some of the components of a simple C program.
3     Written by: your name here
4     Date:  date program written
5  */
6  #include <stdio.h>
7
8  int main (void)
9  {
10 // Local Declarations
11
12 // Statements
13
14    printf("Hello World!\n");
15
16    return 0;
17 } // main
```

## 2.3 Identifiers

One feature present in all high-level computer languages is the identifier. Identifiers allow the programmer to name data and other objects in the program. Each identified object in the computer is stored at a unique address. The main purpose of the identifiers is to symbolically represent data locations. Otherwise, we would have to know and use each object's address, which would make programming very difficult. Rather than using specific memory addresses, it's simpler to assign identifiers to data and let the compiler keep track of where the data are physically located.

Different programming languages use different syntactical rules to form identifiers. In C, the rules for identifiers are very simple. The only valid name symbols are the capital letters A through Z, the lowercase letters a through z, the digits 0 through 9, and the underscore. The first character of the identifier cannot be a digit.

To avoid confusion, make sure your names do not duplicate system names. For example, do not begin an application program with an underscore, because many of the identifiers in the C system libraries start with an underscore. Also, keep in mind that names cannot be keywords. Keywords, also known as reserved words, include syntactical words, such as `if` and `while`.

Good identifier names are descriptive but short and often contain abbreviations. One way to abbreviate an identifier is to remove any vowels in the middle of the word. For example, `student` could be abbreviated `stdnt`. C allows names to be up to 63 characters long. If the names are longer than 63 characters, then only the first 63 are used. **Table 2-1** summarizes the rules for identifiers.

**Table 2-1** Rules for Identifiers

| |
|---|
| 1. First character must be alphabetic character or underscore. |
| 2. Must consist only of alphabetic characters, digits, or underscores. |
| 3. First 63 characters of an identifier are significant. |
| 4. Cannot duplicate a keyword. |

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

You might be curious as to why the underscore is included among the possible characters that can be used for an identifier. It is there so that you can separate different parts of an identifier. To make identifiers descriptive, you may combine two or more words. When the names contain multiple words, the underscore makes it easier to read the name. As indicated earlier, an identifier must start with a letter or underscore; it may not include a space or a hyphen.

Another way to separate the words in a name is to capitalize the first letter in each word. The traditional method of separation in C uses the underscore. A growing group of programmers, however, prefer to capitalize the first letter of each word. **Table 2-2** contains examples of valid and invalid names.

Two more comments about identifiers. Note that some of the identifiers in Table 2-2 are capitalized—that is, they start with uppercase letters. Typically, capitalized names are reserved for preprocessor-defined names. The second comment is that C is case sensitive. This means that even though two identifiers are spelled the same, if the case of each corresponding letter doesn't match, C thinks of them as different names. Under this rule, num, Num, and NUM are three different identifiers.

**Table 2-2** Examples of valid and invalid names

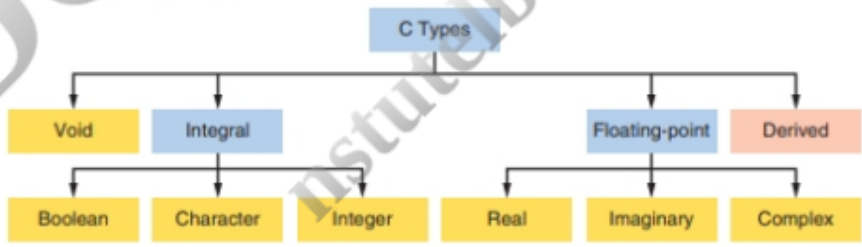
| VALID NAMES | | INVALID NAMES | |
|---|---|---|---|
| `a` | `// Valid but poor style` | `$sum` | `// $ is illegal` |
| `student_name` | | `2names` | `// First char digit` |
| `_aSystemName` | | `sum-salary` | `// Contains hyphen` |
| `_Bool` | `// Boolean System id` | `stdnt Nmbr` | `// Contains spaces` |
| `INT.MIN` | `// System Defined Value` | `int` | `// Keyword` |

## 2.4 Types

A type defines a set of values and a set of operations that can be applied on those values. For example, a light switch can be compared to a computer type. It has a set of two values, on and off. Only two operations can be applied to a light switch: turn on and turn off.

The C language has defined a set of types that can be divided into four general categories: void, integral, floating-point, and derived, as shown in **Figure 2-7**.

**Figure 2-7** Data types




This chapter focuses on only the first three types (void, Integral, and Floating-point). You will learn more about the derived type as you gain more experience as a programmer.

### Void Type

The void type, designated by the keyword `void`, has no values and no operations, Although having no values and operations might seem unusual, the `void` type is a very useful data type. For example, it is used to designate that a function has no parameters as we saw in the `main` function. It can also be used to specify that a function has no return value. It can also be used to define a pointer to generic data as you will learn as you become a more experienced programmer.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Integral Type

The C language has three integral types: Boolean, character, and integer. Integral types cannot contain a fraction part; they are whole numbers.

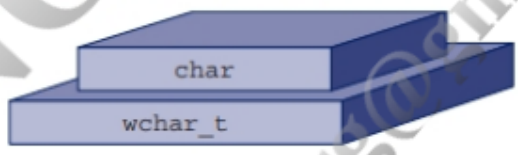
### Boolean

With the release of C99, the C language incorporated a Boolean type. Named after the French mathematician/philosopher George Boole, a Boolean type can represent only two values: `true` or `false`. Prior to C99, C used integers to represent the Boolean values: a nonzero number (positive or negative) was used to represent true, and zero was used to represent false. For backward compatibility, integers can still be used to represent Boolean values; however, it is recommended that new programs use the Boolean type. The Boolean type, which is referred to by the keyword `bool`, is stored in memory as 0 (`false`) or 1 (`true`).

### Character

The third type is character. Although characters are thought of as the letters of the alphabet, a computer has another definition. To a computer, a character is any value that can be represented in the computer's alphabet, or as it is better known, its character set. As illustrated in **Figure 2-8**, C standard provides two character types: `char` and `wchar_t`. The latter, `wchar_t`, is a wide character data type used to represent characters that require more memory than standard `char` data type.

**Figure 2-8** Character types




Most computers use the American Standard Code for Information Interchange (ASCII—pronounced "ask-key") alphabet. You do not need to memorize this alphabet as you did when you learned your natural languages; however, you will learn many of the special values by using them.

Most of the personal, mini-, and mainframe computers use 1 byte to store the `char` data types. A byte is 8 bits. With 8 bits, there are 256 different values in the `char` set. Although the size of `char` is machine dependent and varies from computer to computer, normally it is 1 byte, or 8 bits.

If you examine the ASCII code carefully, you will notice that there is a pattern to its alphabet that corresponds to the English alphabet. The first 32 ASCII characters and the last ASCII character are control characters. They are used to control physical devices, such as monitors and printers, and in telecommunication systems. The rest are characters that are used to compose words and sentences.

All the lowercase letters are grouped together, as are all the uppercase letters and the digits. Many of the special characters, such as the shift characters on the top row of the keyboard, are grouped together, but some are found spread throughout the alphabet.

What makes the letter *a* different from the letter *x*? In English, it is the visual formation of the graphic associated with the letter. In the computer, it is the underlying value of the bit configuration for the letter. The letter *a* is binary `0110 0001`. The letter *x* is `0111 1000`. The decimal values of these two binary numbers are 97 and 120, respectively.

To support non-English languages and languages that don't use the Roman alphabet, the C99 standard created the wide character type (`wchar_t`). Without going into all of the complexities, C supports two international standards, one for four-type characters and one for two-byte characters. Both of these standards support the traditional characters found in ASCII; that is, all extensions occur above the last ASCII character. The original ASCII characters are now known as the basic Latin character set. Generally speaking, the wide-character set is beyond the scope of an introductory programming text and is not covered in this text.

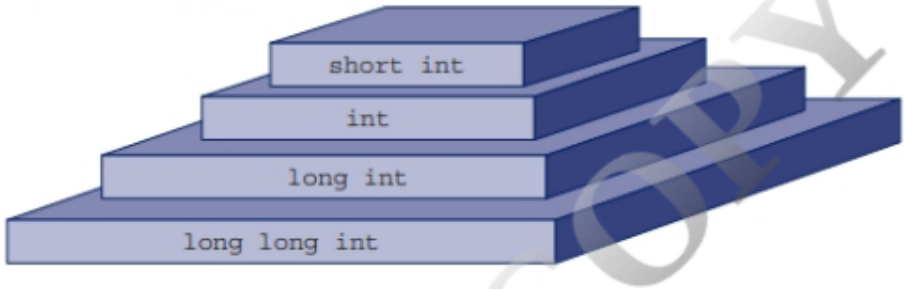
Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

### Integer

An integer type is a number without a fraction part. C supports four different sizes of the integer data type: `short int`, `int`, `long int`, and `long long int`. A `short int` can also be referred to as `short`, `long int` can be referred to as `long`, and `long long int` can be referred to as `long long`. C defines these data types so that they can be organized from the smallest to the largest, as shown in **Figure 2-9**. The type also defines the size of the field in which data can be stored. In C, this is true even though the size is machine dependent and varies from computer to computer.

**Figure 2-9** Integer types




If you need to know the size of any data type, C provides an operator, `sizeof`, that will tell you the exact size in bytes. Although the size is machine-dependent, C requires that the following relationship always be true:

```
sizeof (short) ≤ sizeof (int) ≤ sizeof (long) ≤ sizeof (long long)
```

Each integer size can be a signed or an unsigned integer. If the integer is `signed`, then one bit must be used for a signed (0 is plus, 1 is minus). The unsigned integer can store a positive number that is twice as large as the signed integer of the same size. For a complete discussion, see Appendix B, "Numbering Systems." **Table 2-3** contains typical values for the integer types. Recognize, however, that the actual sizes are dependent on the physical hardware.

**Table 2-3** Typical integer sizes and values for signed integers

| TYPE | BYTE SIZE | MINIMUM VALUE | MAXIMUM VALUE |
|---|---|---|---|
| `short int` | 2 | –32,768 | 32,767 |
| `int` | 4 | –2,147,483,648 | 2,147,483,647 |
| `long int` | 4 | –2,147,483,648 | 2,147,483,647 |
| `long long int` | 8 | –9,223,372,036,854,775,807 | 9,223,372,036,854,775,806 |

To provide flexibility across different hardware platforms, C has a library, `limits.h`, that contains size information about integers. For example, the minimum integer value for the computer is defined as `INT_MIN`, and the maximum value is defined as `INT_MAX`.

## Floating-Point Types

The C standard recognizes three floating-point types: real, imaginary, and complex. Like the limits library for integer values, there is a standard library, `float.h`, for the floating-point values. Unlike the integral type, real type values are always signed.

### Real

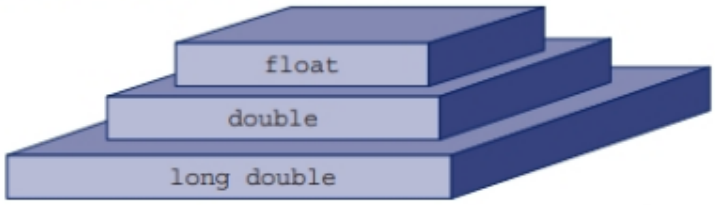
The real type holds values that consist of an integral and a fractional part, such as 43.32. The C language supports three different sizes of real types: `float`, `double`, and `long double`. As was the case for the integer type, real numbers are defined so that they can be organized from smallest to largest. The relationship among the real types is seen in **Figure 2-10**.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 2-10** Floating-point types




Regardless of machine size, C requires that the following relationship must be true:

```
sizeof (float) ≤ sizeof (double) ≤ sizeof (long double)
```

### Imaginary Type

C defines an imaginary type. An imaginary number is used extensively in mathematics and engineering. An imaginary number is a real number multiplied by the square root of –1 ($\sqrt{-1}$). The imaginary type, like the real type, can be of three different sizes: float imaginary, double imaginary, and long double imaginary.

Most C implementations do not support the imaginary type yet and the functions to handle them are not part of the standard. They are mentioned here because the imaginary type is one of the components of the complex type.

### Complex

C defines a complex type, which is implemented by most compilers. A complex number is a combination of a real and an imaginary number. The complex type, like the real type, can be of three different sizes: float complex, double complex, and long long complex. The size needs to be the same in both the real and the imaginary part. The following provide two program examples that use complex numbers at the end of this chapter.

## Type Summary

**Table 2-4** provides a summary of the four standard data type categories.

**Table 2-4** Type summary

| CATEGORY | TYPE | C IMPLEMENTATION |
|---|---|---|
| Void | Void | `void` |
| Integral | Boolean | `bool` |
| | Character | `char, wchar_t` |
| | Integer | `short int, int, long int, long long int` |
| Floating-Point | Real | `float, double, long double` |
| | Imaginary | `float imaginary, double imaginary, long double imaginary` |
| | Complex | `float complex, double complex, long double complex` |

# 2.5 Variables

Variables are named memory locations that have a type, such as integer or character, which is inherited from their type. The type determines the values that a variable may contain and the operations that may be used with its values.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

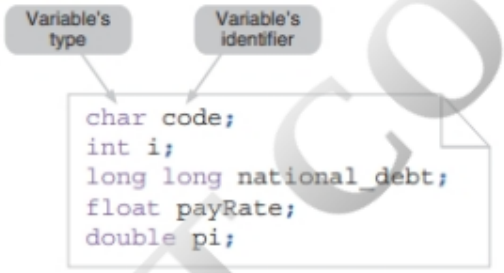
Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Variable Declaration

Each variable in your program must be declared and defined. In C, a declaration is used to name an object, such as a variable. Definitions are used to create the object. With one exception, a variable is declared and defined at the same time. The exception, as you will see later, declares them first and then defines them at a later time. For variables, definition assumes that the declaration has been done or is being done at the same time. While this distinction is somewhat of an oversimplification, it works in most situations.

When variables are created, the declaration gives them a symbolic name and the definition reserves memory for them. Once defined, variables are used to hold the data that are required by the program for its operation. Generally speaking, where the variable is located in memory is not a programmer's concern; it is a concern only of the compiler. From the programmer's perspective, the main concern is being able to access the data through their symbolic names, their identifiers. The concept of variables in memory is illustrated in **Figure 2-11**.

**Figure 2-11** Variables




A variable's type can be any of the data types, such as character, integer, or real. The one exception to this rule is the type void; a variable cannot be type void.

To create a variable, we first specify the type, which automatically specifies its size (precision), and then its identifier, as shown below in the definition of a real variable named `price` of type float.

```
float price;
```

**Table 2-5** shows some examples of variable declarations and definitions. As you study the variable identifiers, note the different styles used to make them readable. You should select a style and use it consistently. Most programmers prefer to use an uppercase letter to identify the beginning of each word after the first one.

**Table 2-5** Examples of variable declarations and definitions

| DECLARATION | DEFINITION | DESCRIPTION |
|---|---|---|
| `bool` | `fact;` | |
| `short` | `maxItems;` | `// Word separator: Capital` |
| `long` | `Long_national_debt;` | `// Word separator: underscore` |
| `float` | `payRate;` | `// Word separator: Capital` |
| `double` | `tax;` | |
| `float` | `complex voltage;` | |
| `char` | `code, kind;` | `// Poor style. It's better to define each in its own statement.` |
| `int` | `a, b;` | `// Poor style. See text. It's better to define each in its own statement.` |

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

C allows multiple variables of the same type to be defined in one statement. The last two entries in Table 2-5 use this format. Even though many professional programmers use it, this is considered poor programming style. It is much easier to find and work with variables if they are defined on separate lines. This makes the compiler work a little harder, but the resulting code is no different. This is one situation in which ease of reading the program and programmer efficiency are more important than the convenience of coding multiple declarations on the same line.

## Variable Initialization

We can initialize a variable at the same time we declare it by including an initializer. When present, the initializer establishes the first value that the variable will contain. To initialize a variable when it is defined, the identifier is followed by an equals sign (=), which is known as the assignment operator, and then the initializer, which is the variable's starting value when the function starts. This simple initialization format is shown below.

```
int count = 0;
```

Every time the function containing `count` is entered, `count` is set to zero. Now, what will be the result of the following initialization? Are both `count` and `sum` initialized or is only `sum` initialized?

```
int count, sum = 0;
```

The answer is that the initializer applies only to the variable defined immediately before it. Therefore, only sum is initialized! If you wanted both variables initialized, you would have to provide two initializers.

```
int count = 0, sum = 0;
```

Again, to avoid confusion and error, it is preferred to use only one variable definition per line. The preferred code in this case would be

```
int count = 0;
int sum = 0;
```

**Figure 2-12** repeats Figure 2-11, initializing the values in each of the variables.

**Figure 2-12** Variable initialization

```
char        code       = 'B';
int         i          = 14;
long long   natl_debt  = 1000000000000;
float       payRate    = 14.25;
double      pi         = 3.1415926536;
```

Program

```
B code
14  i
1000000000000  natl_debt
14.25 payRate
3.1415926536  pi
```

Memory

It is important to remember that, with a few exceptions that we will see later, variables are not initialized automatically. When variables are defined, they usually contain garbage (meaningless values left over from a previous use), so we need to initialize them or store data in them (using run-time statements) before accessing their values. Many compilers display a warning message when a variable is accessed before it is initialized.

**Note** | When a variable is defined, it is not initialized. It is best practice to initialize any variable requiring prescribed data when the function starts.

One final point about initializing variables when they are defined: Although the practice is convenient and saves you a line of code, it also can lead to errors. It is better, therefore, to initialize the variable with an assignment statement at the proper place in the body of the code. This may take another statement, but the efficiency of the resulting program is exactly the same, and you will make fewer errors in your code.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

At this point you might like to see what a more complex program looks like. As you read **Program 2-2**, note the blank lines that separate different groups of code. This is a good technique for making programs more readable. You should use blank lines in your programs the same way you use them to separate the paragraphs in a report.

## Program 2-2 | Print sum of three numbers

```
1  /* This program calculates and prints the sum of three numbers
2     input by the user at the keyboard.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7
8  int main (void)
9  {
10 // Local Declarations
11    int a;
12    int b;
13    int c;
14    int sum;
15
16 // Statements
17    printf("\nWelcome. This program adds\n");
18    printf("three numbers. Enter three numbers\n");
19    printf("in the form: nnn nnn nnn <return>\n");
20    scanf("%d %d %d", &a, &b, &c);
21
22    // Numbers are now in a, b, and c. Add them.
23    sum = a + b + c;
24
25    printf("The total is: %d\n\n", sum);
26
27    printf("Thank you. Have a good day.\n");
28    return 0;
29 } // main
```

*(continue)*

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Program 2-2 Print sum of three numbers *(continued)*

**Output**

```
Welcome. This program adds
three numbers. Enter three numbers
in the form: nnn nnn nnn <return>
11 22 33
The total is: 66
Thank you. Have a good day.
```

Study the style of this program carefully. First, note how it starts with a welcome message that tells the user exactly what needs to be entered. Similarly, at the end of the program, an ending message is printed. It is a good style to print a start and end message.

This program contains three different processes. First it reads three numbers. The code to read the numbers includes the printed instructions and a read (`scanf`) statement. The second process adds the three numbers. While this process consists of only a comment and one statement, it is separated from the read process. This makes it easier for the reader to follow the program. Finally, the results are printed. Again, the print process is separated from the calculate process by a blank line.

## 2.6 Constants

A constant is a data value that cannot be changed during the execution of a program. Like variables, constants have a type. In this section, Boolean, character, integer, real, complex, and string constants are discussed.

### Constant Representation

In this section, we show how to use symbols to represent constants. In the next section, you'll learn how to code constants.

#### Boolean Constants

A Boolean data type can take only two values: `true` and `false`. So it makes sense that only two symbols are used to represent a Boolean type: 0 (`false`) and 1 (`true`). Because we use the constant `true` or `false` in our program, we need to include the Boolean library, `stdbool.h`.

#### Character Constants

A character constant is enclosed between two single quotes (apostrophes). In addition to the character, a backslash (\) before the character is used. The backslash is known as the escape character. It is used when the character we need to represent does not have any graphic associated with it—that is, when it cannot be printed or when it cannot be entered from the keyboard. The escape character says that what follows is not the normal character but something else. For example, `'\n'` represents the newline character (line feed). So, even though there may be multiple symbols in the character constant, they always represent only one character.

Wide-character constants are coded by prefixing the constant with an `L`, as shown in the following example.

```
L'x'
```

The character in the character constant comes from the character set supplied by the hardware manufacturer. Most computers use the ASCII character set, or as it is sometimes called, the ASCII alphabet.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

C has named the critical character values so we can refer to them symbolically. Note that these control characters use the escape character followed by a symbolic character. They are shown in **Table 2-6**.

**Table 2-6** Symbolic names for control characters

| ASCII CHARACTER | SYMBOLIC NAME |
|---|---|
| null character | `'\0'` |
| alert (bell) | `'\a'` |
| backspace | `'\b'` |
| horizontal tab | `'\t'` |
| newline | `'\n'` |
| vertical tab | `'\v'` |
| form feed | `'\f'` |
| carriage return | `'\r'` |
| single quote | `'\' '` |
| double quote | `'\" '` |
| backslash | `'\\'` |

## Integer Constants

Although integers are always stored in their binary form, they are simply coded as they would be used in everyday life. Thus, the value 15 is simply coded as 15.

When you need to code the number as a series of digits, use the type signed integer, or signed long integer if the number is large. You can change this default signed by specifying unsigned (`u` or `U`), and `long` (`l` or `L`) or `long long` (`ll` or `LL`), after the number. The codes may be combined and may be coded in any order. Note that there is no way to specify a `short int` constant. When the suffix on a literal is omitted, it defaults to `int`. While both upper- and lowercase codes are allowed, it's best to use uppercase to avoid confusion (especially with the lowercase letter *l*, which often looks like the number 1). **Table 2-7** shows several examples of integer constants. The default types are typical for a personal computer.

**Table 2-7** Examples of integer constants

| REPRESENTATION | VALUE | DEFAULT TYPE |
|---|---|---|
| `+ 123` | 123 | `int` |
| `-378` | –378 | `int` |
| `-32271L` | –32,271 | `long int` |
| `76542LU` | 76,542 | `unsigned long int` |
| `12789845LL` | 12,789,845 | `long long int` |

## Real Constants

The default form for real constants is `double`. If you need the resulting data type to be `float` or `long double`, you must explicitly specify that in your code. As you might anticipate, `f` and `F` are used for `float` and `l` and `L` are used for `long double`. Again, do not use the lowercase `l` for `long double`; it is too easily confused with the number *1*.

**Table 2-8** shows several examples of real constants.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Table 2-8** Examples of real constants

| REPRESENTATION | VALUE | TYPE |
|---|---|---|
| `0.` | 0.0 | `double` |
| `.0` | 0.0 | `double` |
| `2.0` | 2.0 | `double` |
| `3.1416` | 3.1416 | `double` |
| `-2.0f` | –2.0 | `float` |
| `3.1415926536L` | 3.1415926536 | `long double` |

## Complex Constants

Complex constants are widely used in engineering. They are coded as two parts: the real part and the imaginary part, separated by a plus sign. The real part is coded using the real format rules. The imaginary part is coded as a real number times (*) the imaginary constant (`_Complex_I`). If the complex library (`complex.h`) is included, the imaginary constant can be abbreviated as `I`. **Table 2-9** shows several examples of complex constants. Note that programmers usually use the abbreviated form for the imaginary part.

**Table 2-9** Examples of complex constants

| REPRESENTATION | VALUE | TYPE |
|---|---|---|
| `12.3 + 14.4 * I` | $12.3 + 14.4 * (-1)^{1/2}$ | double complex |
| `14F + 16F * I` | $14 + 16 * (-1)^{1/2}$ | float complex |
| `1.4736L + 4.567561 * I` | $1.4736 + 4.56756 * (-1)^{1/2}$ | long double complex |

The default form for complex constants is `double`. If you need the resulting data type to be `float` or `long double`, you must specify that explicitly in your code. As you might anticipate, `f` and `F` are used for `float` and `l` and `L` are used for `long double`. Again, do not use the lowercase letter *l* for `long double`; it is too easily confused with the number *1*.

**Note** The two components of a complex constant must be of the same precision. That is, if the real part is type `double`, then the imaginary part must also be type `double`.

## String Constants

A string constant is a sequence of zero or more characters enclosed in double quotes. You used a string in your first program without even knowing that it was a string! Look at Program 2-1 to see if you can identify the string.

Listed in **Figure 2-13** are several strings, including the one from Program 2-1. The first example, an empty string, is simply two double quotes in succession. The second example, a string containing only the letter h, differs from a character constant in that it is enclosed in double quotes. In C, strings are treated differently. This means there's a big difference between storing `h` as a character and as a string. The last example in Figure 2-13 is a string that uses wide characters.

**Figure 2-13** Some strings

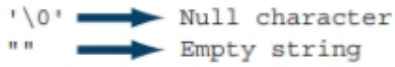
```
""                          // A null string
"h"
"Hello World\n"
"HOW ARE YOU"
"Good Morning!"
L"This string contains wide characters."
```

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

It is important to understand the difference between the null character (see Table 2-6) and an empty string. The null character represents no value. As a character, it is 8 zero bits. An empty string, on the other hand, is a string containing nothing. **Figure 2-14** shows the difference between these two constant types.

**Figure 2-14** Null characters and null strings




At this point, this is all you need to know about strings. As you become a more experienced programmer, you will learn how they are stored in a computer.

**Note** | Use single quotes for character constants. Use double quotes for string constants.

## Coding Constants

This section presents three different ways to code constants in programs: literal constants, defined constants, and memory constants.

### Literal Constants

A literal is an unnamed constant used to specify data. If we know that the data cannot be changed, then we can simply code the data value itself in a statement.

Literals are coded as part of a statement using the constant formats described in the previous section. For example, the literal 5 is used in the following statement.

```
a = b + 5;
```

### Defined Constants

Another way to designate a constant is to use the preprocessor command `define`. Like all preprocessor commands, it is prefaced with the pound sign (#). The `define` directive is usually placed at the beginning of the program, although it is legal anywhere. Placing them at the beginning of the program makes them easy to find and change. A typical `define` command might be

```
#define SALES_TAX_RATE .0825
```

Note that the preceding example contains a sales tax rate, which can change often. It's a good idea to place constants that are likely to change at the beginning of the program, so you can easily find them when you need to change them.

As the preprocessor reformats the program for the language translator, it replaces each defined name, `SALES_TAX_RATE` in the previous example with its value (.0825) wherever it is found in the source program. This action is just like the search-and-replace command found in a text editor. The preprocessor does not evaluate the code in any way—it just blindly makes the substitution. For a complete discussion of defined constants, see Appendix C, "Preprocessor Commands."

### Memory Constants

The third way to use a constant is with memory constants. A memory constant uses a C type qualifier, `const`, to indicate that the data cannot be changed. Its format is:

```
const type identifier = value;
```

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

As you have seen, a variable definition does nothing more than give a type and size to a named object in memory. Now assume that you want to fix the contents of this memory location so that they cannot be changed. This is the same concept as a literal, only now you give it a name. The following code creates a memory constant, `cPi`. To help you remember that it is a constant, you preface the identifier name with c.

```
const float cPi = 3.14159;
```

Three points merit discussion: (1) The type qualifier comes first. (2) Then there must be an initializer. If the constant is not initialized, then it would take whatever happened to be in memory at `cPi`'s location when the program starts. (3) Finally, since `cPi` is a constant, it cannot be changed.

**Program 2-3** demonstrates the three different ways to code `PI` as a constant.

## Program 2-3 | Memory constants

```
1  /* This program demonstrates three ways to use constants.
2
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7  #define PI 3.1415926536
8
9  int main (void)
10 {
11 // Local Declarations
12    const double cPi = PI;
13
14 // Statements
15    printf("Defined constant PI: %f\n", PI);
16    printf("Memory constant cPi: %f\n", cPi);
17    printf("Literal constant: %f\n", 3.1415926536);
18    return 0;
19 } // main
```

**Output**

```
Defined constant PI: 3.141593
Memory constant cPi: 3.141593
Literal constant:  3.141593
```

## 2.7 Input/Output

Although previous programs have implicitly shown how to print messages, they have not formally discussed how to use C facilities to input and output data. As you become a more experienced programmer, you will learn more about C input/output facilities and how to use them. In this section, we describe simple input and output formatting.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Streams

In C, data is input to and output from a stream. A stream is a source of, or destination for, data. It is associated with a physical device, such as a terminal, or with a file stored in auxiliary memory.

C uses two forms of streams: text and binary. A text stream consists of a sequence of characters divided into lines with each line terminated by a new-line (`\n`). A binary stream consists of a sequence of data values such as integer, real, or complex using their memory representation.

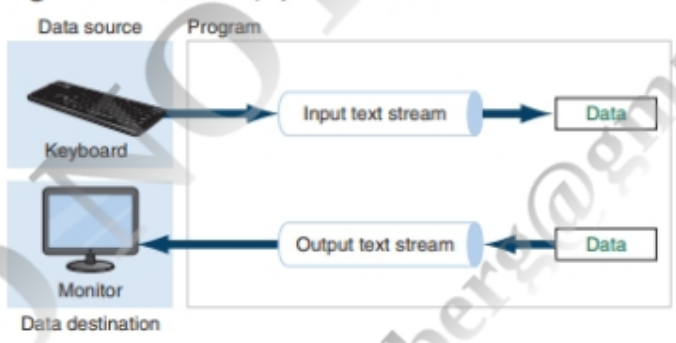
A terminal can be associated only with a text stream because a keyboard can only send a stream of characters into a program and a monitor can only display a sequence of characters. A file, on the other hand, can be associated with a text or binary stream. Data is stored in a file and later retrieved as a sequence of characters (text stream) or as a sequence of data values (binary stream).

In this chapter, it is assumed that the source of data is the keyboard and the destination of data is the monitor. In other words, the terminal devices produce or consume text streams. In C, the keyboard is known as standard input and the monitor is known as standard output.

**Note** | A terminal keyboard and monitor can be associated only with a text stream. A keyboard is a source for a text stream; a monitor is a destination for a text stream.

**Figure 2-15** illustrates the concept of streams and the two physical devices associated with input and output text streams.

**Figure 2-15** Stream physical devices




## Formatting Input/Output

The previous section discussed the terminal as a text stream source and destination. The program can only receive text streams from a terminal (keyboard) and send text streams to a terminal (monitor). However, these text streams often represent different data types, such as integer, real, and Boolean. The C language provides two formatting functions: `printf` for output formatting and `scanf` for input formatting. The `printf` function converts data stored in the program into a text stream for output to the monitor; the `scanf` function converts the text stream coming from the keyboard to data values and stores them in program variables. In other words, the `printf` and `scanf` functions are data to text stream and text stream to data converters.

### Output Formatting: `printf`

The output formatting function is `printf`. The `printf` function takes a set of data values, converts them to a text stream using formatting instructions contained in a format control string, and sends the resulting text stream to the standard output (monitor). For example, an integer 234 stored in the program is converted to a text stream of three numeric ASCII characters (2, 3, and 4) and then is sent to the monitor. What is seen on the monitor is these three characters, not the integer 234. However, we interpret the three characters together as an integer value. **Figure 2-16** illustrates this concept.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**Figure 2-16** Output formatting concept


## Basic Concept

The `printf` function uses an interesting design to convert data into text streams. Here we describe how the text stream should be formatted using a format control string containing zero or more conversion specifications. In addition to the conversion specifications, the control string may contain textual data and control characters to be displayed.

Each data value to be formatted into the text stream is described as a separate conversion specification in the control string. The specifications describe the data values' type, size, and specific format information, such as how wide the display width should be. The location of the conversion specification within the format control string determines its position within the text stream.

The control string and data values are passed to the print function (`printf`) as parameters, the control string as the first parameter and one parameter for each value to be printed. In other words, the following information is supplied to the `print` function:

1. The format control string, including any textual data to be inserted into the text stream.
2. A set of zero or more data values to be formatted.

**Figure 2-17** is a conceptional representation of the format control string and two conversion specifications.

**Figure 2-17** Output stream formatting example

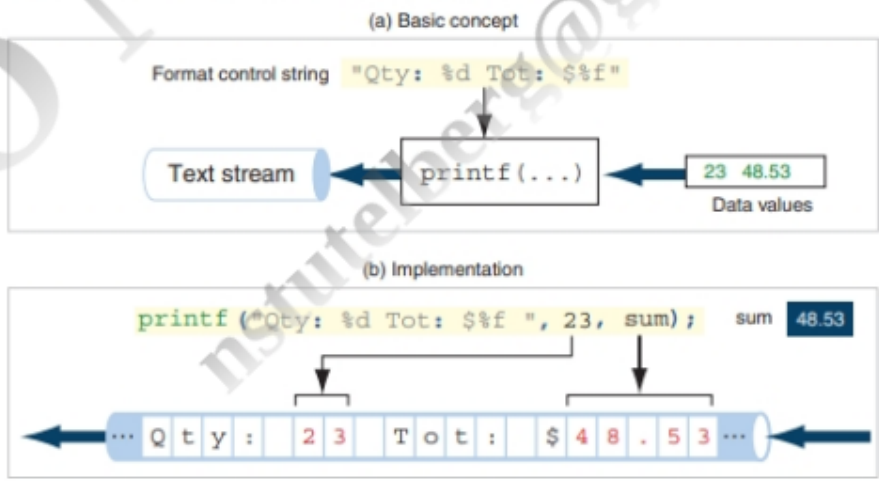



Figure 2-17(a) shows the format string and the data values as parameters for the print function. Within the control string quantity (`Qty:`) and total (`Tot:`) are specified as textual data and two conversion specifications (`%d` and `%f`). The first specification requires an integer type value; the second requires a real type value. The conversion specifications are discussed in detail in the following section.

Figure 2-17(b) shows the formatting operation and the resulting text stream. The first data value is a literal integer; the second data value is the contents of a variable named `tot`. This part of Figure 2-17 shows how the `printf` function expands the control stream and inserts the data values and text characters.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

### Format Control String Text

The control string may also contain text to be printed, such as instructions to the user, captions or other identifiers, and other text intended to make the output more readable. In fact, as you have already seen, the format string may contain nothing but text, in which case the text will be printed exactly as shown. You employed this concept in the greeting program. In addition, you can printcontrol characters, such as tabs (`\t`), newlines (`\n`), and alerts (`\a`), by including them in the format string. Tabs are used to format the output into columns. Newlines terminate the current line and continue formatting on the next line. Alerts sound an audio signal, usually to alert the user to a condition that needs attention. These control characters were listed earlier in Table 2-6.

### Conversion Specification

To insert data into the stream, we use a conversion specification that contains a start token (`%`), a conversion code, and up to four optional modifiers as shown in **Figure 2-18**. Only the field-specification token (`%`) and the conversion code are required.

**Figure 2-18** Conversion specification


Approximately 30 different conversion codes are used to describe data types. For now, however, we are concerned with only three: character (`c`), integer (`d`), and floating point (`f`). These codes, with some examples, are shown in **Table 2-10**. Note that one item in this table, Size, is discussed in the next section.

**Table 2-10** Format codes for output

| TYPE | SIZE | CODE | EXAMPLE |
|---|---|---|---|
| `Char` | None | `c` | `%c` |
| `short int` | `h` | `d` | `%hd` |
| `Int` | None | `d` | `%d` |
| `long int` | None | `d` | `%ld` |
| `long long int` | `ll` | `d` | `%lld` |
| `Float` | None | `f` | `%f` |
| `Double` | None | `f` | `%f` |
| `long double` | `L` | `f` | `%Lf` |

The size modifier is used to modify the type specified by the conversion code. There are four different sizes: `h`, `l` (`el`), `ll` (`el el`), and `L`. The `h`, used with the integer codes to indicate a short integer value, is a carry-over from assembler language where it meant "half word." The `l` is used to indicate a long integer value; the `ll` is used to indicate a long long integer value; and the `L` is used with floating-point numbers to indicate a long double value.

A width modifier may be used to specify the minimum number of positions in the output. (If the data require using more space than allowed, then `printf` overrides the width.) It is very useful to align output in columns, such as when we need to print a column of numbers. If a width modifier is not used, each output value will take just enough room for the data.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

If a floating-point number is being printed, then the number of decimal places to be printed with the precision modifier must be specified. The precision modifier has the format

```
.m
```

where `m` is the number of decimal digits. If no precision is specified, `printf` prints six decimal positions. These six decimal positions are often more than is necessary.

When both width and precision are used, the width must be large enough to contain the integral value of the number, the decimal point, and the number of digits in the decimal position. Thus, a conversion specification of %7.2f is designed to print a maximum value of 9999.99. Some examples of width specifications and precision are shown below.

```
%2hd    // short integer—2 print positions
%4d     // integer—4 print positions
%81d    // long int—8 (not 81) positions
%7.2f   // float—7 print positions: nnnn.dd
%10.3Lf // long double—10 positions: nnnnnn.ddd
```

The flag modifier is used for four print modifications: justification, padding, sign, and numeric conversion variants. The first three are discussed here. You'll learn about the conversion variants as you gain more experience as a programmer.

Justification controls the placement of a value when it is shorter than the specified width. Justification can be left or right. If there is no flag and the defined width is larger than required, the value is right-justified. The default is right justification. To left justify a value, the flag is set to minus (-).

Padding defines the character that fills the unused space when the value is smaller than the print width. It can be a space, the default, or zero. If there is no flag defined for padding, the unused width is filled with spaces; if the flag is 0, the unused width is filled with zeroes. Note that the zero flag is ignored if it is used with left justification because adding zeros after a number changes its value.

The sign flag defines the use or absence of a sign in a numeric value. Three formats can be specified: default formatting, print signed values, or prefix positive values with a leading space. Default formatting inserts a sign only when the value is negative. Positive values are formatted without a sign. When the flag is set to a plus (+), signs are printed for both positive and negative values. If the flag is a space, then positive numbers are printed with a leading space and negative numbers with a minus sign.

**Table 2-11** documents three of the more common flag options.

**Table 2-11** Flag formatting options

| FLAG TYPE | FLAG CODE | FORMATTING |
|---|---|---|
| Justification | None | right justified |
| | - | left justified |
| Padding | None | space padding |
| | 0 | zero padding |
| Sign | None | positive value: no sign |
| | | negative value: - |
| | + | positive value: + |
| | | negative value: - |
| | None | positive value: space |
| | | negative value: - |

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

### Output Examples

This section contains several output examples. Here are the `printf` statements, followed by what would be printed. Cover up the solution and try to predict the results.

#### Example: `printf` output 1

```
printf ("%d%c%f", 23, 'z', 4.1);
```

**Output**

```
23z4.100000
```

Note that because there are no spaces between the conversion specifications, the data are formatted without spaces between the values.

#### Example: `printf` output 2

```
printf("%d %c %f", 23, 'z', 4.1);
```

**Output**

```
23 z 4.100000
```

This is a repeat of `printf` output 1 with spaces between the conversion specifications.

#### Example: `printf` output 3

```
int num1 = 23;
char zee = 'z';
float num2 = 4.1;
printf("%d %c %f", num1, zee, num2);
```

**Output**

```
23 z 4.100000
```

Again, the same example, this time using variables.

#### Example: `printf` output 4

```
printf("%d\t%c\t%5.1f\n", 23, 'Z', 14.2);
printf("%d\t%c\t%5.1f\n", 107, 'A', 53.6);
printf("%d\t%c\t%5.1f\n", 1754, 'F', 122.0);
printf("%d\t%c\t%5.1f\n", 3, 'P', 0.1);
```

**Output**

```
23      Z       14.2
107     A       53.6
1754    F       122.0
3       P       0.1
```

In addition to the conversion specifications, note the tab character (`\t`) between the first and second, and second and third conversion specifications. Because the data are to be printed in separate lines, each format string ends with a newline (`\n`).

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

### Example: printf output 5

```
printf("The number%dis my favorite number.", 23);
```

**Output**

```
The number23is my favorite number.
```

Since there are no spaces before and after the format code (`%d`), the number `23` is run together with the text before and after.

### Example: printf output 6

```
printf("The number%7d my favorite number.", 23);
```

**Output**

```
The number is      23
```

If you count the spaces carefully, you will note that five spaces follow the word is. The first space comes from the space after `is` and before the `%` in the format string. The other four come from the width in the conversion specification.

### Example: printf output 7

```
printf("The tax is %6.2f this year.", 233.12);
```

**Output**

```
The tax is 233.12 this year.
```

In this example, the width is six and the precision two. Because the number of digits printed totals five (three for the integral portion and two for the decimal portion), and the decimal point takes one print position, the full width is filled with data. The only spaces are the spaces before and after the conversion code in the format string.

### Example: printf output 8

```
printf("The tax is %8.2f this year.", 233.12);
```

**Output**

```
The tax is 233.12 this year.
```

### Example: printf output 9

```
printf("The tax is %08.2f this year.", 233.12);
```

**Output**

```
The tax is 00233.12 this year.
```

This example uses the zero flag to print leading zeros. Note that the width is eight positions. Three of these positions are taken up by the precision of two digits and the decimal point. This leaves five positions for the integral portion of the number. Because there are only three digits (`233`), `printf` inserts two leading zeros.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Example: printf output 10

```
printf("\"%8c %d\"", "       h 23", 23);
```

**Output**

```
"                   h            23"
```

In this example, we want to print the data within quotes. Because quotes are used to identify the format string, we can't use them as print characters. To print them, therefore, we must use the escape character with the quote (\"), which tells printf that what follows is not the end of the string but a character to be printed, in this case, a quote mark.

## Example: printf output 11

```
printf ("This line disappears.\r...A new line\n");
printf ("This is the bell character \a\n");
printf ("A null character\0kills the rest of the line\n");
printf ("\nThis is \'it\' in single quotes\n");
printf ("This is \"it\" in double quotes\n");
printf ("This is \\ the escape character itself\n");
```

**Output**

```
...A new line
This is the bell character
A null character
This is 'it' in single quotes
This is "it" in double quotes
This is \ the escape character itself
```

These examples use some of the control character names found in Table 2-6. Two of them give unexpected results. In example 11, the return character (\r) repositions the output at the beginning of the current line without advancing the line. Therefore, all data that were placed in the output stream are erased.

The null character effectively kills the rest of the line. If a newline character (\n) was not used at the beginning of the next line, it would have started immediately after character.

## Example: printf output 12

New example with multiple flags.

```
printf("|%-+8.2f| |%0+8.2f| |%-0+8.2f|", 1.2, 2.3, 3.4);
```

**Output**

```
|+1.20 | |+0002.30| |+3.40 |
```

This example uses multiple flags. As shown in the output, each value is enclosed in vertical bars. The first value is printed left justified with the positive flag set. The second example uses zero fill with a space for the sign. Note that there is a leading space in the output. This represents the plus value. It is then followed by the leading zeros. The last example demonstrates that the zero fill is ignored when a numeric value is printed with left justification.

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

### Common Output Errors

Each of the following examples has at least one error. Try to find each one before you look at the output. Your results may vary depending on your compiler and hardware.

#### Example: `printf` output error 1

```
printf ("%d %d %d\n", 44, 55);
```

**Output**

```
44 55 0
```

This example has three conversion specifications but only two values.

#### Example: `printf` output error 2

```
printf ("%d %d\n", 44, 55, 66);
```

**Output**

```
44 55
```

This example has two conversion specifications with three values. In this case, `printf` ignores the third value.

#### Example: `printf` output error 3

```
float x = 123.45;
printf("The data are: %d\n", x);
```

**Output**

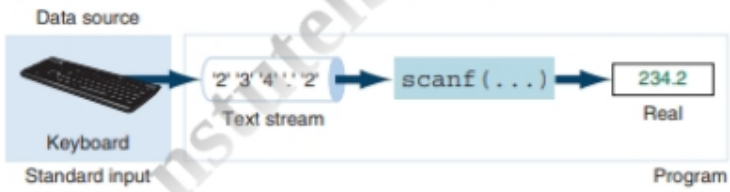
```
The data are: 1079958732
```

This is a very common error in which the format specification (integer) does not match the data type (real).

### Input Formatting: `scanf`

The standard input formatting function in C is `scanf` (scan formatting). This function takes a text stream from the keyboard, extracts and formats data from the stream according to a format control string, and then stores the data in specified program variables. For example, the stream of 5 characters `'2'`, `'3'`, `'4'`, `'.'`, and `'2'` are extracted as the real `234.2`. **Figure 2-19** shows the concept.

**Figure 2-19** Formatting text from an input stream




The `scanf` function is the reverse of the `printf` function. The following is a list of guidelines for using `scanf` function.

1. A format control string describes the data to be extracted from the stream and reformatted.
2. Rather than data values as in the `printf` function, `scanf` requires the variable addresses where the pieces of data are to be stored. Unlike the `printf` function, the destination of the data items cannot be literal values, they must store in the variables.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY
nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

3. With the exception of the character specification, leading whitespaces are discarded.
4. Any nonconversion specification characters in the format string must be exactly matched by the next characters in the input stream.

We must be careful about extra characters in the control stream. Extra characters in the control string can be divided into two categories: nonwhitespace and whitespace.

Nonwhitespace characters in the control string must exactly match characters entered by the user and are discarded by the `scanf` after they are read. If they don't match, then `scanf` goes into an error state and the program must be manually terminated.

It is recommended that you don't use nonwhitespace characters in the format string, at least until you learn how to recover from errors. However, there are some uses for them. For example, if the users want to enter dates with slashes, such as 5/10/06, the slashes must either be read and discarded using the character format specification (see the discussion of the assignment suppression flag in the later section, "Conversion Specification") or coded as nonwhitespace in the format specification. Reading and discarding them is preferred.

Whitespace characters in the format string are matched by zero or more whitespace characters in the input stream and discarded. There are two exceptions to this rule: the character conversion code and the scan set (see Chapter 11) do not discard whitespace. It is easy, however, to manually discard whitespace characters when necessary to read a character. Simply code a space before the conversion specification, or between the two parts of the conversion specification, as shown below. Either one works.

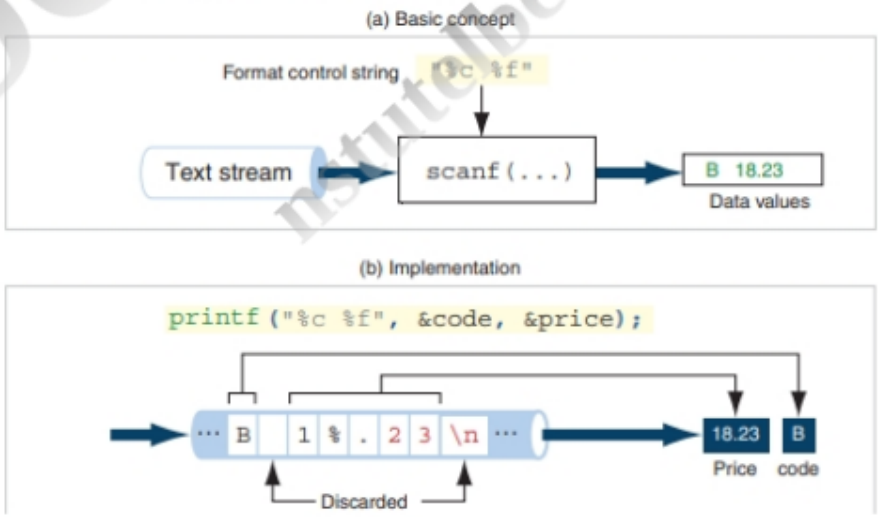
```
" %c" or "% c"
```

Remember that whenever you read data from the keyboard, there is a return character from a previous read. If you don't flush the whitespace characters when you read a character, therefore, you will get the whitespace from the previous read. To read a character, you should always code at least one whitespace character in the conversion specification. Otherwise the whitespace remaining in the input stream is read as the input character. For example, to read three characters, you should code the following format string. Note the spaces before each conversion specification.

```
scanf(" %c %c %d", &c1, &c2, &c3);
```

**Figure 2-20** demonstrates the input format string concept with a control string having two fields (%d and %f). The first one defines that a character will be inserted here; the second defines that a real will be inserted there. We will discuss these place holders, or format specifiers, later in the chapter.

**Figure 2-20** Input stream formatting example




Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

### Format Control String

Like the control string for `printf`, the control string for `scanf` is enclosed in a set of quotation marks and contains one or more conversion specifications that describe the data types and indicate any special formatting rules and/or characters.

### Conversion Specification

To format data from the input stream, we use a conversion specification that contains a start token (`%`), a conversion code, and up to three optional modifiers as shown in **Figure 2-21**. Only the field-specification token (`%`) and the conversion code are required.

**Figure 2-21** Conversion specification


There are only three differences between the conversion codes for input formatting and output formatting. First, there is no precision in an input conversion specification. It is an error to include a precision; if `scanf` finds a precision it stops processing and the input stream is in the error state.

There is only one flag for input formatting, the assignment suppression flag (`*`). More commonly associated with text files, the assignment suppression flag tells `scanf` that the next input field is to be read but not stored. It is discarded. The following `scanf` statement reads an integer, a character, and a floating-point number from the input stream. The character is read and discarded. The other fields are read, formatted, and stored. Note that there is no matching address parameter for the data to be discarded.

```
scanf ("%d %*c %f", &x, &y);
```

The third difference is the width specification; with input formatting it is a maximum, not a minimum, width. The width modifier specifies the maximum number of characters that are to be read for one format code. When a width specification is included, therefore, `scanf` reads until the maximum number of characters have been processed or until `scanf` finds a whitespace character. If `scanf` finds a whitespace character before the maximum is reached, it stops.

### Input Parameters

For every conversion specification there must be a matching variable in the address list. The address list contains the address of the matching variable. How do we specify an address? It's quite simple: Addresses are indicated by prefixing the variable name with an ampersand (`&`). In C, the ampersand is known as the address operator. Using the address operator, if the variable name is price, then the address is `&price`. Forgetting the ampersand is one of the most common errors for beginning C programmers, so you will have to concentrate on it when you use the `scanf` function. Note that `scanf` requires variable addresses in the address list.

Remember that the first conversion specification matches the first variable address, the second conversion specification matches the second variable address, and so on. This correspondence is very important. It is also very important that the variable's type match the conversion specification type. The C compiler does not verify that they match. If they don't, the input data will not be properly formatted when they are stored in the variable.

### End of File and Errors

In addition to whitespace and width specifications, two other events stop the `scanf` function. If the user signals that there is no more input by keying end of file (EOF), then `scanf` terminates the input process. While there is no EOF on the keyboard, it can be simulated in most systems. For example, Windows uses the `<ctrl + z>` key combination to signal EOF. Unix and Apple Macintosh use `<ctrl + d>` for EOF. The C user's manual for your system should specify the key sequence for EOF.

Second, if `scanf` encounters an invalid character when it is trying to convert the input to the stored data type, it stops. The most common error is finding a nonnumeric character when it is trying to read a number. The valid characters

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

are leading plus or minus, digits, and one decimal point. Any other combination, including any alphabetic characters, will cause an error. Although it is possible to detect this error and ask the user to re-input the data, you won't learn about the logic for detecting errors until you become a more experienced programmer. Until then, full caution should be taken when entering data into your program.

### Input Formatting Summary

The following list summarizes the rules for using `scanf`.

1. The conversion operation processes until:
   a. End of file is reached.
   b. The maximum number of characters has been processed.
   c. A whitespace character is found after a digit in a numeric specification.
   d. An error is detected.
2. There must be a conversion specification for each variable to be read.
3. There must be a variable address of the proper type for each conversion specification.
4. Any character in the format string other than whitespace or a conversion specification must be exactly matched by the user during input. If the input stream does not match the character specified, an error is signaled and `scanf` stops.
5. It is a fatal error to end the format string with a whitespace character. Your program will not run correctly if you do.

### Input Examples

This section contains several examples. We list the data that will be input first. This allows you to cover up the function and try to formulate your own `scanf` statement.

1. `214 156 14Z`

   ```
   scanf("%d%d%d%c", &a, &b, &c, &d);
   ```

   Note that if there were a space between the 14 and the Z, it would create an error because `%c` does not skip whitespace! To prevent this problem, put a space before the `%c` code as shown below. This will cause it to skip leading whitespace.

   ```
   scanf("%d%d%d %c", &a, &b, &c, &d);
   ```

2. `2314 15 2.14`

   ```
   scanf("%d %d %f", &a, &b, &c);
   ```

   Note the whitespace between the conversion specifications. These spaces are not necessary with numeric input, but it is a good idea to include them.

3. `14/26 25/66`

   ```
   scanf("%2d/%2d %2d/%2d", &num1, &den1, &num2, &den2);
   ```

   Note the slashes (/) in the format string. Because they are not a part of the conversion specification, the user must enter them exactly as shown or `scanf` will stop reading.

4. `11-25-56`

   ```
   scanf ("%d-%d-%d", &a, &b, &c);
   ```

   Again, we see some required user input, this time dashes between the month, day, and year. While this is a common date format, it can cause problems. A better solution would be to prompt the user separately for the month, the day, and the year.

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

### Common Input Errors

Each of the following examples has at least one error. Try to find it before you look at the solution. Your results may vary depending on your compiler and hardware.

1. 
```
int a = 0;
scanf ("%d", a);
printf("%d\n", a);
```

**Input:** 234
**Output:** 0

This example has no address token on the variable (&a). If the program runs at all, the data are read into an unidentified area in memory. What is printed is the original contents of the variable, in this case 0.

2. 
```
float a = 2.1;
scanf ("%5.2f", &a);
printf ("%5.2f", a);
```

**Input:** 74.35
**Output:** 2.10

This example has no precision in the input conversion specification. When scanf finds a precision, it stops processing and returns to the function that called it. The input variable is unchanged.

3. 
```
int a;
int b;
scanf ("%d%d%d", &a, &b);
printf ("%d %d\n", a, b);
```

**Input:** 5 10
**Output:** 5 10

This example has three conversion specifications but only two addresses. Therefore, scanf reads the first two values and quits because no third address if found.

4. 
```
int a = 1;
int b = 2;
int c = 3;
scanf ("%d%d", &a, &b, &c);
printf ("%d %d %d\n", a, b, c);
```

**Input:** 5 10 15
**Output:** 5 10 3

This example has only two conversion specifications, but it has three addresses. Therefore, scanf reads the first two values and ignores the third address. The value 15 is still in the input stream waiting to be read.

## Programming Example: Working with Input and Output

In this section, we show some programming examples to emphasize the ideas and concepts we have discussed about input/output.

**Program 2-4** is a very simple program that prints "Nothing!"

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Program 2-4 A Program that prints "nothing!"

```
1  /* Prints the message "Nothing!".
2     Written by:
3     Date:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9  // Statements
10    printf("This program prints\n\n\t\"Nothing!\"");
11    return 0;
12 } // main
```

**Output**

```
This program prints
   "Nothing !"
```

**Program 2-5** demonstrates printing Boolean values. As the program shows, however, while a Boolean literal contains either `true` or `false`, when it is printed, it is printed as `0` or `1`. This is because there is no conversion code for Boolean. To print it, you must use the integer type, which prints its stored value, `0` or `1`.

## Program 2-5 Demonstrate printing Boolean constants

```
1  /* Demonstrate printing Boolean constants.
2     Written by:
3     Date:
4  */
5  #include <stdio.h>
6  #include <stdbool.h>
7
8  int main (void)
9  {
10 // Local Declarations
11    bool x = true;
12    bool y = false;
13
```

*(continue)*

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Program 2-5 Demonstrate printing Boolean constants *(continued)*

```
14 // Statements
15    printf ("The Boolean values are: %d %d\n", x, y);
16    return 0;
17 } // main
```

**Output**

```
The Boolean values are: 1 0
```

**Program 2-6** demonstrates that all characters are stored in the computer as integers. We define some character variables and initialize them with values, and then we print them as integers. As you study the output, note that the ASCII values of the characters are printed. The program also shows the value of some nonprintable characters.

## Program 2-6 | Print value of selected characters

```
1  /* Display the decimal value of selected characters,
2     Written by:
3     Date:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9  // Local Declarations
10    char A   = 'A';
11    char a   = 'a';
12    char B   = 'B';
13    char b   = 'b';
14    char Zed    = 'Z';
15    char zed    = 'z';
16    char zero   = '0';
17    char eight  = '8';
18    char NL = '\n'; // newline
19    char HT = '\t'; // horizontal tab
20    char VT = '\v'; // vertical tab
21    char SP = ' ';  // blank or space
22    char BEL    = '\a';    // alert (bell)
23    char dblQuote   = '"';      // double quote
24    char backslash  = '\\'; // backslash itself
25    char oneQuote   = '\'';    // single quote itself
```

*(continue)*

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Program 2-6 Print value of selected characters *(continued)*

```
26
27 // Statements
28    printf("ASCII for char 'A' is: %d\n", A);
29    printf("ASCII for char 'a' is: %d\n", a);
30    printf("ASCII for char 'B' is: %d\n", B);
31    printf("ASCII for char 'b' is: %d\n", b);
32    printf("ASCII for char 'Z' is: %d\n", Zed);
33    printf("ASCII for char 'z' is: %d\n", zed);
34    printf("ASCII for char '0' is: %d\n", zero);
35    printf("ASCII for char '8' is: %d\n", eight);
36    printf("ASCII for char '\\n' is: %d\n", NL);
37    printf("ASCII for char '\\t' is: %d\n", HT);
38    printf("ASCII for char '\\v' is: %d\n", VT);
39    printf("ASCII for char ' ' is: %d\n", SP);
40    printf("ASCII for char '\\a' is: %d\n", BEL);
41    printf("ASCII for char '\"' is: %d\n", dblQuote);
42    printf("ASCII for char '\\' is: %d\n", backslash);
43    printf("ASCII for char '\'' is: %d\n", oneQuote);
44
45    return 0;
46 } // main
```

**Output**

```
ASCII for character 'A' is: 65
ASCII for character 'a' is: 97
ASCII for character 'B' is: 66
ASCII for character 'b' is: 98
ASCII for character 'Z' is: 90
ASCII for character 'z' is: 122
ASCII for character '0' is: 48
ASCII for character '8' is: 56
ASCII for character '\n' is: 10
ASCII for character '\t' is: 9
ASCII for is: 11
ASCII for character ' ' is: 32
ASCII for character '\a' is: 7
ASCII for character '"' is: 34
ASCII for character '\' is: 92
ASCII for character ' ' ' is: 39
```

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The following is a program that calculates the area and circumference of a circle using a preprocessor-defined constant for π. Although we haven't shown you how to make calculations in C, if you know algebra you will have no problem reading the code in **Program 2-7**.

## Program 2-7 | Calculate a circle's area and circumference

```
1  /* This program calculates the area and circumference of a
2     circle using PI as a defined constant.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7  #define PI 3.1416
8
9  int main (void)
10 {
11 // Local Declarations
12    float circ;
13    float area;
14    float radius;
15
16 // Statements
17    printf("\nPlease enter the value of the radius: ");
18    scanf("%f", &radius);
19
20    circ = 2 * PI  * radius;
21    area = PI * radius * radius;
22
23    printf("\nRadius is :  %10.2f", radius);
24    printf("\nCircumference is : %10.2f", circ);
25    printf("\nArea is :   %10.2f", area);
26
27    return 0;
28 } // main
```

**Output**

```
Please enter the value of the radius: 23
Radius is :   23.00
Circumference is :  144.51
Area is :   1661.91
```

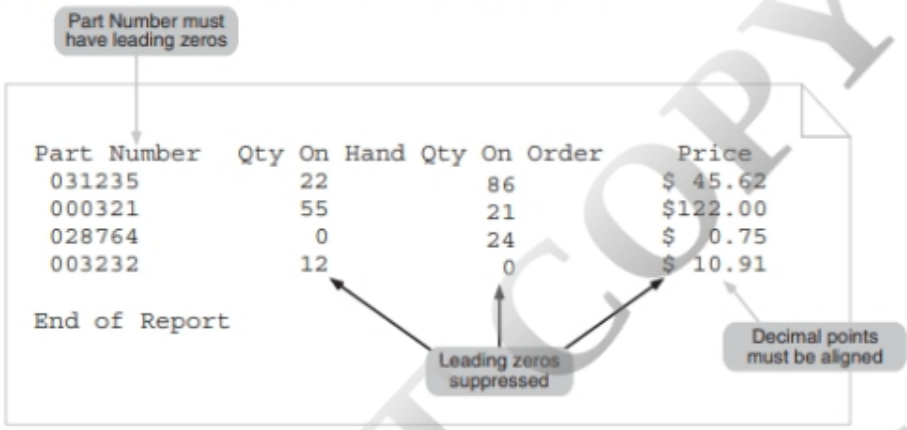
DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

You are assigned to a new project that is currently being designed. To give the customer an idea of what a proposed report might look like, the project leader has asked you to write a small program to print a sample. The specifications for the report are shown in **Figure 2-22**, and the code is shown in **Program 2-8**.

**Figure 2-22** Output specifications for inventory report




The report contains four fields: a part number, which must be printed with leading zeros; the current quantity on hand; the current quantity on order; and the price of the item, printed to two decimal points. All data should be aligned in columns with captions indicating the type of data in each column. The report should be closed with an "End of Report" message.

## Program 2-8 A Sample inventory report

```
1  /* This program will print four lines of inventory data on an
2     inventory report to give the user an idea of what a new report
3     will look like. Because this is not a real report, no input is
4     required. The data are all specified as constants
5
6     Written by:
7     Date:
8  */
9  #include <stdio.h>
10
11 int main (void)
12 {
13 // Statements
14    // Print captions
15    printf("\tPart Number\tQty On Hand");
16    printf("\tQty On Order\tPrice\n");
17
```

*(continue)*

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Program 2-8 A Sample inventory report *(continued)*

```
18     // Print data
19     printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
20            31235, 22, 86, 45.62);
21     printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
22            321, 55, 21, 122.);
23     printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
24            28764, 0, 24, .75);
25     printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
26            3232, 12, 0, 10.91);
27
28     // Print end message
29     printf("\n\tEnd of Report\n");
30     return 0;
31 } // main
```

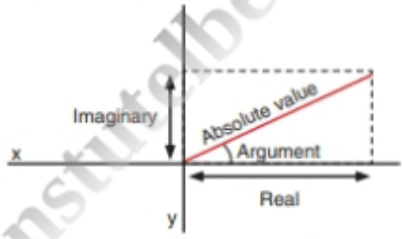
There are a few things about Program 2-8 that you should note. First, it is fully documented. Professional programmers often ignore documentation on "one-time-only" programs, thinking they will throw them away, only to find that they end up using them over and over. It only takes a few minutes to document a program, and it is always time well spent. If nothing else, it helps clarify the program in your mind.

Next, look carefully at the formatting for the `print` statements. Spacing is controlled by a combination of tabs and format code widths. The double spacing for the end of report message is controlled by placing a newline command (`\n`) at the beginning of the message in Statement 29.

Finally, note that the program concludes with a return statement that informs the operating system that it concluded successfully. Attention to details, even in small programs, is the sign of a good programmer.

A complex number is made of two components: a real part and an imaginary part. In mathematics, it can be represented as a vector with two components. The real part is the projection of the vector on the horizontal axis (x) and the imaginary part is the projection of the vector on the vertical axis (y). In C, we use a complex number and a predefined library function to print the real and imaginary values. We can also find the length of the vector, which is the absolute value of the complex number and the angle of the vector, which is the argument of the vector. These four attributes are shown in **Figure 2-23**.

**Figure 2-23** Complex 0




As the figure shows, the absolute value of the complex a + b * I can be found as $(a + b)^{1/2}$. The argument can be found as arctan (b/a). The conjugate of a complex number is another complex number defined as a — b * I.

**Program 2-9** shows how to print the different attributes of a complex number using the predefined functions `creal` `cimag`, `cabs`, and `carg`.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Program 2-9 | Print complex number attributes

```
1  /* Print attributes of a complex number.
2     Written by:
3     Date
4  */
5  #include <stdio.h>
6  #include <math.h>
7  #include <complex.h>
8
9  int main (void)
10 {
11 // Local Declarations
12    double complex x = 4 + 4 * I;
13    double complex xc;
14
15    // Statements
16    xc = conj (x);
17    printf("%f %f %f %f\n", creal(x), cimag(x),
18           cabs(x), carg(x));
19
20    printf("%f %f %f %f\n", creal(xc), cimag(xc),
21           cabs(xc), carg(xc));
22    return 0;
23 } // main
```

**Output**

```
4.000000 4.000000 5.656854 0.785398
4.000000 -4.000000 5.656854 -0.785398
```

In C you can add, subtract, multiply, and divide two complex numbers using the same operators (+, -, *, /) that are used for real numbers. **Program 2-10** demonstrates the arithmetic use of operators with complex numbers.

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

## Program 2-10 | Complex number arithmetic

```
1  /* Demonstrate complex number arithmetic.
2     Written by:
3     Date:
4  */
5  #include <stdio.h>
6  #include <math.h>
7  #include <complex.h>
8
9  int main (void)
10 {
11 // Local Declarations
12    double complex x = 3 + 4 * I;
13    double complex y = 3 - 4 * I;
14    double complex sum;
15    double complex dif;
16    double complex mul;
17    double complex div;
18
19 // Statements
20    sum = x + y;
21    dif = x - y;
22    mul = x * y;
23    div = x / y;
24
25    printf("%f %f %f %f\n", creal(sum), cimag(sum),
26    cabs(sum), carg(sum));
27    printf("%f %f %f %f\n", creal(dif), cimag(dif),
28    cabs(dif), carg(dif));
29    printf("%f %f %f %f\n", creal(mul), cimag(mul),
30    cabs(mul), carg(mul));
31    printf("%f %f %f %f\n", creal(div), cimag(div),
32    cabs(div), carg(div));
33    return 0;
34 } // main
```

**Output**

```
6.000000 0.000000 6.000000 0.000000
0.000000 8.000000 8.000000 1.570796
25.000000 0.000000 25.000000 0.000000
-0.280000 0.960000 1.000000 1.854590
```

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

# Software Engineering

Although this chapter introduces only a few programming concepts, there is still much to be said from a software engineering point of view. We will discuss the concepts of program documentation, data naming, and data hiding.

## Program Documentation

There are two levels of program documentation. The first is the general documentation at the start of the program. The second level is found within each function.

## General Documentation

**Program 2-11** illustrates recommended program documentation. Each program should start with a general description of the program. Following the general description is the name of the author and the date the program was written. Following the date is the program's change history, which documents the reason and authority for all changes. For a production program, whose use spans several years, the change history can become extensive.

**Program 2-11** | Sample of general program documentation

```
1  /* A sample of program documentation. Each program starts
2     with a general description of the program.
3     Often, this description
4     can be taken from the requirements specification
5     given to the programmer.
6     Written by: original author
7     Date:  Date first released to production
8     Change History:
9        <date> Included in this documentation is a short
10       description of each change.
11 */
```

### Module Documentation

Whenever necessary, a brief comment for blocks of code should be included. A block of code is much like a paragraph in a report. It contains one thought—that is, one set of statements that accomplish a specific task. Blocks of code in a program are separated by blank program lines, just as we skip blank lines between paragraphs in reports.

If the block of code is difficult, or if the logic is especially significant, then a short—one- or two-line—description of the block's purpose and/or operation should be provided.

Some programming experts recommend documenting each variable in a program. We disagree with this approach. First, the proper location for variable documentation is in a data dictionary. A data dictionary is a system documentation tool that contains standard names, descriptions, and other information about data used in a system.

Second, good data names eliminate the need for variable comments. In fact, if you think you need to document the purpose of a variable, check your variable name. You will usually find that improving the name eliminates the need for the comment.

### Data Names

Another principle of good structured programming is the use of intelligent data names. This means that the variable name itself should give the reader a good idea about what data it contains and maybe even an idea about how the data are used.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Although there are obvious advantages to keeping names short, the advantage is quickly lost if the names become so cryptic that they are unintelligible. Programmers often struggle for hours to find a bug, only to discover that the problem was an incorrect variable. The time saved keying short, cryptic names is often lost ten- or a hundredfold in debugging time.

Here are some guidelines to help you construct good, intelligent data names:

1. The name should match the terminology of the user as closely as possible.

   Let's suppose that you are writing a program to calculate the area of a rectangle. Mathematicians often label the sides of a rectangle *a* and *b*, but their real names are length and width. Therefore, your program should call the sides of the rectangle length and width. These names are commonly used by anyone describing a rectangle.

2. When necessary for readability, and to separate similar variables from each other, combine terms to form a variable name.

   Suppose that you are working on a project to compute a payroll. There are many different types of taxes. Each of the different taxes should be clearly distinguished from the others by good data names. **Table 2-12** shows both good and bad names for this programming situation. Most of the poor names are either too abbreviated to be meaningful (such as ftr) or are generic names (such as rate) that could apply to many different pieces of data.

**Table 2-12** Examples of good and poor data names

| GOOD NAMES | | POOR NAMES |
|---|---|---|
| `ficaTaxRate` | `fica_tax_rate` | `rate ftr frate fica` |
| `ficaWitholding` | `fica_witholding` | `fwh ficaw wh` |
| `ficaWthldng` | `fica_wthldng` | `fcwthldng wthldng` |
| `ficaMax` | `ficaDlrMax` | `max fmax` |

   Note the two different concepts for separating the words in a variable's name demonstrated in Table 2-12. In the first example, the first letter of each word is capitalized. In the second example, the words are separated with an underscore. Both are good techniques for making a compound name readable. If you use capitalization, keep in mind that C is case sensitive, so you must be careful to use the same cases for the name each time you use it.

3. Do not create variable names that are different by only one or two letters, especially if the differences are at the end of the word. Names that are too similar create confusion. On the other hand, a naming pattern makes it easier to recall the names. This is especially true when user terminology is being used. Thus, all the good names in Table 2-12 start with `fica`.

4. Abbreviations, when used, should clearly indicate the word being abbreviated.

   Table 2-12 also contains several examples of good abbreviations. Whenever possible, use abbreviations created by the users. They will often have a glossary of abbreviations and acronyms that they use.

   Short words are usually not abbreviated. If they are short in the first place, they don't need to be made shorter.

5. Avoid the use of generic names.

   Generic names are programming or user jargon. For example, `count` and `sum` are both generic names. They tell you their purpose but don't give you any clue as to the type of data they are associated with. Better names would be `emplyCnt` and `ficaSum`. Programmers are especially fond of using generic names, but they tend to make the program confusing. Several of the poor names in Table 2-12 are generic.

6. Use memory constants or defined constants rather than literals for values that are hard to read or that might change from system to system.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Some constants are nearly impossible to read. We pointed out the space earlier. If you need a space often, create a defined constant for it. **Table 2-13** contains several examples of constants that are better when coded as defined constants.

**Table 2-13** Examples of defined constants

| | |
|---|---|
| `#define SPACE ' '` | `#define BANG '!'` |
| `#define DBL_QTE ""` | `#define QUOTE '\"'` |
| `#define COMMA ','` | `#define COLON ':'` |

### Data Hiding

In "Structure of a C Program" in Section 2.2, you read about the concept of global and local variables. We pointed out that anything placed before `main` was said to be in the global part of the program. With the exception of data that must be visible to other programs, no variables should be placed in this section.

One of the principles of structured programming states that the data structure should be hidden from view. The two terms you usually hear in connection with this concept are data hiding and data encapsulation. Both of these principles have as their objective protecting data from accidental destruction by parts of your program that don't require access to the data. In other words, if a part of your program doesn't require data to do its job, it shouldn't be able to see or modify the data. Until you learn to use functions, however, you will not be able to provide this data-hiding capability.

Nevertheless, you should start your programming with good practices. And since your ultimate objective is good structured programming, we now formulate our first programming standard: Any variables placed in the global area of your program—that is, before `main`—can be used and changed by every part of your program. This is undesirable and is in direct conflict with the structured programming principles of data hiding and data encapsulation. Thus, you should never place a variable in the global area of a program.

## Tips and Common Programming Errors

1. Well-structured programs use global (defined) constants but do not use global variables.
2. The function header for `main` should be complete. The following format is recommended:

   ```
   int main (void)
   ```

   a. If you forget the parentheses after `main`, you will get a compile error.
   b. If you put a semicolon after the parentheses, you will get a compile error.
   c. If you misspell `main` you will not get a compile error, but you will get an error when you try to link the program. All programs must have a function named `main`.
3. If you forget to close the format string in the `scanf` or `printf` statement, you will get a compile error.
4. Using an incorrect conversion code for the data type being read or written is a run-time error. You can't read an integer with a `float` conversion code. Your program will compile with this error, but it won't run correctly.
5. Not separating read and write parameters with commas is a compile error.
6. Forgetting the comma after the format string in a read or write statement is a compile error.
7. Not terminating a block comment with a close token (*/) is a compile error.
8. Not including required libraries, such as `stdio.h`, at the beginning of your program is an error. Your program may compile, but the linker cannot find the required functions in the system library.

DO NOT COPY nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

9. If you misspell the name of a function, you will get an error when you link the program. For example, if you misspell `scanf` or `printf`, your program will compile without errors, but you will get a linker error. Using the wrong case is a form of spelling error. For example, each of the following function names are different:

```
scanf, Scanf, SCANF printf, Printf, PRINTF
```

10. Forgetting the address operator (&) on a `scanf` parameter is a logic (runtime) error.
11. Do not use commas or other characters in the format string for a `scanf` statement. This will most likely lead to a run-time error when the user does not enter matching commas or characters. For example, the comma in the following statement will create a run-time problem if the user doesn't enter it exactly as coded.

```
scanf ("%d, %d", &a, &b);
```

12. Unless you specifically want to read a whitespace character, put a space before the character conversion specification in a `scanf` statement.
13. Using an address operator (&) with a variable in the `printf` statement is usually a run-time error.
14. Do not put a trailing whitespace at the end of a format string in `scanf`. This is a fatal run-time error.

## Summary

- In 1972, Dennis Ritchie designed C at Bell Laboratories.
- In 1989, the American National Standards Institute (ANSI) approved ANSI C; in 1990, the ISO standard was approved.
- The basic component of a C program is the function.
- Every C function is made of declarations, definitions, and one or more statements.
- One and only one of the functions in a C program must be called `main`.
- To make a program more readable, use comments. A comment is a sequence of characters ignored by the compiler. C uses two types of comments: block and line. A block comment starts with the token /* and ends with the token */. A line comment starts with the // token; the rest of the line is ignored.
- Identifiers are used in a language to name objects.
- C types include `void`, integral, floating point, and derived.
- A `void` type is used when C needs to define a lack of data.
- An integral type in C is further divided into Boolean, character, and integer.
  - A Boolean data type takes only two values: `true` and `false`. It is designated by the keyword `bool`.
  - A character data type uses values from the standard alphabet of the language, such as ASCII or Unicode. There are two character type sizes, `char` and `w_char`.
  - An integer data type is a number without a fraction. C uses four different integer sizes: `short int`, `int`, `long int`, and `long long int`.
- The floating-point type is further divided into real, imaginary, and complex.
  - A real number is a number with a fraction. It has three sizes: `float`, `double`, and `long double`.
  - The imaginary type represents the imaginary part of a complex number. It has three sizes, `float imaginary`, `double imaginary`, and `long double imaginary`.
  - The complex type contains a real and an imaginary part. C uses three complex sizes: `float complex`, `double complex`, and `long double complex`.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

- A constant is data whose value cannot be changed.
- Constants can be coded in three different ways: as literals, as define commands, and as memory constants.
- Variables are named areas of memory used to hold data.
- Variables must be declared and defined before being used in C.
- To input data through the keyboard and to output data through the monitor, use the standard formatted input/output functions.
- `scanf` is a standard input function for inputting formatted data through the keyboard.
- `printf` is a standard output function for outputting formatted data to the monitor.
- As necessary, programs should contain comments that provide the reader with in-line documentation for blocks of code.

Programs that use "intelligent" names are easier to read and understand.

## Key Terms

address list
address operator
ASCII
binary stream
block comment
Boolean
character constant
character set
comments
complex type
constant
conversion code
conversion specifications
declaration
declaration section
definitions
end of file
escape character
flag modifier
floating-point types
format control string
functions
global declaration section
header files
identifier
imaginary type
include
initializer
integral types
intelligent data names
justification
keywords
Latin character set
line comment
literal
memory constant
padding
parameter
parameter list
precision modifier
program documentation
real type
reserved words
sign flag
size modifier
statements
statement section
stream
string
string constant
syntax
token
type
type qualifier
variable
width modifier

## Review Questions

1. The purpose of a header file, such as `stdio.h`, is to store a program's source code.
   a. True
   b. False
2. Any valid printable ASCII character can be used in an identifier.
   a. True
   b. False
3. The C standard function that receives data from the keyboard is `printf`.
   a. True
   b. False
4. Which of the following statements about the structure of a C program is false?
   a. A C program starts with a global declaration section.

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY
nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**b.** Declaration sections contain instructions to the computer.
**c.** Every program must have at least one function.
**d.** One and only one function may be named `main`.
**e.** Within each function there is a local declaration section.

**5.** Which of the following statements about block comments is false?
**a.** Comments are internal documentation for programmers.
**b.** Comments are used by the preprocessor to help format the program.
**c.** Comments begin with a `/*` token.
**d.** Comments cannot be nested.
**e.** Comments end with a `*/` token.

**6.** Which of the following identifiers is not valid?
**a.** `_option`
**b.** `amount`
**c.** `sales_amount`
**d.** `salesAmount`
**e.** `$salesAmount`

**7.** Which of the following is not a data type?
**a.** `char`
**b.** `float`
**c.** `int`
**d.** `logical`
**e.** `void`

**8.** The code that establishes the original value for a variable is known as a(n) _____.
**a.** assignment
**b.** constant
**c.** initializer
**d.** originator
**e.** value

**9.** Which of the following statements about a constant is true?
**a.** Character constants are coded using double quotes (`"`).
**b.** It is impossible to tell the computer that a constant should be a `float` or a `long double`.
**c.** Like variables, constants have a type and may be named.
**d.** Only integer values can be used in a constant.
**e.** The value of a constant may be changed during a program's execution.

**10.** The _____ conversion specification is used to read or write a short integer.
**a.** `%c`
**b.** `%d`
**c.** `%f`
**d.** `%hd`
**e.** `%lf`

**11.** To print data left justified, you would use a _____ in the conversion specification.
**a.** flag
**b.** precision
**c.** size
**d.** width
**e.** width and precision

**12.** The _____ function reads data from the keyboard.
**a.** `displayf`
**b.** `printf`
**c.** `read`
**d.** `scanf`
**e.** `write`

**13.** One of the most common errors for new programmers is forgetting to use the address operator for variables in a `scanf` statement. What is the address operator?
**a.** The address modifier (@) in the conversion specification
**b.** The ampersand (&)
**c.** The caret (^)
**d.** The percent (%)
**e.** The pound sign (#)

## Exercises

**14.** Which of the following is *not* a character constant in C?
**a.** `'C'`
**b.** `'bb'`
**c.** `"C"`
**d.** `'?'`
**e.** `' '`

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**15.** Which of the following is *not* an integer constant in C?
- **a.** `-320`
- **b.** `+45`
- **c.** `-31.80`
- **d.** `1456`
- **e.** `2,456`

**16.** Which of the following is *not* a floating-point constant in C?
- **a.** `45.6`
- **b.** `-14.05`
- **c.** `'a'`
- **d.** `pi`
- **e.** `40`

**17.** What is the type of each of the following constants?
- **a.** `15`
- **b.** `-14.24`
- **c.** `'b'`
- **d.** `"I"`
- **e.** `"16"`

**18.** Which of the following is *not* a valid identifier in C?
- **a.** `A3`
- **b.** `4A`
- **c.** `if`
- **d.** `IF`
- **e.** `tax-rate`

**19.** What is the type of each of the following constants?
- **a.** `"7"`
- **b.** `3`
- **c.** `"3.14159"`
- **d.** `'2'`
- **e.** `5.1`

**20.** What is the type of each of the following constants?
- **a.** `"Hello"`
- **b.** `15L`
- **c.** `8.5L`
- **d.** `8.5f`
- **e.** `'\a'`

**21.** Which of the following identifiers are valid and which are invalid? Explain your answer.
- **a.** `num`
- **b.** `num2`
- **c.** `2dNum`
- **d.** `2d_num`
- **e.** `num#2`

**22.** Which of the following identifiers are valid and which are invalid? Explain your answer.
- **a.** `num-2`
- **b.** `num 2`
- **c.** `num_2`
- **d.** `_num2`
- **e.** `_num_2`

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**23.** What is output from the following program fragment? To show your output, draw a grid of C at least 8 lines with at least 15 characters per line.

```
// Local Declarations
int x = 10;
char w = 'Y';
float z = 5.1234;
// Statements
printf("\nFirst\nExample\n:");
printf("%5d\n, w is %c\n", x, w);
printf("\nz is %8.2f\n", z);
```

**24.** Find any errors in the following program.

```
// This program does nothing
int main
{
   return 0;
}
```

**25.** Find any errors in the following program.

```
#include (stdio.h)
int main (void)
{
   print ("Hello World");
   return 0;
}
```

**26.** Find any errors in the following program.

```
include <stdio>
int main (void)
{
   printf('We are to learn correct');
   printf('C language here');
   return 0;
} // main
```

**27.** Find any errors in the following program.

```
/* This is a program with some errors
   in it to be corrected.
*/
int main (void)
{
// Local Declarations
   integer a;
   floating-point b;
   character c;
// Statements
    printf("The end of the program.");
    return 0;
} // main
```

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

DO NOT COPY nstutelberg@gmail.com

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**28.** Find any errors in the following program.

```
/* This is another program with some errors in it to be
corrected.
*/
int main (void)
{
// Local Declarations
   a int;
   b float, double;
   c, d char;
// Statements
   printf("The end of the program.");
   return 0;
} // main
```

**29.** Find any errors in the following program.

```
/* This is the last program to be
corrected in these exercises.
*/
int main (void)
{
// Local Declarations
   a int;
   b: c : d char;
   d , e, f double float;
// Statements
   printf("The end of the program.");
   return 0;
} // main
```

## Problems

**30.** Code the variable declarations for each of the following:

- **a.** a character variable named `option`
- **b.** an integer variable, sum, initialized to 0
- **c.** a floating-point variable, `product`, initialized to `1`

**31.** Code the variable declarations for each of the following:

- **a.** a short integer variable named code
- **b.** a constant named `salesTax` initialized to .0825
- **c.** a floating-point named sum of size double initialized to 0

**32.** Write a statement to print the following line. Assume the total value is contained in a variable named cost.

```
The sales total is: $ 172.53
```

DO NOT COPY
nstutelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Printed by: nstutelberg@gmail.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

**33.** Write a program that uses four print statements to print the pattern of asterisks shown below.

```
******
******
******
******
```

**34.** Write a program that uses four print statements to print the pattern of asterisks shown below.

```
*
**
***
****
```

**35.** Write a program that uses defined constants for the vowels in the alphabet and memory constants for the even decimal digits (0, 2, 4, 6, 8). It then prints the following three lines using literal constants for the odd digits.

```
a e i o u
0 2 4 6 8
1 3 5 7 9
```

**36.** Write a program that defines five integer variables and initializes them to 1, 10, 100, 1000, and 10000. It then prints them on a single line separated by space characters using the decimal conversion code (`%d`), and on the next line with the float conversion code (`%f`). Note the differences between the results. How do you explain them?

**37.** Write a program that prompts the user to enter a quantity and a cost. The values are to be read into an integer named `quantity` and a float named `unitPrice`. Define the variables and use only one statement to read the values. After reading the values, skip one line and print each value, with an appropriate name, on a separate line.

**38.** Write a program that prompts the user to enter an integer and then prints the integer first as a character, then as a decimal, and finally as a float. Use separate print statements. A sample run is shown below.

```
The number as a character: K
The number as a decimal: 75
The number as a float: 0.000000
```

## Projects

**39.** Write a C program using `printf` statements to print the three first letters of your first name in big blocks. This program does not read anything from the keyboard. Each letter is formed using seven rows and five columns using the letter itself. For example, the letter B is formed using 17 b's, as shown below as part of the initials BEF.

```
BBBB    EEEEE   FFFFF
B   B   E       F
B   B   E       F
BBBB    EEE     FFF
B   B   E       F
B   B   E       F
BBBB    EEEEE   F
```

DO NOT COPY
nstuelberg@gmail.com

Copyright 2023 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.