Introduction to Computers and Programming

Lecture 2 – Formatted I/O
Chap 2 & 3

Tien-Fu Chen

Dept. of Computer Science and Information Engineering

National Yang Ming Chiao Tung Univ.

Recap about 1st program in LAB 0

```
1  #define _CRT_SECURE_NO_WARNINGS
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8      int studentID;
9
10     printf("Enter your student ID: ");
11     scanf("%d", &studentID);
12
13     printf("Hi, %d\n", studentID);
14
15     system("pause");
16
17     return 0;
18 }
```

為了能正常使用scanf開頭輸入 #define _CRT_SECURE_NO_WARNINGS

Include stdio與stdlib兩個library

main()裡面是你要執行的程式碼

宣告變數

印出提示訊息

讀入鍵盤輸入

印出結果

為了不讓新視窗執行完畢後馬上關閉

How a program is generated and executed

A computer program

- A computer program is a sequence of instructions to be executed by computers.
- Examples of computer programs in various forms:

```
0001 1001
1001 1110
1000 1011
1100 1011
1110 0010
1001 0111
1100 1011
1110 0010
1001 0111
1100 1011
```

Machine instructions

```
MOV    AX,10
SUB    BX,AX
MOV    [DX],AX
JMP    200
MOV    CX,5
MOV    AX,10
MUL    AX,CX
CMP    BX,AX
JLE    500
JMP    400
```

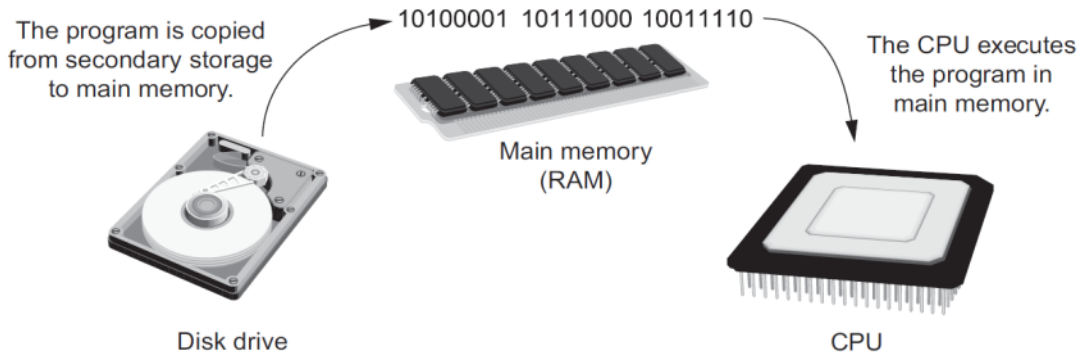
assembly language

```
sum = 0
for (i=0; i<100; i++)
  sum = sum + i
```

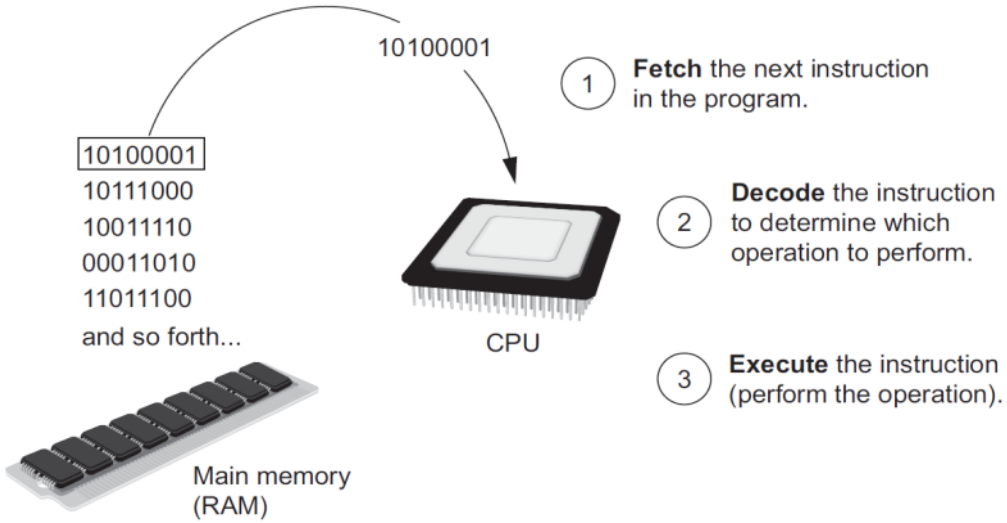
C language

How is a program executed?

1. program is copied into main memory

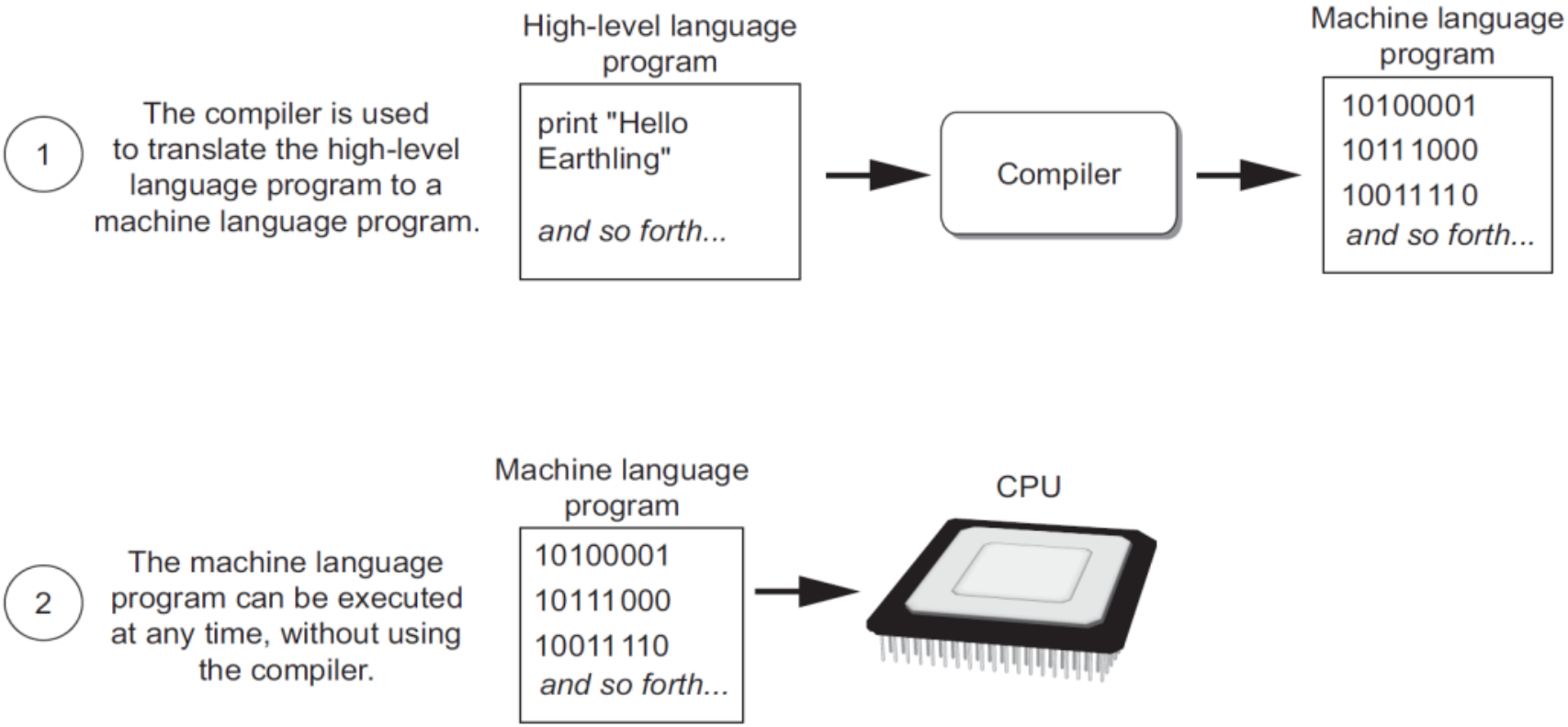


2. CPU executes the instructions by three steps



How program is generated – by compiler

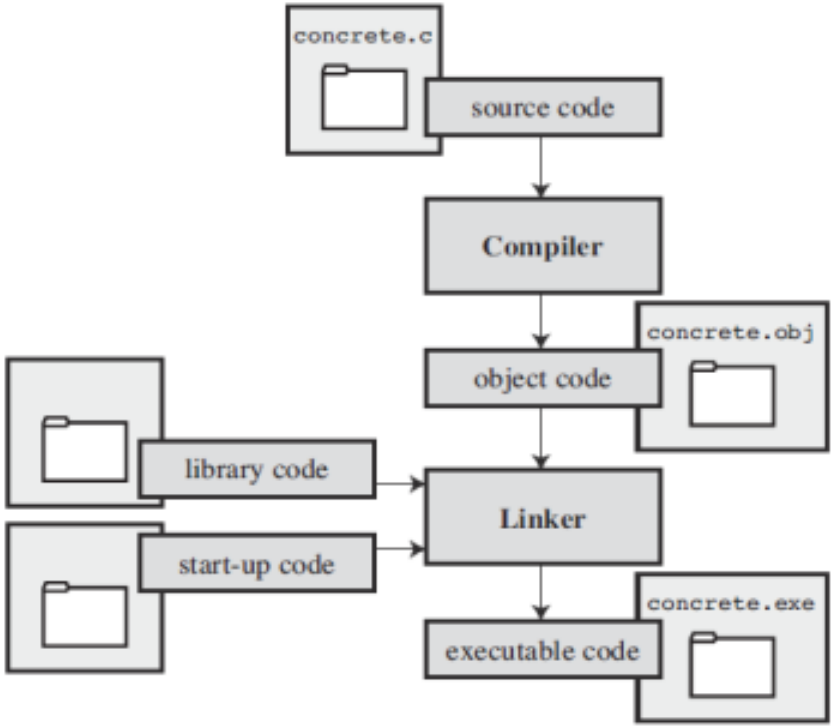
- A compiler is a program that translates a high-level language program into a separate machine language program.



Compiling and Linking

- Three steps to generate programs:
  - ***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.



C Fundamentals

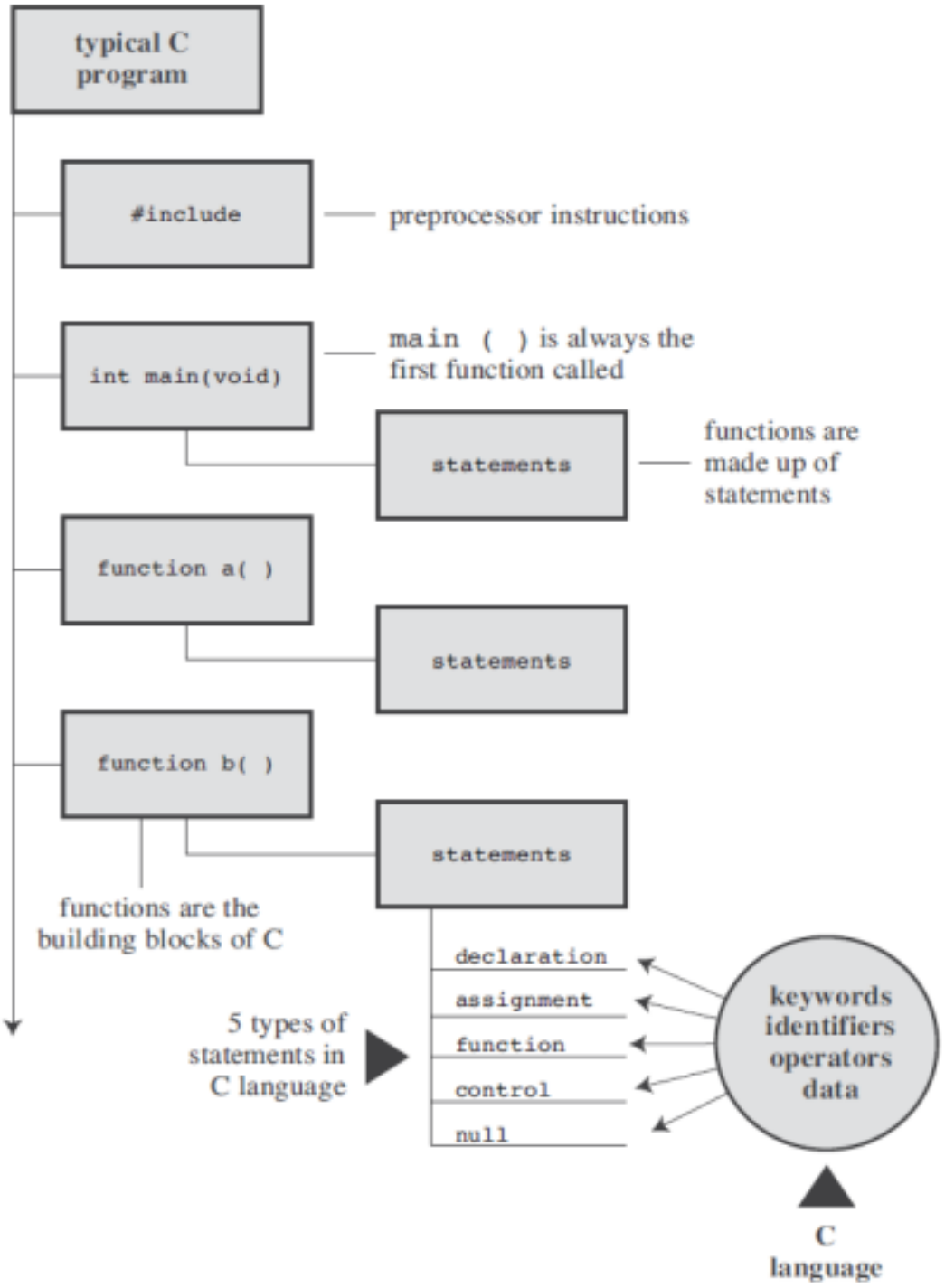
General Form of a Simple Program

- Even the simplest C programs rely on three key language features:
  - Directives
  - Functions
  - Statements

```
#include <stdio.h>

int main(void)
{
  printf("Hi! I'm Lin\n");
}
```

Anatomy of a C program



Directives

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

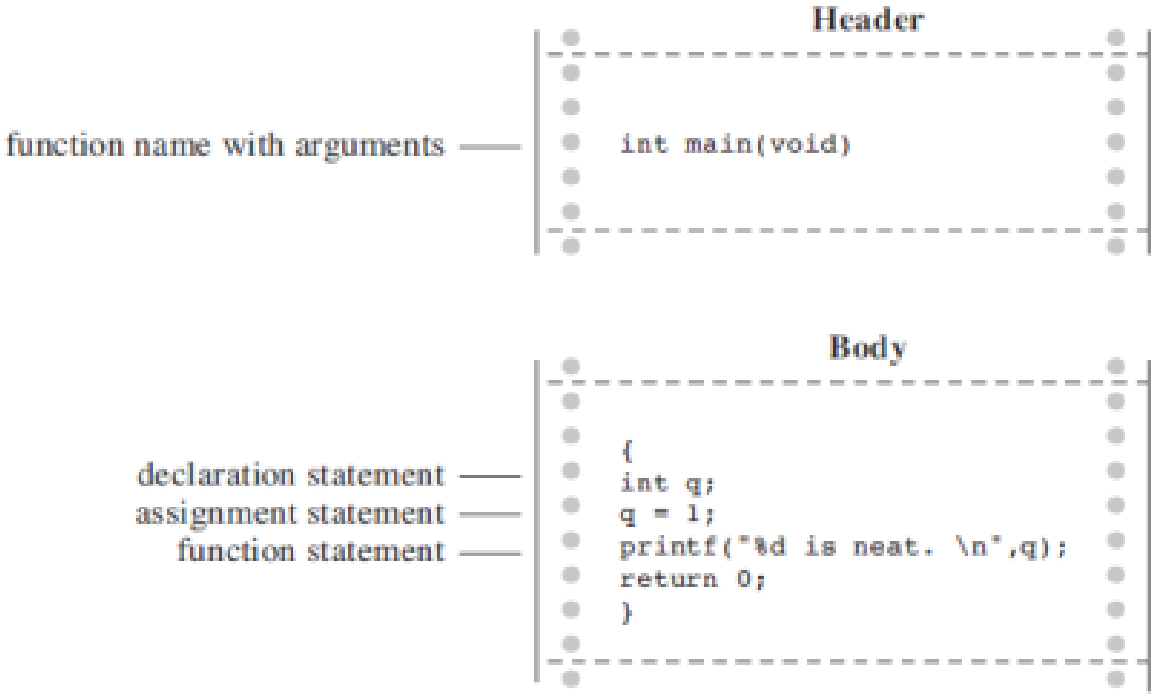
```
#include <stdio.h>
```

- `<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":



The `main` Function

- The `main` function is mandatory.
- `main` is special: 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.
- `pun.c` 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: that is the question.\n");
```

- C requires that each statement end with a semicolon.
  - There's one exception: the compound statement.

Printing Strings

- The statement

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

  could be replaced by two calls of `printf`:

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

- The new-line character can appear more than once in a string literal:

```
printf("Brevity is the soul of wit.\n  --Shakespeare\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.
- Comments may extend over more than one line.

```
/* Name: pun.c
   Purpose: Prints a bad pun.
   Author: K. N. King */
```

Comments

- *Warning:* Forgetting to terminate a comment may cause the compiler to ignore part of your program:

```
printf("My ");     /* forgot to close this
comment...
printf("cat ");
printf("has ");    /* so it ends here */
printf("fleas");
```

Comments in C99

- 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.

More diving in C program

Declarations

- Variables must be ***declared*** before they are used.
- Variables can be declared one at a time:

```
int height;
float profit;
```

- Alternatively, several can be declared at the same time:

```
int height, length, width, volume;
float profit, loss;
```

Declarations

- When `main` contains declarations, these must precede statements:

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

- In C99, declarations don't have to come before statements.

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`.
- A variable of type `int` (short for *integer*) can store a whole number such as 0, 1, 392, or –2553.
  - The largest `int` value is typically 2,147,483,647 but can be as small as 32,767.

Types

- A variable of type `float` (short for *floating-point*) can store much larger numbers than an `int` variable.
- Also, a `float` variable can store numbers with digits after the decimal point, like 379.125.
- Drawbacks of `float` variables:
  - Slower arithmetic
  - Approximate nature of `float` values

Initialization

- Some variables are automatically set to zero when a program begins to execute, but most are not.
- A variable that doesn't have a default value and hasn't yet been assigned a value by the program is said to be ***uninitialized.***
- Attempting to access the value of an uninitialized variable may yield an unpredictable result.
- With some compilers, worse behavior—even a program crash—may occur.

Initialization

- 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, width = 10;
```

- Each variable requires its own initializer.

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

Defining Names for Constants

- `dweight.c` and `dweight2.c` rely on the constant 166, whose meaning may not be clear to someone reading the program.
- Using a feature known as ***macro definition,*** we can name this constant:

```
#define INCHES_PER_POUND 166
```

Defining Names for Constants

- When a program is compiled, the preprocessor replaces each macro by the value that it represents.
- During preprocessing, the statement

```
weight = (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND;
```

will become

```
weight = (volume + 166 - 1) / 166;
```

Defining Names for Constants

- The value of a macro can be an expression:

```
#define RECIPROCAL_OF_PI (1.0f / 3.14159f)
```

- If it contains operators, the expression should be enclosed in parentheses.
- Using only upper-case letters in macro names is a common convention.

Assignment

- A variable can be given a value by means of ***assignment:***

```
height = 8;
```

  The number `8` is said to be a ***constant.***

- Before a variable can be assigned a value—or used in any other way—it must first be declared.

Assignment

- A constant assigned to a `float` variable usually contains a decimal point:

```
profit = 2150.48;
```

- It's best to append the letter `f` to a floating-point constant if it is assigned to a `float` variable:

```
profit = 2150.48f;
```

  Failing to include the `f` may cause a warning from the compiler.

Assignment

- An `int` variable is normally assigned a value of type `int`, and a `float` variable is normally assigned a value of type `float`.
- Mixing types (such as assigning an `int` value to a `float` variable or assigning a `float` value to an `int` variable) is possible but not always safe.

Assignment

- 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.

Program example: Computing the Dimensional Weight of a Box `dweight2.c`

```
/* Computes the dimensional weight of a box from input provided by the user */

#include <stdio.h>

int main(void)
{
  int height, length, width, volume, weight;

  printf("Enter height of box: ");
  scanf("%d", &height);
  printf("Enter length of box: ");
  scanf("%d", &length);
  printf("Enter width of box: ");
  scanf("%d", &width);
  volume = height * length * width;
  weight = (volume + 165) / 166;

  printf("Volume (cubic inches): %d\n", volume);
  printf("Dimensional weight (pounds): %d\n", weight);

  return 0;
}
```

Program results

- Sample output of program:

```
Enter height of box: 8
Enter length of box: 12
Enter width of box: 10
Volume (cubic inches): 960
Dimensional weight (pounds): 6
```

- Note that a prompt shouldn't end with a new-line character.

Layout of a C Program

- A C program is a series of ***tokens.***
- Tokens include:
  - Identifiers
  - Keywords
  - Operators
  - Punctuation
  - Constants
  - String literals

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
```

  It's usually best to avoid identifiers that begin with an underscore.

- Examples of illegal identifiers:

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

Identifiers

- 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
```

Identifiers

- Many programmers use only lower-case letters in identifiers (other than macros), with underscores inserted for legibility:

```
symbol_table  current_page  name_and_address
```

- Other programmers use an upper-case letter to begin each word within an identifier:

```
symbolTable  currentPage  nameAndAddress
```

- C places no limit on the maximum length of an identifier.

Keywords

- 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
```

*C99 only

Keywords

- Keywords (with the exception of `_Bool`, `_Complex`, and `_Imaginary`) must be written using only lower-case letters.
- Names of library functions (e.g., `printf`) are also lower-case.

Formating a C Program

- The statement

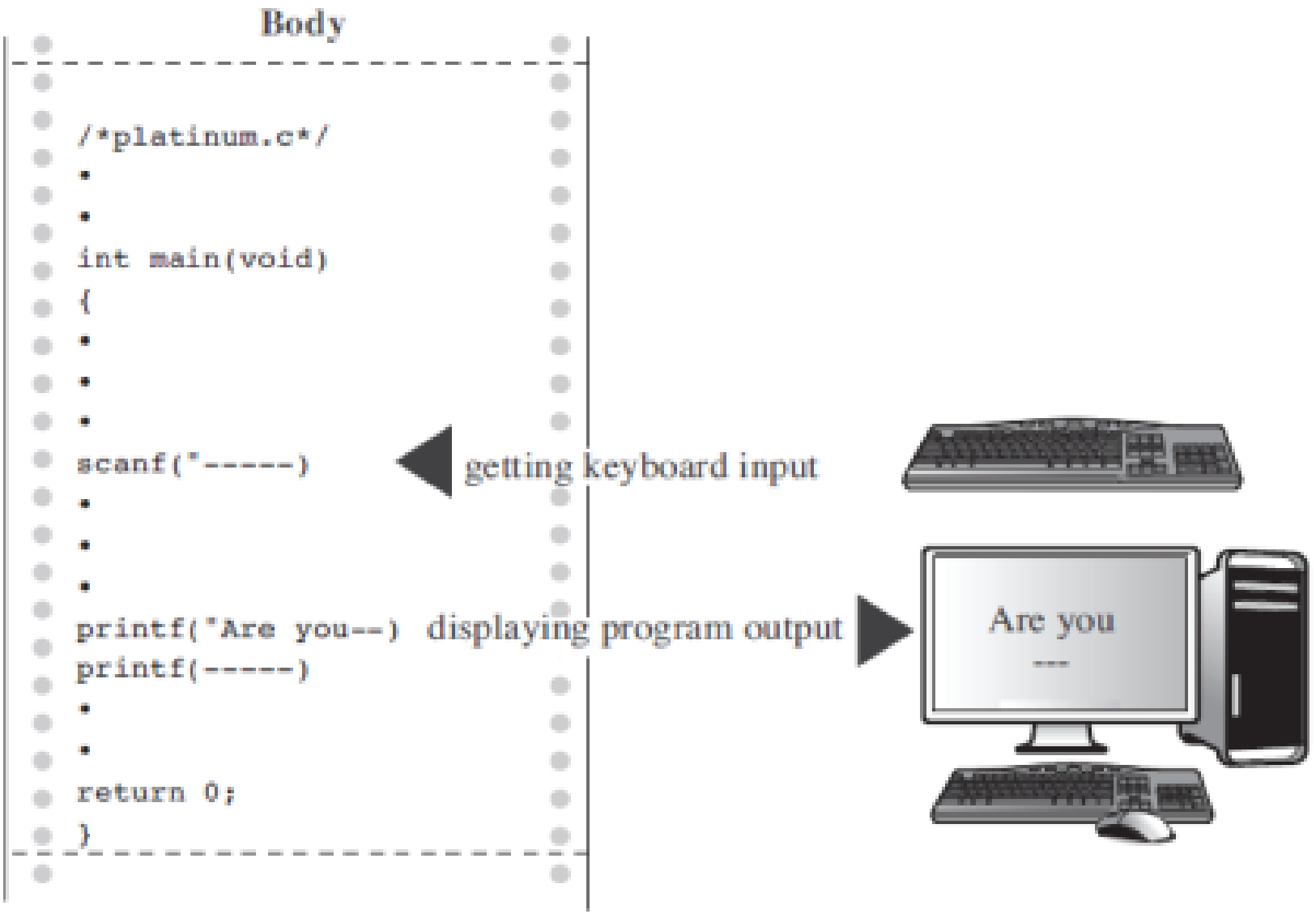
```
printf("Height: %d\n", height);
```

consists of seven tokens:

`printf`	Identifier
`(`	Punctuation
`"Height: %d\n"`	String literal
`,`	Punctuation
`height`	Identifier
`)`	Punctuation
`;`	Punctuation

Formatted Input/Output

Overview of scanf() and printf() functions



The `printf` Function

- 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(`*string*`, `*expr*`1, `*expr*`2, …);`

- 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

The `printf` Function

- 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
```

The `printf` Function

- 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 ***/
```

Conversion Specifications

- A conversion specification can have the form `%`*m*`.`*p*X or `%-`*m*`.`*p*X, where *m* and *p* are integer constants and *X* is a letter.
- Both *m* and *p* are optional; if *p* is omitted, the period that separates *m* and *p* is also dropped.

- `%10.2f`, *m* is 10, *p* is 2, and *X* is `f`.
- `%10f`, *m* is 10 and *p* (along with the period) is missing, but in the specification `%.2f`, *p* is 2 and *m* is missing.

Conversion Specifications

- The ***minimum field width,*** *m*, specifies the minimum number of characters to print.
- If the value to be printed requires fewer than *m* characters, it is right-justified within the field.
  - `%4d` displays the number 123 as `•123`. (• represents the space character.)
- If the value to be printed requires more than *m* characters, the field width automatically expands to the necessary size.
- Putting a minus sign in front of *m* causes left justification.
  - The specification `%-4d` would display 123 as `123•`.

Conversion Specifications

- The meaning of the ***precision,*** *p*, depends on the choice of *X*, the ***conversion specifier.***
- The `d` specifier is used to display an integer in decimal form.
  - *p* indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary).
  - If *p* is omitted, it is assumed to be 1.

Conversion Specifications

- Conversion specifiers for floating-point numbers:

  `e` — Exponential format. *p* indicates how many digits should appear after the decimal point (the default is 6). If *p* is 0, no decimal point is displayed.

  `f` — "Fixed decimal" format. *p* has the same meaning as for the `e` specifier.

  `g` — Either exponential format or fixed decimal format, depending on the number's size. *p* indicates the maximum number of significant digits to be displayed. The `g` conversion won't show trailing zeros. If the number has no digits after the decimal point, `g` doesn't display the decimal point.

tprintf.c

```
  /* Prints int and float values in various formats */
#include <stdio.h>

  int main(void)
  {
    int i;
    float x;

    i = 40;
    x = 839.21f;

    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
    printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

    return 0;
  }
```

- Output:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21    |
```

Escape Sequences

- The `\n` code that used in format strings is called an ***escape sequence.***
- Escape sequences enable strings to contain nonprinting (control) characters and characters that have a special meaning (such as ").
- A partial list of escape sequences:

Alert (bell)	`\a`
Backspace	`\b`
New line	`\n`
Horizontal tab	`\t`

Escape Sequences

- A string may contain any number of escape sequences:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

- Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

Escape Sequences

- Another common escape sequence is \", which represents the " character:

```
printf("\"Hello!\"");
  /* prints "Hello!" */
```

- To print a single \ character, put two \ characters in the string:

```
printf("\\");
  /* prints one \ character */
```

The `scanf` Function

- In many cases, a `scanf` format string will contain only conversion specifications:

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

scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Sample input:

```
1 -20 .3 -4.0e3
```

`scanf` will assign 1, –20, 0.3, and –4000.0 to `i`, `j`, `x`, and `y`, respectively.

How `scanf` Works

- `scanf` tries to match groups of input characters with conversion specifications in the format string.
- For each conversion specification, `scanf` tries to locate an item of the appropriate type in the input data, skipping blank space if necessary.
- `scanf` then reads the item, stopping when it reaches a character that can't belong to the item.
  - If the item was read successfully, `scanf` continues processing the rest of the format string.
  - If not, `scanf` returns immediately.

How `scanf` Works

- As it searches for a number, `scanf` ignores ***white-space characters*** (space, horizontal and vertical tab, form-feed, and new-line).
- A call of `scanf` that reads four numbers:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- The numbers can be on one line or spread over several lines:

```
   1
-20   .3
   -4.0e3
```

- `scanf` sees a stream of characters (¤ represents new-line):

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr
```

(`s` = skipped; `r` = read)

- `scanf` "peeks" at the final new-line without reading it.

How `scanf` Works

- When asked to read an integer, `scanf` first searches for a digit, a plus sign, or a minus sign; it then reads digits until it reaches a nondigit.
- When asked to read a floating-point number, `scanf` looks for
  - a plus or minus sign (optional), followed by
  - digits (possibly containing a decimal point), followed by
  - an exponent (optional). An exponent consists of the letter `e` (or `E`), an optional sign, and one or more digits.
- `%e`, `%f`, and `%g` are interchangeable when used with `scanf`.

How `scanf` Works

- Sample input:

```
1-20.3-4.0e3¤
```

- The call of `scanf` is the same as before:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Here's how `scanf` would process the new input:
  - `%d`. Stores 1 into `i` and puts the – character back.
  - `%d`. Stores –20 into `j` and puts the `.` character back.
  - `%f`. Stores 0.3 into `x` and puts the – character back.
  - `%f`. Stores –4.0 × 103 into `y` and puts the new-line character back.

Ordinary Characters in Format Strings

- Examples:
  - If the format string is "`%d/%d`" and the input is `•5/•96`, `scanf` succeeds.
  - If the input is `•5•/•96` , `scanf` fails, because the `/` in the format string doesn't match the space in the input.
- To allow spaces after the first number, use the format string "`%d /%d`" instead.

Confusing `printf` with `scanf`

- Incorrectly assuming that `scanf` format strings should resemble `printf` format strings is another common error.
- Consider the following call of `scanf`:

```
scanf("%d, %d", &i, &j);
```

  - `scanf` will first look for an integer in the input, which it stores in the variable `i`.
  - `scanf` will then try to match a comma with the next input character.
  - If the next input character is a space, not a comma, `scanf` will terminate without reading a value for `j`.

Program: Adding Fractions

- The `addfrac.c` program prompts the user to enter two fractions and then displays their sum.
- Sample program output:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

addfrac.c

```
/* Adds two fractions */
#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Enter first fraction: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Enter second fraction: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 *denom1;
  result_denom = denom1 * denom2;
  printf("The sum is %d/%d\n",result_num, result_denom)

  return 0;
}
```