Introduction to C

1.6 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.
- Computer languages may be divided into three general types:
  - Machine languages
  - Assembly languages
  - High-level languages
- Any computer can directly understand only its own machine language.
- Machine language is the "natural language" of a computer and as such is defined by its hardware design.

1.6 Machine Languages, Assembly Languages and High-Level Languages (Cont.)

- Machine language is often referred to as object code.
- Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time.
- Machine languages are machine dependent (i.e., a particular machine language can be used on only one type of computer).

1.6 Machine Languages, Assembly Languages and High-Level Languages

- Such languages are cumbersome for humans, as illustrated by the following section of an early machine-language program that adds overtime pay to base pay and stores the result in gross pay:
  - +1300042774
    +1400593419
    +1200274027
- Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations.
- These abbreviations formed the basis of assembly languages.

1.6 Machine Languages, Assembly Languages and High-Level Languages

- Translator programs called assemblers were developed to convert early assembly-language programs to machine language at computer speeds.
- The following section of an assembly-language program also adds overtime pay to base pay and stores the result in gross pay:
  - ```
    load    basepay
    add     overpay
    store   grosspay
    ```
- Although such code is clearer to humans, it's incomprehensible to computers until translated to machine language.
    ```

1.6 Machine Languages, Assembly Languages and High-Level Languages

- Computer usage increased rapidly with the advent of assembly languages, but programmers still had to use many instructions to accomplish even the simplest tasks.
- To speed the programming process, high-level languages were developed in which single statements could be written to accomplish substantial tasks.
- Translator programs called compilers convert high-level language programs into machine language.
- High-level languages allow programmers to write instructions that look almost like everyday English and contain commonly used mathematical notations.

1.6 Machine Languages, Assembly Languages and High-Level Languages

- A payroll program written in a high-level language might contain a statement such as
  - `grossPay = basePay + overTimePay;`
- C, C++, Microsoft's .NET languages (e.g., Visual Basic, Visual C++ and Visual C#) and Java are among the most widely used high-level programming languages.
- Interpreter programs were developed to execute high-level language programs directly (without the delay of compilation), although slower than compiled programs run.

1.7 History of C

- 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.
- Both BCPL and B were “typeless” languages—every data item occupied one “word” in memory, and the burden of typing variables fell on the shoulders of the programmer.

1.7 History of C (Cont.)

- The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a DEC PDP-11 computer in 1972.
- C initially became widely known as the development language of the UNIX operating system.
- Today, virtually all new major operating systems are written in C and/or C++.
- C is available for most computers.
- C is mostly hardware independent.
- With careful design, it's possible to write C programs that are portable to most computers.

1.7 History of C (Cont.)

- By the late 1970s, C had evolved into what is now referred to as "traditional C." The publication in 1978 of Kernighan and Ritchie's book, *The C Programming Language*, drew wide attention to the language.
- The rapid expansion of C over various types of computers (sometimes called hardware platforms) led to many variations that were similar but often incompatible.
- In 1989, the C standard was approved; this standard was updated in 1999.
- C99 is a revised standard for the C programming language that refines and expands the capabilities of C.

Portability Tip 1.1

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

1.8 C Standard Library

- C programs consist of modules or pieces called functions.
- You can program all the functions you need to form a C program, but most C programmers take advantage of a rich collection of existing functions called the C Standard Library.
- Avoid reinventing the wheel.
- Instead, use existing pieces—this is called software reusability, and it's a key to the field of object-oriented programming, as you'll see when you study C++.
- When programming in C you'll typically use the following building blocks:
  - C Standard Library functions
  - Functions you create yourself
  - Functions other people have created and made available to you

1.8 C Standard Library (Cont.)

- If you use existing functions, you can avoid reinventing the wheel.
- In the case of the Standard C functions, you know that they're carefully written, and you know that because you're using functions that are available on all Standard C implementations, your programs will have a greater chance of being portable and error-free.

Performance Tip 1.1

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

Performance Tip 1.2

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

1.15 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 (Fig. 1.1).
- These are: edit, preprocess, compile, link, load and execute.
- Phase 1 consists of editing a file.
- This is accomplished with an editor program.

1.15 Typical C Program Development Environment (Cont.)

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

1.15 Typical C Program Development Environment (Cont.)

- 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.

1.15 Typical C Program Development Environment (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.

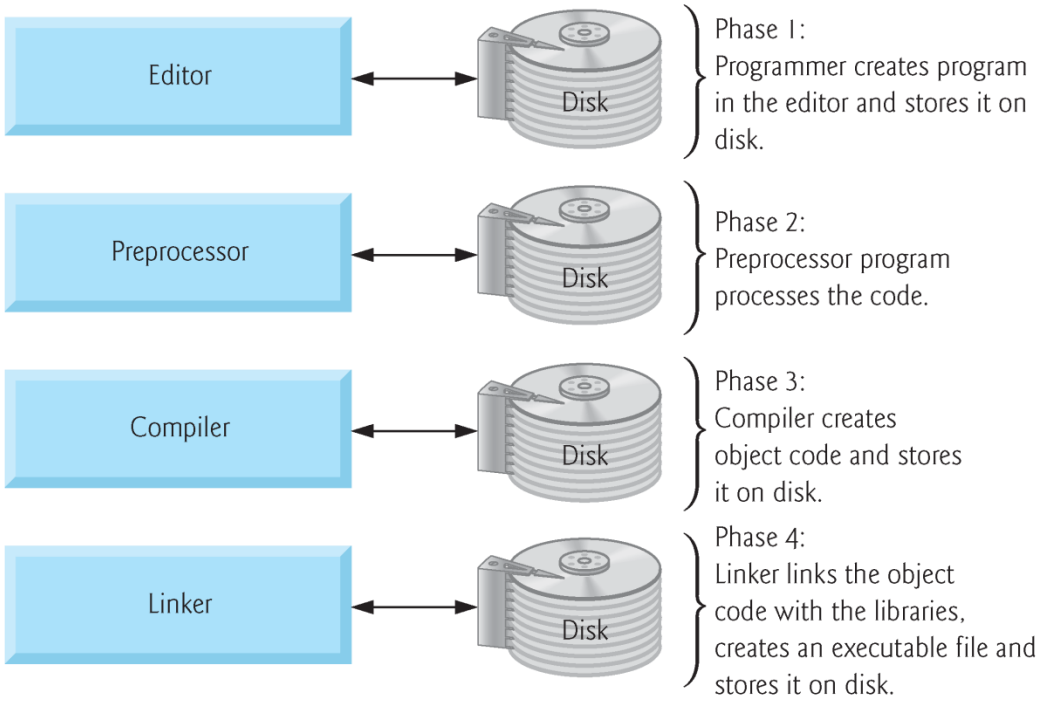

Fig. 1.1 | Typical C development environment. (Part 1 of 2.)

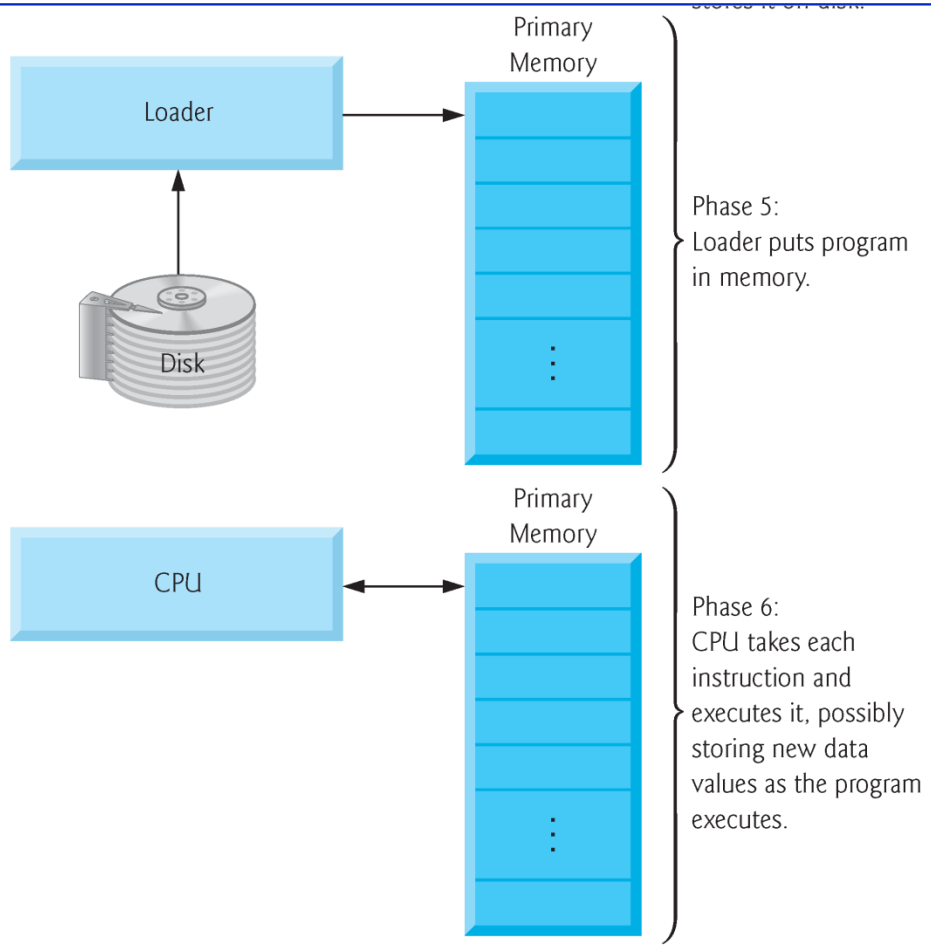

Fig. 1.1 | Typical C development environment. (Part 2 of 2.)

1.15 Typical C Program Development Environment (Cont.)

- 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 `cc` (or `gcc`).

1.15 Typical C Program Development Environment (Cont.)

- To compile and link a program named `welcome.c` type
  - `cc welcome.c`
- at the Linux prompt and press the *Enter* key (or *Return* key).
- [Note: Linux commands are case sensitive; make sure that you type lowercase `c`'s 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.

1.15 Typical C Program Development Environment (Cont.)

- 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.
- Finally, the computer, under the control of its CPU, executes the program one instruction at a time.

1.15 Typical C Program Development Environment (Cont.)

- To load and execute the program on a Linux system, type `./a.out` at the Linux prompt and press *Enter*.
- 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.

1.15 Typical C Program Development Environment (Cont.)

- 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.
- 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.

1.15 Typical C Program Development Environment (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.

Common Programming Error 1.1

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

Good Programming Practice 1.1

Write your C programs in a simple and straightforward manner. This is sometimes referred to as KIS ("keep it simple"). Do not "stretch" the language by trying bizarre usages.

Portability Tip 1.2

Although it's possible to write portable C programs, there are many problems between different C compilers and different computers that make portability difficult to achieve. Simply writing programs in C does not *guarantee portability. You'll often need to deal directly with computer variations.*

Software Engineering Observation 1.2

Read the manuals for the version of C you're using. Reference these manuals frequently to be sure you're aware of the rich collection of C features and that you're using these features correctly.

Software Engineering Observation 1.3

Your computer and compiler are good teachers. If you're not sure how a C feature works, write a program with that feature, compile and run the program and see what happens.

Your First C Program

```
# include <stdio.h>
main(void)
{
   printf("Hello, CSCI N305!\n");
}
```

Try it!

How to Compile A C Program Using Linux

gcc *first1*.c

- C programs end in the ".c" extension
- The executable file is called *a.out*

gcc *first1.c*

How to Execute Your Program

./*a.out*

- Type the name of the executable file at the prompt to run your program