Formatted Input/Output

The input and output facilities provided by **`printf`** and **`scanf`** are both simple and powerful. You will **NOT** learn these functions by reading and taking notes alone. You **MUST** practice with examples on your own.

Simple Formatted Output

The most common facility used to output information is the **`printf`** function (**`print f`**ormatted). The general form of the function is:

```
int printf(const char *format_string, ...);
```

where:

format_string	A string (character array)
...	An optional list of expressions to print.

Another look at the general form:

```
int printf(string, expression1, expression2, expression3, ...  );
```

`printf` can only print strings (words). If you wish to print a number, you have to convert it to a string first. Fortunately, **`printf`** makes this very easy. Within *string*, there are usually one or more *conversion specifiers* that determine how to print numeric values.

Examples:

```
#include <stdio.h>

int main(void)
{
  int age = 18;
  float gpa = 3.78F;
  double pi = 3.1415926;

  printf("Hello\n");
  printf("John's age is %i\n", age);
  printf("John's age is %d\n", age);
  printf("His GPA is %f\n", gpa);
  printf("The value of PI is %f\n", pi);
  printf("John is %i years old and his GPA is %f\n", age, gpa);

  return 0;
}
```

Output:

```
Hello
John's age is 18
John's age is 18
His GPA is 3.780000
The value of PI is 3.141593
John is 18 years old and his GPA is 3.780000
```

Incorrect usage:

```
printf("John is %i years old and his GPA is %f\n", age);
printf("John is %i years old\n", age, gpa);
printf("John is %f years old\n", age);
printf("His GPA is %i\n", gpa);
```

Output when compiled with gcc:

```
John is 18 years old and his GPA is 0.000000
John is 18 years old
John is -0.000000 years old
His GPA is -1610612736
```

Output from Borland's C compiler:

```
John is 18 years old and his GPA is 0.000000
John is 18 years old
John is +NAN years old
His GPA is -1610612736
```

Output when compiled with Microsoft's C compiler:

```
John is 18 years old and his GPA is 79789135994290289000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000.000000
John is 18 years old
John is 79648294434967839000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000.000000 years old
His GPA is -1610612736
```

Fortunately, the GNU compiler will alert you to these errors:

```
printf("John is %i years old and his GPA is %f\n", age); /* warning: too few arguments for format              */
printf("John is %i years old\n", age, gpa);              /* warning: too many arguments for format             */
printf("John is %f years old\n", age);                   /* warning: double format, different type arg (arg 2) */
printf("His GPA is %i\n", gpa);                          /* warning: int format, double arg (arg 2)            */
```

An example using a larger expression:

```
int a = 3;
int b = 5;

printf("The sum of a and b is %i\n", a + b);
printf("a = %i and b = %i, so 2 * (a + b) - 3 = %i\n", a, b, 2 * (a + b) - 3);

The sum of a and b is 8
a = 3 and b = 5, so 2 * (a + b) - 3 = 13
```

More examples:

```
float gpa = 3.78F;
double pi = 3.1415926;

printf("GPA is %g\n", gpa);
printf("GPA is %e\n", gpa);
printf("PI is %g\n", pi);
printf("PI is %e\n", pi);
```

Output:

```
GPA is 3.78
GPA is 3.780000e+000
PI is 3.14159
PI is 3.141593e+000
```

Common **printf** conversion specifiers:

```
%c - characters
%s - strings (NULL terminated C strings)
%d - integers
%i - integers
%f - floating point (float and double)
%e - scientific notation
%g - floating point (minimum digits)
%p - pointers (displays in hex)
%x - hexadecimal integers
%u - unsigned
%ld, %li - long integers
%lu - unsigned long integers
```

Controlling Size and Precision

The conversion specifiers determine the type of the data to display. If you want to have more control over the output, you need to specify additional information in the format string.

The general form is this:

```
%[flags][width][.precision]type
```

where

flags	Optional characters to control justification and characters used for padding. Default is right justification. A minus sign indicates left justification.
width	Optional number that controls the minimum number of *characters* to output.
.precision	Optional number that controls the whether or not to print a decimal point and how many digits to the right of the point to print. For integers, it specifies the minimum number of *digits* to print.

A few examples will explain this better:

```
int age = 21;
float wt = 165.89F;

printf("|%i|%5i|%-5i|%5.4i|\n", age, age, age, age);
printf("|%f|%10.3f|%10.3e|%-10g|\n", wt, wt, wt, wt);
```

Output:

```
|21|   21|21   | 0021|
|165.889999|   165.890| 1.659e+02|165.89    |
```

Showing with a dot • for each space:

```
|21|•••21|21•••|•0021|
|165.889999|•••165.890|•1.659e+02|165.89••••|
```

Escape Sequences

Certain characters are not printable, meaning nothing is displayed on the screen (or printer) when you output them with **printf**. Some of them are print control codes (special values that the display or printer interpret differently). Common non-printing control characters

```
Value  Sequence        Meaning
-----------------------------------
  7        \a         bell (alarm)
  8        \b          backspace
  9        \t          horiz. tab
 10        \n          line feed
 11        \v          vert. tab
 12        \f          form feed
 13        \r          carriage return
```

Other escape sequences:

```
Sequence     Meaning
-------------------------
  \0      null character
  \"       double quote
  \\       backslash
```

Keep in mind these sequences are considered *single characters*.

The percent sign is special. To display that, you must use two of them: `%%`

Example:

```
printf("\"\\%%\t%%\t%%\\\"\n");
```

Output: (10 characters. There is a TAB character between each percent sign.)

```
"\%     %       %\"
```

Simple Input

A common way to input information is the **`scanf`** function (**`scan f`**ormatted). The general form of the function is very similar to **`printf`**:

```
int scanf(const char *format_string, ...);
```

where:

format_string	A string (character array)
...	A list of memory locations to store the values that were read in.

Another look at the general form:

```
int scanf(string, location1, location2, location3, ...  );
```

and an example. After the value is input into **`age`**, it is displayed by the **`printf`** function:

```
int age;
scanf("%d", &age);
printf("Your age is %d\n", age);
```

Although **`scanf`** looks a lot like **`printf`**, they are different.

- The first thing to note is the special character, &, that is placed before each argument in the list.
- **`printf`** is used to format an entire string of characters, **`scanf`** isn't really formatting a string.
- Often, **`printf`** will end with a newline character; **`scanf`** shouldn't.
- Putting whitespace around the conversion specifiers in a **`scanf`** can lead to different results than without spaces.

Any of these characters are considered whitespace characters:

- space
- newline
- horizontal tab
- vertical tab
- carriage return
- formfeed

Reading in four integers and printing them out to the screen:

```
int a, b, c, d;
scanf("%d%d%d%d", &a, &b, &c, &d);
printf("%d %d %d %d\n", a, b, c, d);
```

Incorrect way to read in the same four integers:

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

Fortunately, most compilers today will recognize the fatal mistake and warn you about it:

```
main.c:30: warning: format argument is not a pointer (arg 2)
main.c:30: warning: format argument is not a pointer (arg 3)
main.c:30: warning: format argument is not a pointer (arg 4)
main.c:30: warning: format argument is not a pointer (arg 5)
```

You might see this with Microsoft's compilers:

```
main.c(30) : warning C4700: local variable 'd' used without having been initialized
main.c(30) : warning C4700: local variable 'c' used without having been initialized
```

```
main.c(30) : warning C4700: local variable 'b' used without having been initialized
main.c(30) : warning C4700: local variable 'a' used without having been initialized
```

Another example:

```
int i1, i2;
float f1, f2;
scanf("%d%f%d%f", &i1, &f1, &i2, &f2);
```

The user could enter the data in this fashion:

```
1 3.14   -15     2.71e2
```

or like this:

```
1 3.14-15 2.71e2
```

or this:

```
1
3.14
-15
2.71e2
```

or this:

```
1
  3.14-15


    2.71e2
```

Another example:

```
int i1, i2;
float f1, f2;
scanf("%d%d%f%f", &i1, &i2, &f1, &f2);
```

Input:

```
5-30.7-3.14e2
```

The program will print this:

```
5 -30 0.700000 -314.000000
```

Usually, you will pair up calls to **`scanf`** and **`printf`**. Otherwise, the user has no idea what the computer is waiting for.

```
#include <stdio.h> /* printf, scanf */

int main(void)
{
  int age;
  float gpa;

  printf("What is your age? ");
  scanf("%d", &age);

  printf("What is your GPA? ");
  scanf("%f", &gpa);

  printf("Your age is %d and your GPA is %.2f\n", age, gpa);
  return 0;
}
```

Sample session: (The characters in **red** were typed in by the user.)

```
What is your age? 21
What is your GPA? 3.478
Your age is 21 and your GPA is 3.48
```

A full example:

```
#include <stdio.h> /* printf, scanf */

int main(void)
{
  float celsius;    /* To hold the celsius value     */
  float fahrenheit; /* To hold the fahrenheit value. */

    /* Ask user for the Celsius temperature */
  printf("Enter a temperature in Celsius: ");

    /* Read the input into the variable */
  scanf("%f", &celsius);

    /* Calculate fahrenheit from celsius. */
  fahrenheit = (9.0f * celsius / 5.0f) + 32;

    /* Display fahrenheit with 2 decimal places. */
  printf("The temperature in Fahrenheit is %.2f\n", fahrenheit);

    /* Return value to the OS. */
  return 0;
}
```

```
#include <stdio.h> /* printf, scanf */

int main(void)
{
  float celsius;    /* To hold the celsius value     */
  float fahrenheit; /* To hold the fahrenheit value. */

    /* Ask user for the Celsius temperature and save it */
  printf("Enter a temperature in Celsius: ");
  scanf("%f", &celsius);

    /* Calculate fahrenheit from celsius. */
  fahrenheit = (9.0f * celsius / 5.0f) + 32;

    /* Display fahrenheit with 2 decimal places. */
  printf("The temperature in Fahrenheit is %.2f\n", fahrenheit);

    /* Return value to the OS. */
  return 0;
}
```

Sample runs:

```
Enter a temperature in Celsius: 41.5
The temperature in Fahrenheit is 106.70

Enter a temperature in Celsius: 100
The temperature in Fahrenheit is 212.00

Enter a temperature in Celsius: 0
The temperature in Fahrenheit is 32.00
```

Additional points:

- Both **`printf`** and **`scanf`** use *buffered I/O*, which increases performance compared to non-buffered I/O.
- By default, C programs have 3 files (streams) that are available for input and output:
  - **`stdout`** - (Standard Output) This is where output functions like **`printf`** send their output (by default, the screen).
  - **`stdin`** - (Standard Input) This is where input functions like **`scanf`** get their input (by default, the keyboard).
  - **`stderr`** - (Standard Error) This is where errors are usually sent (by default, the screen).
- You can change where these streams read/write. (e.g. you can *redirect* **`stdout`** to go to a file instead of the screen).
- There are several other input/output facilities available:
  - **`putchar`**, **`puts`**, **`fwrite`** are examples of other output functions.
  - **`getchar`**, **`gets`**, **`fread`** are examples of other input functions.
- Flushing **`stdin`**: Calling fflush(stdin) is not standard. Also, alternative ways to discard the input.
