Chapter 1 Introducing C

The Origins of C

- C is a by-product of UNIX, developed at Bell Laboratories by Ken Thompson, Dennis Ritchie, and others.
- Thompson designed a small language named B.
- B was based on BCPL, a systems programming language developed in the mid-1960s.
- By 1971, Ritchie began to develop an extended version of B, called NB ("New B") at first.
- As the language began to diverge more from B, he changed its name to C.
- The language was stable enough by 1973 that UNIX could be rewritten in C

Standardization of C

C89/C90

- ANSI standard X3.159-1989 (completed in 1988; formally approved in December 1989)
- International standard ISO/IEC 9899:1990

C99

- International standard ISO/IEC 9899:1999, which incorporates changes from Amendment of 1995

Properties of C

- Low-level
- Small
- Permissive

Strengths of C

- Efficiency
- Portability
- Power
- Flexibility
- Standard library
- Integration with UNIX

Weaknesses of C

- Programs can be error-prone.
- Programs can be difficult to understand.
- Programs can be difficult to modify.

Chapter 2 C Fundamentals

An Example Program

```
#include <stdio.h>
int main(void)
{
  printf("To C, or not to C.\n");
  return 0;
}
```

- This program might be stored in a file with a file name, say, pun.c.
- The file name doesn't matter, but the .c extension is often required.

Compiling and Linking

- Before a program can be executed, three steps are usually necessary:
  - *Preprocessing.* The *preprocessor* obeys commands that begin with # (known as *directives*)
  - *Compiling.* A *compiler* translates then translates the program into machine instructions (*object code*).
  - *Linking.* A *linker* combines the object code produced by the compiler with any additional code needed to yield a complete executable program.
- The preprocessor is usually integrated with the compiler.

- To compile and link the pun.c program under UNIX, enter the following command in a terminal or command-line window:

```
% cc pun.c
```

- Linking is automatic when using cc; no separate link command is necessary.
- After compiling and linking the program, cc leaves the executable program in a file named a.out by default.
- The -o option lets us choose the name of the file containing the executable program.
- The following command causes the executable version of pun.c to be named pun:

```
% cc -o pun pun.c
```

The GCC Compiler

- GCC is one of the most popular C compilers.
- GCC is supplied with Linux but is available for many other platforms as well.
- Using this compiler is similar to using cc:

```
% gcc -o pun pun.c
```

Integrated Development Environments

- An ***integrated development environment*** *(IDE)* is a software package that makes it possible to edit, compile, link, execute, and debug a program without leaving the environment.

The General Form of a Simple Program

```
directives

int main(void)
{
    statements
}
```

- C uses { and } in much the same way that some other languages use words like begin and end.
- Even the simplest C programs rely on three key language features:
  - Directives
  - Functions
  - Statements

Directives

- Before a C program is compiled, it is first edited by a preprocessor.
- Commands intended for the preprocessor are called directives.
- Example:

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

- `<stdio.h>` is a *header* containing information about C's standard I/O library.
- Directives always begin with a # character.
- By default, directives are one line long; there's no semicolon or other special marker at the end.

Functions

- A ***function*** is a series of statements that have been grouped together and given a name.
- ***Library functions*** are provided as part of the C implementation.
- A function that computes a value uses a return statement to specify what value it "returns":

```
return x + 1;
```

- **main** is special and mandatory: it gets called automatically when the program is executed.
- main returns a status code; the value 0 indicates normal program termination.
- If there's no return statement at the end of the main function, many compilers will produce a warning message.

Statements

- A ***statement*** is a command to be executed when the program runs.
- The program pun.c

```
#include <stdio.h>
int main(void)
{
  printf("To C, or not to C.\n");
  return 0;
}
```

uses only two kinds of statements. One is the return statement; the other is the *function call.*

- Asking a function to perform its assigned task is known as *calling* the function.
- pun.c calls printf to display a string:

```
printf("To C, or not to C.\n");
```

- C requires that each statement end with a semicolon.
- Directives are normally one line long, and they don't end with a semicolon.

Statements

- When the `printf` function displays a *string literal*—characters enclosed in double quotation marks—it doesn't show the quotation marks.
- `printf` doesn't automatically advance to the next output line when it finishes printing.
- To make `printf` advance one line, include `\n` (the *new-line character*) in the string to be printed.
- The statement

```
printf("To C, or not to C:
        that is the question.\n");
```

is incorrect. But could be corrected by two calls of printf:

```
printf("To C, or not to C: ");
printf("that is the question.\n");
```

Comments

- A *comment* begins with /* and end with */.

```
/* This is a comment */
```

- Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text.
- In C99, comments can also be written in the following way:

```
// This is a comment
```

- This style of comment ends automatically at the end of a line.
- Advantages of // comments:
  - Safer: there's no chance that an unterminated comment will accidentally consume part of a program.
  - Multiline comments stand out better.

Variables and Assignment

- Most programs need to a way to store data temporarily during program execution.
- These storage locations are called ***variables.***
- Every variable must have a ***type.***
- C has a wide variety of types, including int and float.
- Variables must be declared before they can be used.
- When main contains declarations, these must precede statements:

```
int main(void)
{
 declarations
 statements
}
```

- In C99, declarations don't have to come before statements.
- A variable can be given a value by means of ***assignment***, e.g.

```
height = 8;
```

- An int variable is normally assigned a value of type int, and a float variable is normally assigned a value of type float.
- Once a variable has been assigned a value, it can be used to help compute the value of another variable:

```
height = 8;
length = 12;
width = 10;
volume = height * length * width;
/* volume is now 960 */
```

- The right side of an assignment can be a formula (or ***expression,*** in C terminology) involving constants, variables, and operators.
- The initial value of a variable may be included in its declaration:

```
int height = 8;
```

The value 8 is said to be an ***initializer.***

- Any number of variables can be initialized in the same declaration:

```
int height = 8, length = 12;
```

- Each variable requires its own initializer.

```
int height, length, width = 10;
  /* initializes only width */
```

Identifiers

- Names for variables, functions, macros, and other entities are called *identifiers.*
- An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore:

```
times10  get_next_char  _done
```

- Examples of illegal identifiers:

```
10times  get-next-char
```

- C is *case-sensitive:* it distinguishes between upper-case and lower-case letters in identifiers.
- For example, the following identifiers are all different:

```
job  joB  jOb  jOB  Job  JoB  JOb  JOB
```

- C places no limit on the maximum length of an identifier.
- The following ***keywords*** can't be used as identifiers:

```
auto      enum      restrict*  unsigned
break     extern    return     void
case      float     short      volatile
char      for       signed     while
const     goto      sizeof     _Bool*
continue  if        static     _Complex*
default   inline*   struct     _Imaginary*
do        int       switch
double    long      typedef
else      register  union
```

* indicates C99 only

Chapter 3 Formatted Input/Output

- The printf function must be supplied with a *format string*, followed by any values that are to be inserted into the string during printing:

```
printf(format string, expr1, expr2,…);
```

- The format string may contain both ordinary characters and *conversion specifications*, which begin with the % character.
- A conversion specification is a placeholder representing a value to be filled in during printing.
  - %d is used for int values
  - %f is used for float values
- Ordinary characters in a format string are printed as they appear in the string; conversion specifications are replaced.
- Example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i =%d, j= %d, x= %f, y=%f\n",
       i, j, x, y);
```

- Output:

```
i =10, j= 20, x= 43.289200, y=5527.000000
```

- Compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items.
- Too many conversion specifications:

```
printf("%d %d\n", i); /** WRONG **/
```

- Too few conversion specifications:

```
printf("%d\n", i, j); /** WRONG **/
```

- Compilers aren't required to check that a conversion specification is appropriate.
- If the programmer uses an incorrect specification, the program will produce meaningless output:

```
printf("%f %d\n", i, x);/** WRONG **/
```