Chapter 1
Introduction to Computers, the Internet and the Web

C How to Program, 8/e

1.2 Hardware and Software

- Computers process data under the control of sequences of instructions called computer programs.

1.3 Data Hierarchy

- Data items processed by computers form a data hierarchy that becomes larger and more complex in structure as we progress from bits to characters to fields, and so on.

Why should I know this?

Prepare for understanding data structure

C how C manipulates data?

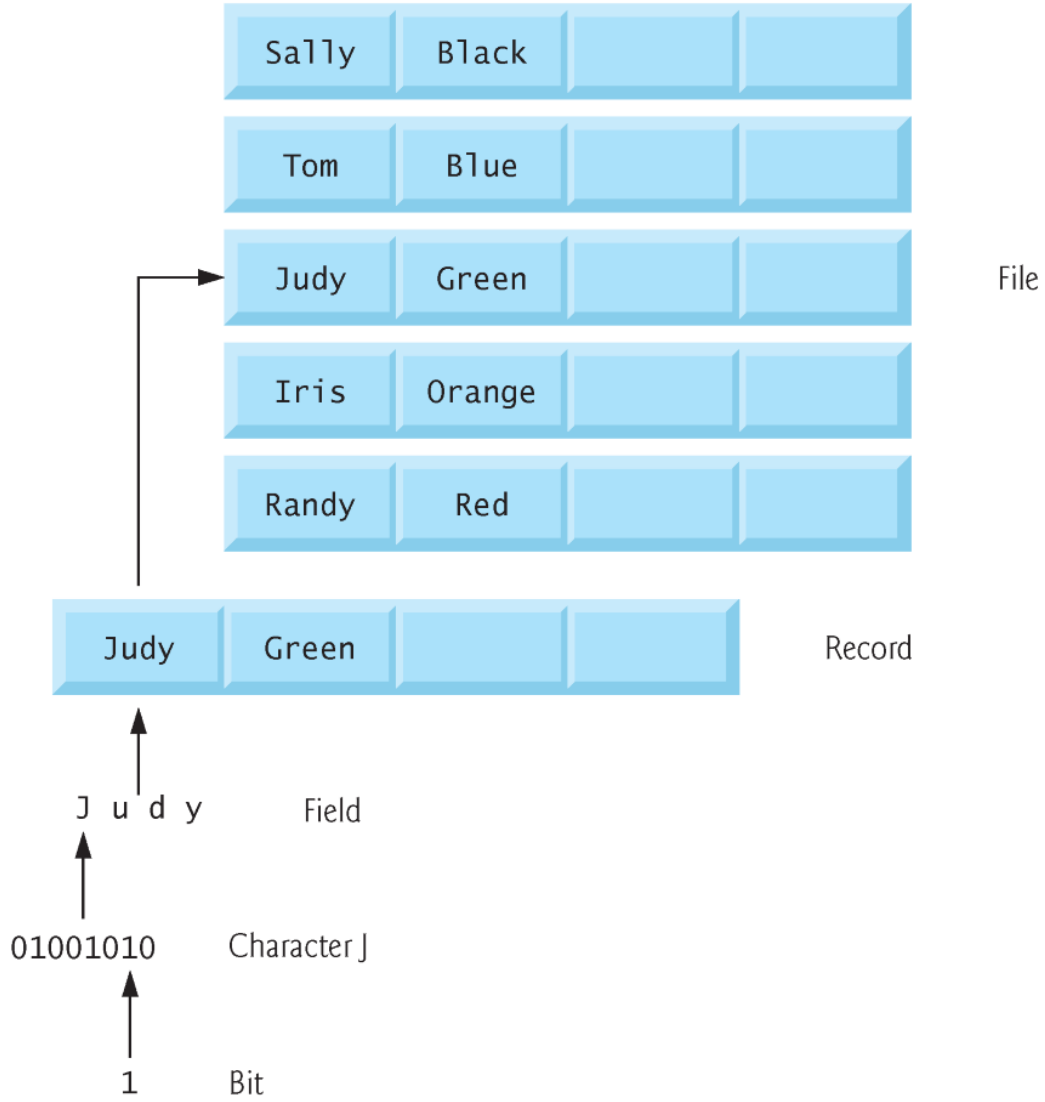

Fig. 1.2 | Data hierarchy.

1.3 Data Hierarchy

- Bits
  - The smallest data item in a computer can assume the value 0 or the value 1.
  - Short for “binary digit”—a digit that can assume one of *two* values.
- Characters
  - It’s tedious for people to work with data in the low-level form of bits
  - Instead, they prefer to work with
    - *decimal digits* (0–9),
    - *letters* (A–Z and a–z), and
    - *special symbols* (e.g., $, @, %, &, *, (,), –, +, ", :, ? and /)
  - The computer’s **character set** is the set of all the characters used to write programs and represent- data items.
  - Computers process only 1s and 0s, so a computer’s character set represents every character as a pattern of 1s and 0s.
  - C supports various character sets (including **Unicode®**)
  - that are composed of characters containing one, two or four bytes (8, 16 or 32 bits).
  - Unicode contains characters for many of the world’s languages.
  - The minimum Unsigned integer in a byte is 0 and the maximum is 255 in all 256 values.
  - https://www.branah.com/unicode-converter

1.3 Data Hierarchy

- Fields
  - Composed of characters or bytes.
  - A field is a group of characters or bytes that conveys meaning.
  - e.g; CS203, F18
- Records
  - Several related fields can be used to compose a **record**—a group of related fields.

1.3 Data Hierarchy

- File
  - A group of related records.
  - More generally, a file contains arbitrary data in arbitrary formats.
  - In some operating systems, a file is viewed simply as a *sequence of bytes*—any organization of the bytes in a file, such as organizing the data into records, is a view created by the application programmer.
- Database
  - A **database** is a collection of data organized for easy access and manipulation.
  - The most popular model is the *relational database*, in which data is stored in simple *tables*.
  - A table includes *records* and *fields*.
  - You can *search*, *sort* and otherwise manipulate the data based on its relationship to multiple tables or databases.
- Big Data
  - According to IBM, approximately 2.5 quintillion bytes (2.5 *exabytes*) of data are created daily and 90% of the world's data was created in just the past two years
  - According to an IDC study, the global data supply will reach 40 *zettabytes* (equal to 40 trillion gigabytes) annually by 2020.
  - **Big data** applications deal with massive amounts of data

1.4 Machine Languages, Assembly Languages and High-Level Languages

- Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate *translation* steps.
- Any computer can directly understand only its own machine language, defined by its hardware design.
- Machine languages generally consist of numbers (ultimately reduced to 1s and 0s). Such languages are cumbersome for humans.
- Programming in machine language—the numbers that computers could directly understand—was simply too slow and tedious for most programmers.
- Instead, they began using Englishlike abbreviations to represent elementary operations.
- These abbreviations formed the basis of assembly languages.
- *Translator programs* called assemblers were developed to convert assembly-language programs to machine language.

1.4 Machine Languages, Assembly Languages and High-Level Languages

- Although assembly-language code is clearer to humans, it's incomprehensible to computers until translated to machine language.
- To speed the programming process even further, high-level languages were developed in which single statements could be written to accomplish substantial tasks.
- High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical expressions.
- Translator programs called compilers convert high-level language programs into machine language.
- Interpreter programs were developed to execute high-level language programs directly, although more slowly than compiled programs.
- Scripting languages such as JavaScript and PHP are processed by interpreters.

Planet gobbledygook: https://www.youtube.com/watch?v=_C5AHaS1mOA

https://www.youtube.com/watch?v=1veGrAjgJBY

1.5 The C Programming Language

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

1.5 The C Programming Language (Cont.)

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

1.5 The C Programming Language (Cont.)

Built for Performance

- C is widely used to develop systems that demand performance, such as operating systems, embedded systems, real-time systems and communications systems (Figure 1.4).

1.6 C Standard Library

- As you'll learn in Chapter 5, C programs consist of pieces called functions.
- You can program all the functions you need to form a C program, but most C programmers take advantage of the rich collection of existing functions called the C Standard Library.
- Visit the following website for the C Standard Library documentation:

  `www.dinkumware.com/manuals/#Standard%20C%20Library`

1.6 C Standard Library (Cont.)

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

1.6 C Standard Library (Cont.)

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

Performance Tip 1.1

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

Portability Tip 1.2

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

1.7 C++ and Other C-Based Languages

- C++ was developed by Bjarne Stroustrup at Bell Laboratories.
- It has its roots in C, providing a number of features that "spruce up" the C language.
- More important, it provides capabilities for object-oriented programming.
- Objects are essentially reusable software components that model items in the real world.
- Using a modular, object-oriented design and implementation approach can make software development groups more productive.

1.9 Typical C Program Development Environment

- C systems generally consist of several parts: a program development environment, the language and the C Standard Library.
- C programs typically go through six phases to be executed
- These are: edit, preprocess, compile, link, load and execute.

1.9 Typical C Program Development Environment (Cont.)

- [Note: The programs we write will run with little or no modification on most current C systems, including Microsoft Windows-based systems.]
- If you're not using a Linux system, refer to the manuals for your system.

1.9 Phase 1: Creating a Program

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

1.9 Phases 2 and 3: Preprocessing and Compiling a C Program

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

1.9 Phases 2 and 3: Preprocessing and Compiling a C Program (Cont.)

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

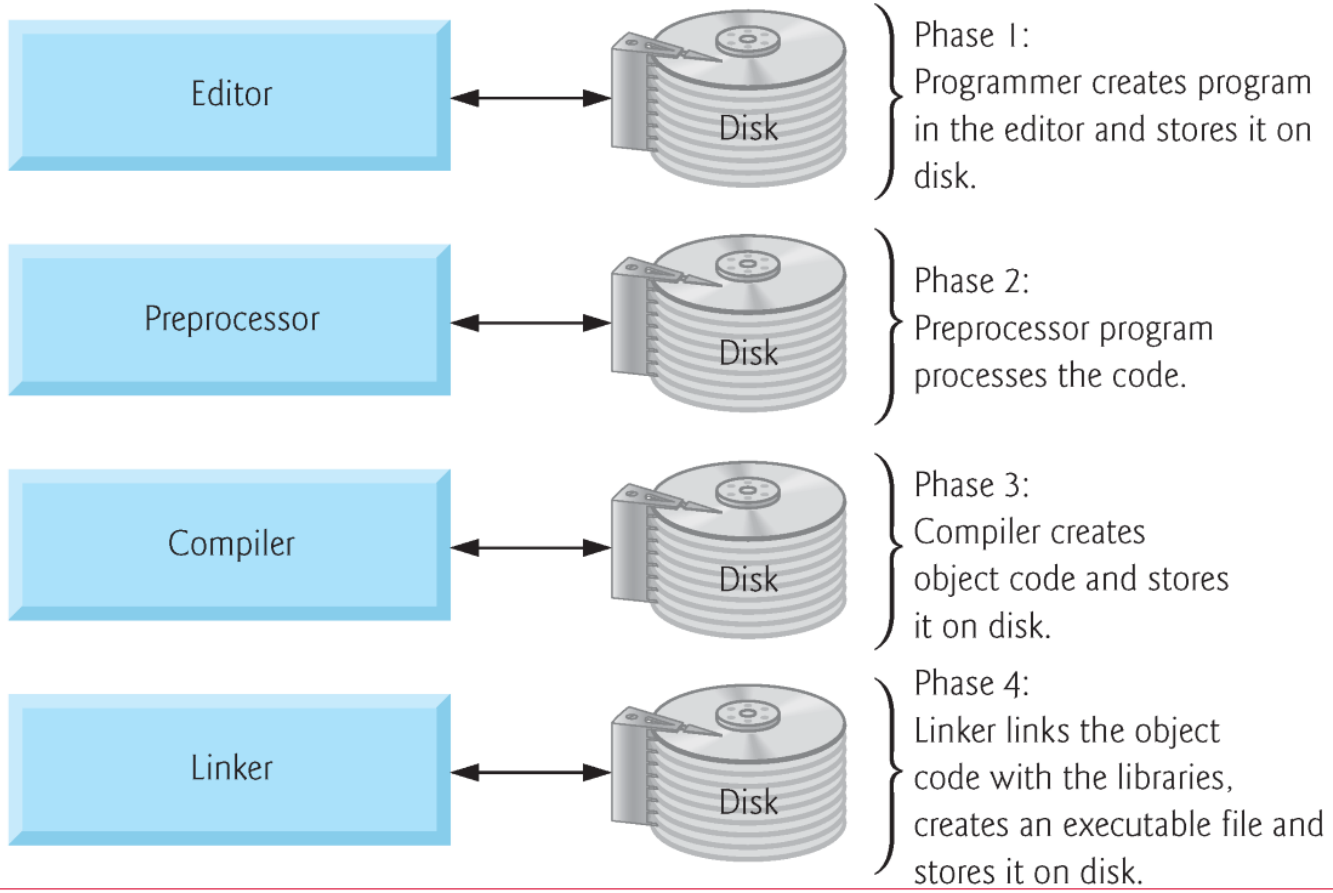

Fig. 1.6 | Typical C development environment. (Part 1 of 2.)

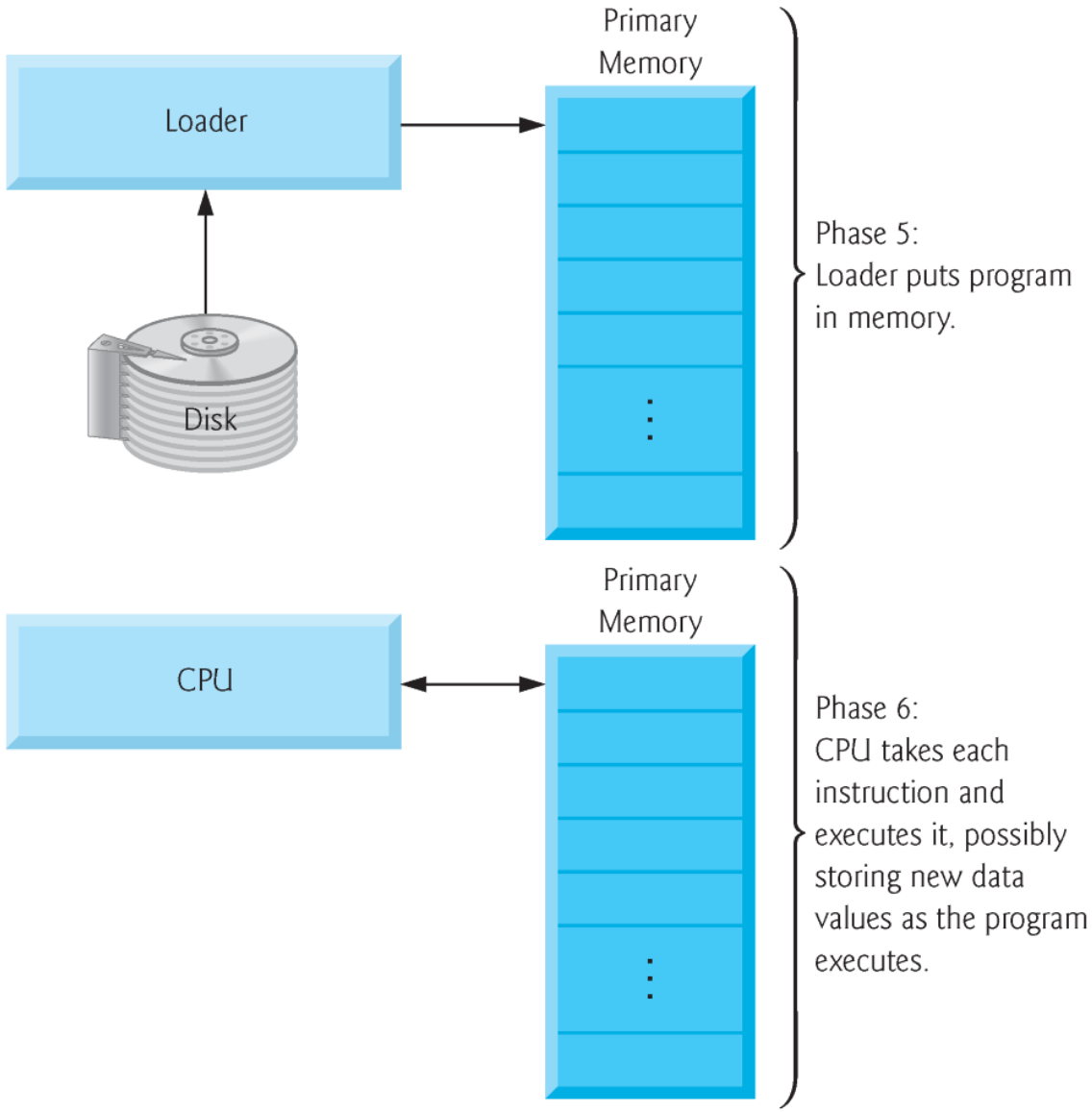

Fig. 1.6 | Typical C development environment. (Part 2 of 2.)

1.9 Phase 4: Linking

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

1.9 Phase 4: Linking (Cont.)

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

1.9 Phase 5: Loading

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

1.9 Phase 6: Execution

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

1.9 Problems That May Occur at Execution Time

- Programs do not always work on the first try.
- Each of the preceding phases can fail because of various errors that we'll discuss.
- For example, an executing program might attempt to divide by zero (an illegal operation on computers just as in arithmetic).
- This would cause the computer to display an error message.
- You would then return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections work properly.

Common Programming Error 1.1

Errors such as division-by-zero occur as a program runs, so they are called runtime errors or execution-time errors. Divide-by-zero is generally a fatal error, i.e., one that causes the program to terminate immediately without successfully performing its job. Nonfatal errors allow programs to run to completion, often producing incorrect results.

1.9 Standard Input, Standard Output and Standard Error Streams

- Most C programs input and/or output data.
- Certain C functions take their input from `stdin` (the standard input stream), which is normally the keyboard, but `stdin` can be connected to another stream.
- Data is often output to `stdout` (the standard output stream), which is normally the computer screen, but `stdout` can be connected to another stream.
- When we say that a program prints a result, we normally mean that the result is displayed on a screen.

1.9 Standard Input, Standard Output and Standard Error Streams (Cont.)

- Data may be output to devices such as disks and printers.
- There is also a standard error stream referred to as `stderr`.
- The `stderr` stream (normally connected to the screen) is used for displaying error messages.
- It's common to route regular output data, i.e., `stdout`, to a device other than the screen while keeping `stderr` assigned to the screen so that the user can be immediately informed of errors.

1.10 Test-Driving a C Application in Windows, Linux and Mac OS X

- In this section, you'll run and interact with your first C application.
- You'll begin by running an entertaining guess-the-number game, which picks a number from 1 to 1000 and prompts you to guess it.
- If your guess is correct, the game ends.
- If your guess is not correct, the application indicates whether your guess is higher or lower than the correct number. There is no limit on the number of guesses you can make.

1.10.2 Running a C Application Using GNU C with Linux

```
~$ cd examples/ch01/GuessNumber/GNU
~/examples/ch01/GuessNumber/GNU$
```

Fig. 1.14 | Changing to the **GuessNumber** application's directory.

```
~/examples/ch01/GuessNumber/GNU$ gcc -std=c11 GuessNumber.c -o GuessNumber
~/examples/ch01/GuessNumber/GNU$
```

Fig. 1.15 | Compiling the **GuessNumber** application using the `gcc` command.

```
~/examples/ch01/GuessNumber/GNU$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Fig. 1.16 | Running the **GuessNumber** application.

```
~/examples/ch01/GuessNumber/GNU$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Fig. 1.17 | Entering an initial guess.

```
~/examples/ch01/GuessNumber/GNU$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

Fig. 1.18 | Entering a second guess and receiving feedback.

```
Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384

Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )?
```

Fig. 1.19 | Entering additional guesses and guessing the correct number.

```
Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )? 1

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Fig. 1.20 | Playing the game again.

```
Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )? 2

~/examples/ch01/GuessNumber/GNU$
```

Fig. 1.21 | Exiting the game.