Introducing C

Lecture 18

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Printing the Value of a Variable

- `printf` can be used to display the current value of a variable.
- To write the message

```
int height = 10;
printf("Height: %d\n", height);
```

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

Printing the Value of a Variable

- **`%d`** works only for **`int`** variables; to print a **`float`** variable, use **`%f`** instead.
- By default, **`%f`** displays a number with six digits after the decimal point.
- To force `%f` to display *p* digits after the decimal point, put ***`.p`*** between `%` and `f`.
- To print the line

  ```
  Profit: $2150.48
  ```

  use the following call of `printf`:

  ```
  printf("Profit: $%.2f\n", profit);
  ```

Printing the Value of a Variable

- 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);
```

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 */
```

Printing Expressions

- `printf` can display the value of any **numeric expression**.
- The statements

```
volume = height * length * width;
printf("%d\n", volume);
```

could be replaced by

```
printf("%d\n", height * length * width);
```

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:

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

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

Reading Input

- Reading a **`float`** value requires a slightly different call of `scanf`:

```
scanf("%f", &x);
```

- **"`%f`"** tells `scanf` to look for an input value in `float` format (the number may contain a decimal point, but doesn't have to).

Defining Names for Constants

- 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;
```

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
```

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**

Layout of a C Program

- The whole program can't be put on one line, because each preprocessing directive requires a separate line.
- Compressing programs in this fashion isn't a good idea.
- In fact, adding spaces and blank lines to a program can make it easier to read and understand.

Layout of a C Program

- C allows any amount of space—blanks, tabs, and new-line characters—between tokens.
- Consequences for program layout:
  - *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.

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, 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

The `printf` Function

- **Ordinary characters** in a format string are printed as they appear in the string; **conversion specifications** are **replaced**.
- Example (01.c):

```
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: (02.c)

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

- Too few conversion specifications: (03.c)

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

The `printf` Function

- 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**: (04.c)

```
int y = 10;

float x = 3.14f;
…
printf("%d %f\n", x, y); /*** WRONG!!! ***/
```

Conversion Specifications

- A conversion specification can have the form ***%m . pX*** or ***%−m . pX***, 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.

Conversion Specifications

- *%m.pX* **or** *%-m.pX*
- 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•`.
- Example: 05.c

Conversion Specifications

- `%`*m* . ***p***`X` **or** `%-`*m* . ***p***`X`
- 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.
  - Example: 06.c

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` — Display a floating-point number in either exponential format or fixed decimal format, depending on the number's size. *p* indicates the maximum number of significant digits.

- Example: 07.c

Program: Using `printf` to Format Numbers

- The `tprintf.c` program uses `printf` to display integers and floating-point numbers in various formats.

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|\n", x, x);

  return 0;
}
```

- Output:

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

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

- Double quotation mark: `"`
- Another common escape sequence is `\"`, which represents the `"` character: (08.c)

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

- Backslash: `\`
- 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.
  - "pattern-matching" function that tries to match up groups of input characters with conversion specifications
- A `scanf` format string may contain both ordinary characters and conversion specifications.

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

- The conversions specifications allowed with `scanf` are essentially the same as those used with `printf`.

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 -4.25
```

`scanf` will assign 1, –20, 0.300000, and –4.250000 to `i`, `j`, `x`, and `y`, respectively.

The `scanf` Function

- When using `scanf`, the programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable.

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

- **`&`** symbol, which normally precedes each variable in a `scanf` call.
- The **`&`** is usually (but not always) required, and it's the programmer's responsibility to remember to use it.

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 the beginning of 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
   -4.25
```

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` and `%f` are interchangeable when used with `scanf`.

How `scanf` Works

- 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

- Sample input:

```
1-20.3-4.25
```

- 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 –4.25 into `y` and puts the new-line character back.

How `scanf` Works

- Sample input: (09.c)

```
1-20.3-4.25
```

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

```
scanf("%d%d%f%d", &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.
  - `%d`. Stores –4 into `y` and puts the point back.