ITC213: STRUCTURED PROGRAMMING

Bhaskar Shrestha
National College of Computer Studies
Tribhuvan University

Lecture 07: Data Input and Output

Readings: Chapter 4

Input /Output Operations

- A program needs to read data in variable names: ***input operation***
- Also data stored in variables need to be displayed: ***output operation***
- C language does not define any keyword to perform these input/output operations
- C language provides functions to perform input/output operations
  - `getchar`, `putchar`, `scanf`, `printf`, `gets` and `puts`
- All these functions are prototyped under `stdio.h`

Reading a Single Character through `getchar`

- The `getchar` function reads the next character available from the standard input device (typically a keyboard) and returns the character read
  - Returns `EOF` when it encounters end of file
- The character returned can be assigned to a variable for further use
- In general terms, a function reference would be written as
  `variable_name = getchar();`
  where *`variable_name`* refers to some previously defined character variable

```
char ch;
ch = getchar();
```

Writing a Single Character through `putchar`

- The `putchar` function can be used to write a single character to a standard output device (typically a monitor)
- It takes the form as shown below:
  `putchar(`*`character_value`*`)`
  where `character_value` can be a character variable or an expression that evaluates to a character value that is going to be printed

```
ch1 = getchar();
ch2 = ch1 - 'a' + 'A';
putchar(ch2);
putchar(ch1 - 'a' + 'A');
putchar('\n');
```

Formatted Output Function: `printf`

- Can be used to output data of different types in different formats
- Can perform rounding, aligning columns, right/left justification, inserting literal characters, exponential format, hexadecimal format, and fixed width and precision
- The general from of an output function is:
  - `printf(`*`format-string`*`, `*`arg1`*`, `*`arg2`*`, ...., `*`argn`*`)`
    Here, *`format-string`* is a string and *`arg1`*, *`arg2`*, `...`, *`argn`* are any valid C expressions

Formatted Output Function: `printf`

- format-string describes the output format
- *`format-string`* consists of two types of items
  - ordinary characters, which are exactly printed as written
  - conversion specifications that define the way the subsequent arguments are displayed
- Other-arguments (*`arg1`*, *`arg2`*, `...`, *`argn`*): correspond to each conversion specification in format-string

Example

- There must be exactly the same of number of arguments (after the format string) as there are conversion specifications, and the conversion specifications and the arguments are matched in order from left to right
- The arguments can be any valid expression and they must match the data type as specified in the conversion specification

```
printf("Answer x = %d \n", x);
printf("a = %d, b = %f \n", a, b);
printf("Square of %d is %d\n", a, a*a);
```

Conversion Specifications

- Controls the type and format of the value to be printed
- Each conversion specification begins with a `%` and ends with a ***conversion character***.
- Conversion character tells the data type of the corresponding argument
- Between the `%` and the conversion character there may be, in order:
  - A minus sign, which specifies left adjustment of the converted argument
  - A number that specifies the minimum field width
  - A period, which separates the field width from the precision
  - A number, the precision: exact meaning depends on type printed
  - An `h` if the integer is to be printed as a `short`, or `l` if as a `long`

Conversion Characters

Conversion Character	**Printed as**
`c`	single character
`d, i`	signed decimal integers
`e`	floating-point value in scientific notation (lowercase `e`)
`E`	floating-point value in scientific notation (uppercase `E`)
`f`	floating-point value without an exponent
`g`	uses `e` or `f`, whichever is shorter
`G`	uses `E` or `f`, whichever is shorter
`o`	unsigned octal
`s`	string of characters
`u`	unsigned decimal integers
`x`	unsigned hexadecimal (lowercase letters)
`X`	unsigned hexadecimal (uppercase letters)
`%`	prints a `%` sign

Printing Integers

- Only negative integers are preceded by a `–` sign
- To print short integers place `h` before the conversion character
- To print long integers place `l` before the conversion character listed below

Conversion Character	**Corresponding Argument Printed as**
`d, i`	signed decimal integers
`o`	unsigned octal
`u`	unsigned decimal integers
`x`	unsigned hexadecimal (uses lowercase letters, `a-f`)
`X`	unsigned hexadecimal (uses uppercase letters, `A-F`)

Printing Floating-Point Numbers

- Use the `f` conversion character to print floating-point numbers
  - There is at least one digit to left of decimal
  - 6 digits are printed after decimal point
- Use either `e` or `E` character to print in exponential form
- Using `g` or `G` conversion character tells print to use either `f` or `e` whichever results in shorter output
  - No trailing zeros (`1.2300` becomes `1.23`)
- Precede the conversion character with `l` to print a `long double`

Printing Characters and Strings

- **`c`**: Prints a char argument
  - Cannot be used to print the first character of a string
- **`s`**: prints a string, argument must be a pointer to **`char`**
  - Prints characters until NULL (`'\0'`) encountered
  - Cannot print a `char` argument
- Remember
  - Single quotes for character constants (`'z'`)
  - Double quotes for strings `"z"` (which actually contains two characters, `'z'` and `'\0'`)

Other conversion character

- `p`: Displays pointer value (address)
- `n`: Stores number of characters already output by current `printf` statement
  - Takes a pointer to an integer as an argument
  - Nothing printed by a `%n` specification
- Every `printf` call returns a value
  - Number of characters output
  - Negative number if error occurs
- `%`: Prints a percent sign
  - `%%`

Specifying the Minimum Field Width

- An integer placed between the **%** sign and the conversion character acts as a ***minimum field width***
- It specifies the minimum number of character position to be taken by the output data
  - If width larger than data, output will be filled with leading spaces to reach the field width
  - If output data is larger than width, it will be printed in full
  - If you wish to fill with `0`’s, place a `0` before the field width specifier

```
printf(":%10d:",12);
printf(":%010d:",12);
```

$\longrightarrow$

```
:        12:
:0000000012:
```

The Precision Specifier

- It consists of a period followed by an integer
- Its exact meaning depends upon the type of data to which it is applied
  - When applied to `%f`, `%e`, `%g`, it determines the number of decimal places displayed
  - When applied to `%s`, it specifies the maximum field length
  - When applied to integers, it determines the minimum number of digits that will appear for each number

```
printf("%5.2f\n", 12345.267);
printf("%3.8d\n", 1002);
printf("%10.15s\n", "This is a sample text");
```

Justifying Output

- If the field width is larger than the data printed, the data will be placed on the right edge of the field
- You can force output to be left justified by placing a minus sign directly after the **`%`**
- For example, **`%-10.2f`** left justifies a floating-point number with two decimal places in a 10-character field

```
printf(".........................|\n");
printf("right-justified: %8d|\n", 100);
printf(" left-justified: %-8d|\n", 100);
```

```
.........................|
right-justified:      100|
 left-justified: 100     |
```

Flags in Conversion specification

- A flag is placed immediately after % specifier
- `-`, `0`: discussed earlier
- `+`: A sign (either `+` or `-`) will precede each numerical data
- `#` (with `o` and `x` type conversion):
  - Causes octal and hexadecimal values to be preceded by `0` and `0x` respectively
- `#` (with `f`, `e` and `g` type conversion):
  - Causes a decimal point to be present in all floating point numbers
  - Prevents truncation of trailing zeros in g type conversion
- Space: Prints a space before a positive value not printed with + flag

Formatted Input Function: `scanf`

- General-purpose console input routine
- Can read all the built-in data types and automatically convert numbers into the proper internal format
- The general form of `scanf` is
  `scanf(`*`format-string`*`, `*`arg1`*`, `*`arg2`*`, ..., `*`argn`*`)`
- `scanf` reads characters from the standard input, interprets them according to the specification in *`format-string`*, and stores the results in the remaining arguments *`arg1`*, *`arg2`*, `...`, *`argn`*, each of which must be a pointer that indicate where the corresponding converted input should be stored

Format-string

- The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:
  - Blanks or tabs, which are not ignored
  - Ordinary characters (not `%`), which are expected to match the next non-white space character of the input stream
  - Conversion specifications, consisting of the character `%` followed by a conversion character that determines the type of data to be read next
- A conversion specification directs the conversion of the next input field
- Any character that cannot be interpreted according to the conversion specification terminates the current input field, and is put back into the input buffer. This character is then the first one read for the next input item

Conversion Character

Conversion Character	**Meaning**
`c`	Reads a single character
`d`	Reads a decimal integer
`i`	Reads an integer. The integer may be in octal (with leading 0) or hexadecimal (leading `0x` or `0X`)
`u`	Reads an unsigned decimal integer
`x`	Reads a hexadecimal integer
`o`	Reads a octal integer
`e, f, g`	Reads a floating-point number (`float`)
`s`	Reads a string
`[...]`	Scans for a sets of characters
`p`	Reads an address
`n`	Does not read, stores no of characters read up to the point %n was encountered
`%`	Reads a percent sign

Reading Integers

- To read integers, use either the `%d` or `%i`
  - For `%i` you can enter the number either in decimal, octal or hexadecimal form (`0` must precede an octal integer and `0x` or `0X` must precede an octal form)
- Use `%o` to read an integer in octal form and `%x` or `%X` to read in hexadecimal form
  - No need to precede with `0` or `0x`
- To read unsigned integer, use `%u`
- `scanf` function stops reading a integer when the first nonnumeric character is encountered
- To read a long integer, put an `l` in front of any conversion character mentioned above
- To read a short integer, put an `h` in front of any conversion character

Reading floating-point values

- To read a floating-point number, use `%f`, `%e`, or `%g`
- Inputted number can begin with optional sign, optional decimal point and optional exponent
- To read a `double`-precision number, put `l` before any of the above conversion character
- To read a long `double`, put `L` before any of the above conversion character
- As with integers, `scanf` stops reading when the first invalid character is encountered

Reading single characters

- Use `%c` to read a single character
- Although spaces, tabs and newlines are used as field separates when reading other types of data; when reading a single character, white-space characters are read like any other character
- For example, with input `x y`, this code fragment

  ```
  scanf("%c%c%c", &a, &b, &c);
  ```

  stores `x` in `a`, a space in `b` and the character `y` in `c`

Reading Strings

- Use `%s` to read a string
  - Reads characters until it encounters a white-space character
  - Characters read are put into the character array pointed to by the corresponding argument
  - The result is null terminated
  - Skips leading white-spaces
- For example

```
char str[80];
scanf("%s", str);
```

If input was `This is a test`, `str` only contains `This`

Scanset

- Set of characters enclosed in square brackets `[]`
  - Preceded by `%` sign
- Scans input stream, looking only for characters in scan set
  - Whenever a match occurs, stores character in specified array
  - Stops scanning once a character not in the scan set is found
- Inverted scan sets
  - Use a caret `^`: `[^aeiou]`
  - Causes characters not in the scan set to be stored

Skipping characters

- A white-space character in format-string causes `scanf` to skip one or more white-space characters
- A non-white-space character in format-string causes `scanf` to read and discard matching characters in the input
- Suppressing input
  - An `*` placed before conversion character tells `scanf` to read a field but not assign to any variable
  - In effect, that input field is skipped. Such a conversion specification corresponds to no variable argument

Maximum Field Width Specifier

- An unsigned integer placed between % and conversion character acts as a maximum field width specifier
- It indicates the maximum number of characters to be read and converted
- The next input field begins with the first character not yet processed
- For example
  `scanf("%3d%d", &a, &b);`
  If the input was `12345`, `a` contains `123` and `b` contains `45`

Printing strings with `puts`

- The `puts` function writes its string argument to the screen followed by a newline
- `puts` recognizes the same backslash escape sequence as `printf` does, such as `\t` for tab
- A call to `puts` requires less overhead than the same call to `printf` because puts can only output a string of characters—it cannot output numbers or do format conversions
- Example

```
char str[] = "Hello World";
puts(str);
puts("\tWelcome!!");
```

$\longrightarrow$

```
Hello World
        Welcome
```

Reading strings with `gets`

- The `gets` function reads characters from input and places them into the character array pointed by the given argument
  - Characters are read until a newline or an EOF is received
  - The new line character is not made part of the string; instead it is translated into a null character to terminate the string
  - Unlike using `%s` in `scanf`, `gets` does terminate input on white-space character
- For example

  ```
  char str[80];
  gets(str);
  ```

  If input was `This is a test`, `str` contains the entire string `This is a test`