C Fundamentals & Formatted Input/Output

adopted from KNK C Programming : A Modern Approach

C Fundamentals

Program: Printing a Pun

- The file name doesn't matter, but the `.c` extension is often required.
  - for example: pun.c

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

int main(void)     // function
{                  /* statements begin */
  printf("To C, or not to C: that is the question.\n");
  return 0;
}                  /* statements end */
```

- to compile

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

or

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

- to run

```
% ./pun
```

The General Form of a Simple Program

Three key language features of C programs

1 *directives* Example: `#include <stdio.h>`

2
```
int main(void)
{                    begin
```
3
```
    statements
}                    end
```

The General Form: Directives

Three key language features of C programs

1 *directives* Example: `#include <stdio.h>`

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

- `<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.

The General Form: Directives

Three key language features of C programs

2

```
int main(void)
{
    # of statements
    return x + 1;
}
```

a series of statements that have been grouped together and given a name.

A function that computes a value uses a `return` statement to specify what value it "returns":

- ***Library functions*** are provided as part of the C implementation.
- 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.

The General Form: Statements

Three key language features of C programs

3	*statements*	a command to be executed when the program runs. C requires that each statement end with a semicolon.

- `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");
```

Printing Strings: `printf`

- 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.

Same effect

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

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

```
printf("Brevity is the soul of wit.\n  --Shakespeare\n");
```

more on `printf` on following section

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 */
```

- In C99, comments can also be written in the following way:

```
// A comment, which 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.***

- To use variables and assignments, you need to know
  1. type
  2. declaration
  3. initialization

```
1     2           3
 int  height  =  183;
```

Variables and Assignment: Types

- 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.

- 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

Variables and Assignment: Declarations

- Variables must be ***declared*** before they are used.

- Both are legal

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

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

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

Variables and Assignment: Assignment (1/2)

- 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.
- 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.

Variables and Assignment: Assignment (2/2)

- 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) is **possible but not always safe**.

- 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.

Variables and Assignment: 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.***

- Accessing the value of an uninitialized variable causes an unpredictable result.
  - With some compilers, worse behavior—even **a program crash**—may occur.

- The initial value of a variable may be included in its declaration:

```
int height = 8;      // The value 8 is said to be an initializer.
int height = 8, length = 12, width = 10;
int height, length, width = 10; // initializes only width
```

Printing the Value of a Variable

- `printf` can be used to display the current value of a variable.
- To write the following message, we'd use the following call of `printf`:

```
Height: h
```

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

- `%d` is a placeholder indicating where the value of `height` is to be filled in.

- There's no limit to the number of variables that can be printed by a single call of `printf`:

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

Details of `printf` on following section

Reading Input

- `scanf` is the C library’s counterpart to `printf`.
- `scanf` requires a ***format string*** to specify the appearance of the input data.

- Example of using `scanf` to read an `int` value:

```
scanf("%d", &i);       /* reads an integer; stores into i */
```

- The `&` symbol is usually (but not always) required when using `scanf`.

Details of `scanf` on following section

Defining Names for Constants

- Using a feature known as ***macro definition,*** we can name this constant:

```
#define INCHES_PER_POUND 166
```

It is common convention to use only upper-case letters in macro names

- 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;
```

- 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.

Identifiers

- ***Identifiers:*** Names for variables, functions, macros, and other entities
  - Letters, digits, and underscores, but must begin with a letter or underscore:

```
times10   _done          symbol_table   current_page
                         symbolTable    currentPage
```

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

- Examples of illegal identifiers:

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

- C is ***case-sensitive*** `job   joB   jOb   jOB   Job   JoB   JOb   JOB`

  all are different

Keywords

- The following ***keywords*** can’t be used as identifiers:

`auto`	`do`	`goto`	`return`	`typedef`	`inline*`
`break`	`double`	`if`	`short`	`union`	`restrict*`
`case`	`else`	`int`	`signed`	`unsigned`	`_Bool*`
`char`	`enum`	`long`	`sizeof`	`void`	`_Complex*`
`const`	`extern`	`register`	`static`	`volatile`	`_Imaginary*`
`continue`	`float`		`struct`	`while`	*C99 only
`default`	`for`		`switch`		

- 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.

Layout of a C Program (1/2)

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

- The statement

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

consists of seven tokens:

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

Layout of a C Program (2/2)

- *Statements can be divided* over any number of lines.
- *Space between tokens* (such as before and after each operator, and after each comma) makes it easier for the eye to separate them.
- *Indentation* can make nesting easier to spot.
- *Blank lines* can divide a program into logical units

Formatted Input and Output

The `printf` Function (1/3)

- 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*`,` *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 (2/3)

- 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);
```

Result:

The `printf` Function (3/3)

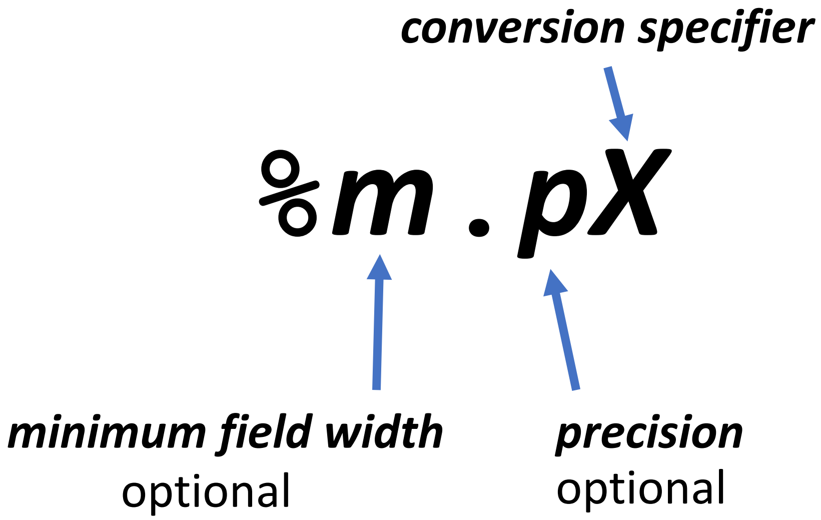
- Compilers aren’t required to check that the number of conversion specifications in a format string matches the number of output items.

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

- Compilers aren’t required to check that a conversion specification is appropriate.
  - An incorrect specification will produce meaningless output:

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

Conversion Specifications (1/2)



`%d`	–	Integer
`%e`	–	Exponential format
`%f`	–	Fixed decimal
`%g`	–	Either exponential format or fixed decimal format

12345.6789

```
%5.3f   12345.678
%8d        12345
%-8d    12345
```

Conversion Specifications (1/2)

Format specifier	Description	Supported data types

`%c`	Character	`char`
		`unsigned char`

`%d`	Signed Integer	`short`
		`unsigned short`
		`int`
		`long`

`%e or %E`	Scientific notation of float values	`float` `double`
`%f`	Floating point	`float`
`%g or %G`	Similar as %e or %E	`float` `double`
`%hi`	Signed Integer(Short)	`short`
`%hu`	Unsigned Integer(Short)	`unsigned short`

`%i`	Signed Integer	`short`
		`unsigned short`
		`int`
		`long`

`%l or %ld or %li`	Signed Integer	`long`
`%lf`	Floating point	`double`
`%Lf`	Floating point	`long double`

`%lu`	Unsigned integer	`unsigned int`
		`unsigned long`

Format specifier	Description	Supported data types
`%lli, %lld`	Signed Integer	`long long`
`%llu`	Unsigned Integer	`unsigned long long`

`%o`	Octal representation of Integer.	`short`
		`unsigned short`
		`int`
		`unsigned int`
		`long`

`%p`	Address of pointer to void void *	`void *`
`%s`	String	`char *`

`%u`	Unsigned Integer	`unsigned int`
		`unsigned long`

`%x or %X`	Hexadecimal representation of Unsigned Integer	`short`
		`unsigned short`
		`int`
		`unsigned int`
		`long`

`%n`	Prints nothing	
`%%`	Prints % character	

Escape Sequences (1/2)

- 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`

- A string may contain any number of escape sequences:

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

```
Item    Unit    Purchase
        Price   Date
```

Escape Sequences (2/2)

- A string may contain any number of escape sequences:

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

```
Item    Unit    Purchase
        Price   Date
```

- 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

- `scanf` reads input according to a particular format.

  `scanf(`*format_string*`, &`*`var1`*`, &`*`var2`*`, …);`

- A `scanf` format string may contain both ***ordinary characters*** and ***conversion specifications.***
- The conversions allowed with `scanf` are essentially the same as those used with `printf`.

- 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 (1/4)

- `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 (2/4)

- 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
```

$\Rightarrow$

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

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

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

How `scanf` Works (3/4)

- 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`.

- When `scanf` encounters a character that can’t be part of the current item, the character is “put back” to be read again during the scanning of the next input item or during the next call of `scanf`.

How `scanf` Works (4/4)

- 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

- When it encounters one or more white-space characters in a format string, `scanf` reads white-space characters from the input until it reaches a non-white-space character (which is “put back”).
- When it encounters a non-white-space character in a format string, `scanf` compares it with the next input character.
  - If they match, `scanf` discards the input character and continues processing the format string.
  - If they don’t match, `scanf` puts the offending character back into the input, then aborts.
- 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` (1/2)

- Although calls of `scanf` and `printf` may appear similar, there are significant differences between the two.
- One common mistake is to put `&` in front of variables in a call of `printf`:

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

- 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`.

Confusing `printf` with `scanf` (2/2)

- Putting a new-line character at the end of a `scanf` format string is usually a bad idea.
- To `scanf`, a new-line character in a format string is equivalent to a space; both cause `scanf` to advance to the next non-white-space character.
- If the format string is `"%d\n"`, `scanf` will skip white space, read an integer, then skip to the next non-white-space character.
- A format string like this can cause an interactive program to “hang.”