BIL104E: Introduction to Scientific and Engineering Computing

Lecture 3

- ❖Reading from and Writing to Standart I/O
- ❖Manuplating Data With Operators

Readin from and Writint to Standart I/O

The input from the user or print the output to the screen:

- The **`getc()`** function
- The **`putc()`** function
- The **`getchar()`** function
- The **`putchar()`** function

For the time being just memorize the followings:

- **`stdin`**–The standard input for reading (usually keyboard)
- **`stdout`**–The standard output for writing. (usually monitor)
- **`stderr`**–The standard error for writing error messages. (always monitor)

Using the `getc()` Function

The **`getc()`** function reads the next character from a file stream, and returns the character as an integer. The syntax for the **`getc()`** function is

```
# include <stdio.h>
int getc(FILE *stream);
```

EXMAPLE

```
#include <stdio.h>
main() {
int ch;
printf("Please type in one character:\n");
ch = getc( stdin );
printf("The character you just entered is: %c\n", ch);
return 0;
}
```

```
Please type in one character:
H
The character you just entered is: H
```

Using the getchar() Function

The C language provides another function, **getchar()**, to perform a similar operation to **getc().** More precisely, the **getchar()** function is equivalent to **getc(stdin)**. The syntax for the **getchar()** function is

```
#include <stdio.h>
int getchar(void);
```

EXAMPLE:

```
#include <stdio.h>
main() {
int ch1, ch2;
printf("Please type in two characters together:\n");
ch1 = getc( stdin );
ch2 = getchar( );
printf("The first character you just entered is: %c\n", ch1);
printf("The second character you just entered is: %c\n", ch2);
return 0;
}
```

```
Please type in two characters together:
Hi
The first character you just entered is: H
The second character you just entered is: i
```

Printing the Output on the Screen putc()

Like putc(), putchar() can also be used to put a character on the screen. The only difference between the two functions is that putchar() needs only one argument to contain the character. You don't need to specify the file stream, because the standard output (stdout) is the default file stream to putchar(). The syntax for the putchar() function is

```
–  #include <stdio.h>
–  int putchar(int c);
```

EXAMPLE:

```
A
B
C
```

```
#include <stdio.h>
main() { putchar(65); putchar(10);
putchar(66); putchar(10); putchar(67); putchar(10);
  return 0;
}
```

Another Function for Writing `putchar()`

Besides **`getc()`** and **`getchar()`** for reading, the C language also provides two functions, **`putc()`** and **`putchar()`**, for writing. The **`putc()`** function writes a character to the specified file stream, which, in our case, is the standard output pointing to your screen. The syntax for the **`putc()`** function is

- **`#include <stdio.h>`**
- **`int putc(int c, FILE *stream);`**

EXAMPLE:

```
#include <stdio.h>
main() {
int ch;
ch = 65;   /* the numeric value of A */
printf("The character that has numeric value of 65 is:\n");
putc(ch, stdout);
return 0;
}
```

```
The character that has numeric value of 65 is:
A
```

Revisiting the `printf()`Function

- The **`printf()`** function is the first C library function you used in this course to print out messages on the screen. **`printf()`** is a very important function in C, so it's worth it to spend more time on it.
- The syntax for the **`printf()`** function is

  - **`#include <stdio.h>`**
  - **`int printf(const char *format-string, ...);`**

- Here const char *format-string is the first argument that contains the format specifier(s); ... indicates the expression section that contains the expression(s) to be formatted according to the format specifiers. The number of expressions is determined by the number of the format

Revisiting the `printf()`Function

The following are all the format specifiers that can be used in **`printf()`**:

- %c The character format specifier.
- %d The integer format specifier.
- %i The integer format specifier (same as %d).
- %f The floating-point format specifier.
- %e The scientific notation format specifier (note the lowercase e).
- %E The scientific notation format specifier (note the uppercase E).
- %g Uses %f or %e, whichever result is shorter.
- %G Uses %f or %E, whichever result is shorter.
- %o The unsigned octal format specifier.
- %s The string format specifier.
- %u The unsigned integer format specifier.
- %x The unsigned hexadecimal format specifier (note the lowercase x).
- %X The unsigned hexadecimal format specifier (note the uppercase X).
- %p Displays the corresponding argument that is a pointer.
- %n Records the number of characters written so far.
- %% Outputs a percent sign (%).

Revisiting the `printf()`Function

```
#include <stdio.h>
main()
{
  printf("Hex(uppercase) Hex(lowercase) Decimal\n");
  printf("%X             %x             %d\n", 0, 0, 0);
  printf("%X             %x             %d\n", 1, 1, 1);
  printf("%X             %x             %d\n", 2, 2, 2);
  printf("%X             %x             %d\n", 3, 3, 3);
  printf("%X             %x             %d\n", 4, 4, 4);
  return 0
}
```

```
Hex(uppercase) Hex(lowercase) Decimal
0              0              0
1              1              1
2              2              2
3              3              3
4              4              4
```

Revisiting the `printf()`Function

Adding the Minimum Field Width

- The C language allows you to add an integer between the percent sign (%) and the letter in a format specifier. The integer is called the minimum field width specifier because it specifies the minimum field width and ensures that the output reaches the minimum width. For example, in %10f, 10 is a minimum field width specifier that ensures that the output is at least 10 character spaces wide.

- The example below shows how to use the minimum field width specifier.

Revisiting the printf()Function

```
#include <stdio.h>
main(){
int num1, num2; num1 = 12; num2 = 12345;
printf("%d\n", num1);
printf("%d\n", num2);
printf("%5d\n", num1);
printf("%05d\n", num1);
printf("%2d\n", num2);
return 0;
}
```

```
12
12345
   12
00012
12345
```

Revisiting the `printf()`Function

Aligning Output

- As you might have noticed in the previous example, all output is right-justified. In other words, by default, all output is placed on the right edge of the field, as long as the field width is longer than the width of the output.

- You can change this and force output to be left- justified. To do so, you need to prefix the minimum field specifier with the minus sign (-). For example, %-12d specifies the minimum field width as 12, and justifies the output from the left edge of the field.

Aligning Output

```
#include <stdio.h>

main(){
int num1, num2, num3, num4, num5;
num1 = 1;
num2 = 12;
num3 = 123;
num4 = 1234;
num5 = 12345;
printf("%8d  %-8d\n", num1, num1);
printf("%8d  %-8d\n", num2, num2);
printf("%8d  %-8d\n", num3, num3);
printf("%8d  %-8d\n", num4, num4);
printf("%8d  %-8d\n", num5, num5);
return 0;
}
```

```
    1  1
   12  12
  123  123
 1234  1234
12345  12345
```

Assignment Operators

Assignment operators abbreviate assignment expressions

`c = c + 3;`

can be abbreviated as c += 3; using the addition assignment operator

-

Statements of the form

`variable = variable operator expression;`

can be rewritten as

`variable operator= expression;`

- Examples of other assignment operators:

```
d -= 4          (d = d - 4)                e *= 5
(e = e * 5)     f /= 3                     (f = f / 3)
g %= 9          (g = g % 9)
```

Increment and Decrement Operators

- **Increment operator (`++`) :**

– Can be used instead of **`c+=1`**

- **Decrement operator (`--`) :**

– Can be used instead of **`c-=1`**

- **Preincrement:**

– Operator is used before the variable **`(++c or --c)`**

– Variable is changed before the expression it is in is evaluated

- **Postincrement:**

– Operator is used after the variable **`(c++ or c--)`**

– Expression executes before the variable is changed

Increment and Decrement Operators

- If c equals 5, then

```
printf( "%d", ++c );
```

– Prints **6**

```
printf( "%d", c++ );
```

– Prints **5**

– In either case, **c** now has the value of **6**

- When variable not in an expression

– Pre-incrementing and post-incrementing have the same effect

```
++c;
printf( “%d”, c );
```

– Has the same effect as

```
c++;
printf( “%d”, c );
```