MODULE 5
C FORMATTED INPUT/OUTPUT

MODULE 18
C++ FORMATTED I/O

My Training Period: hours

Abilities

- Able to understand and use the C formatted input/output functions library.
- Able to understand and use predefined/built-in functions and their respective header files.
- Appreciates other `printf()` and `scanf()` family.

5.1 Introduction

- This Module deals with the formatting features of `printf()`, `scanf()`, `cin` and `cout`, the most frequently use functions.
- Here, you will learn how to use the predefined functions provided by the header files (standard or non standard).
- `printf()` function input data from the standard input stream.
- `scanf()`, output data to the standard output stream.
- For other functions that use the standard input and standard output are `gets()`, `puts()`, `getchar()` and `putchar()`. Keep in mind that some of the functions discussed here are non-standard.
- We have to include header file `stdio.h` (C) or `iostream.h`/`iostream` (C++) in program to call these functions.

5.2 Streams

- All input and output is performed with stream.
- Stream is a sequences of characters organized into lines.
- Each line consists of zero or more characters, and end with the newline character.
- When program execution begins, these streams are connected to the program automatically.
- Normally:

    1. The standard input stream is connected to the **keyboard**.
    2. The standard output stream is connected to the **screen**
    3. Redirection of no. 1 and 2 to other devices, a third stream, **standard error**, connected to the screen. Error messages are output to the standard error stream.

5.3 Formatting output with `printf()`

- For precise output formatting, every `printf()` call contains a format control string that describes the output format.
- The format control string consists of:

    1. Conversion specifiers.
    2. Flags.
    3. Field widths.
    4. Precisions.
    5. Literal characters.

- Together with percent sign (%), these, form conversion specifications. Function `printf()` can perform the following formatting capabilities:

    1. **Rounding** floating-point values to an indicated number of decimal places.
    2. **Aligning** a column of numbers with decimal points appearing one above the other.
    3. Right-justification and left-justification of outputs.
    4. **Inserting literal characters** at precise locations in a line of output.
    5. Representing floating-point number in **exponential format**.
    6. Representing unsigned integers in **octal** and **hexadecimal format**.
    7. Displaying all types of data with fixed-size field widths and precisions.

- As discussed in Module 4, `printf()` is a variadic function. It can accept variable number of arguments. The `printf()` general form is:

  `printf` (*format-control-string*, *other-arguments*);

- For example:

```
printf("Using Source IP: %s port: %u\n", argv[1], atoi(argv[2]));
printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
```

- The *format-control-string* describes the output format, and *other-arguments* (optional) correspond to each conversion specification in the *format-control-string*.
- Each conversion specification begins with a percent sign (%) and ends with a conversion specifier.

5.4 Printing Integers

- Integer is a whole number, such as 880 or –456, that contains no decimal point.
- Table 5.1 is a summary of an integer conversion specifier.

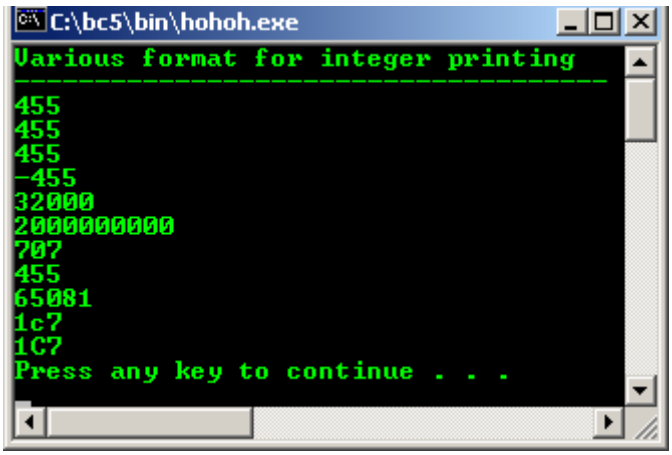
Conversion specifier	Description
`d`	Display a signed decimal integer
`i`	Display a signed decimal integer. (Note: The `i` and `d` specifiers are different when used with `scanf()`.)
`o`	Display an unsigned octal integer.
`u`	Display an unsigned decimal integer.
`x` or `X`	Display an unsigned decimal integer. `x` causes the digit `0-9` and the letters `A-F` to be displayed, and `x` causes the digits `0-9` and `a–f` to be displayed.
`h` or `l` (letter `l`)	Place **before** any integer conversion specifier to indicate that a `short` or `long` integer is displayed respectively.

Table 5.1: Integer conversion specifiers

- Let explore through a program example.

```
//Using the integer conversion specifiers
#include <stdio.h>
#include <stdlib.h>

int main()
{
      printf("Various format for integer printing\n");
      printf("-------------------------------------\n");
      printf("%d\n",  455);
      printf("%i\n",  455);  //i same as d in printf()
      printf("%d\n",  +455);
      printf("%d\n",  -455);
      printf("%hd\n",  32000);
      printf("%ld\n",  2000000000L);
      printf("%o\n",  455);
      printf("%u\n",  455);
      printf("%u\n",  -455);
      //-455 is read by %u and converted to the unsigned
      //value 4294966841 by 4 bytes integer
      printf("%x\n",  455);
      printf("%X\n",  455);
      system("pause");
     return 0;
}
```

Output:



5.5 Printing Floating-point Number

- Contains the decimal point such as 35.5 or 7456.945.
- Table 5.2 summarizes the floating-point conversion specifiers.

Conversion specifier	Description
`e` or `E`	Display a floating-point value in exponential notation.
`f`	Display floating-point values.
`g` or `G`	Display a floating-point value in either the floating-point form `f` or the exponential form `e` (or `E`)
`l`	Place before any floating-point conversion specifier to indicate that a `long double` floating-point value is displayed.

Table 5.2: Floating-point conversion specifiers

- Exponential notation is the computer equivalent of scientific notation used in mathematics. For example, 150.2352 is represented in scientific notation as:

  1.502352×10^2

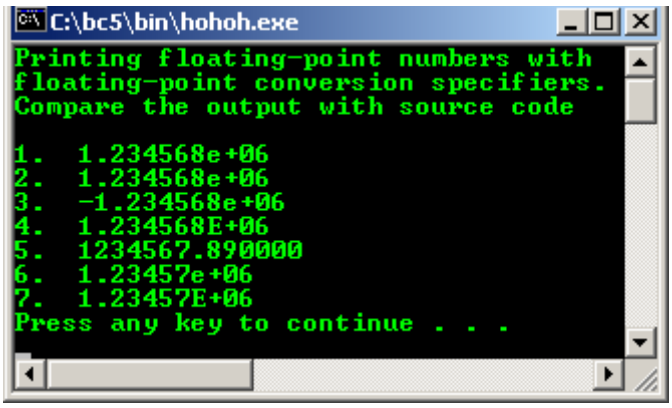
- And is represented in exponential notation as

  `1.502352E+02` by computer

- So, $150.2352 = 1.502352 \times 10^2 = 1.502352E+02$
- **E** stand for exponent.
- **e**, **E** and **f** will output with 6 digits of precision to the right of the decimal point by default.
- Let try a program example.

```
//Printing floating-point numbers with
//floating-point conversion specifiers

#include <stdio.h>
#include <stdlib.h>

void main()
{
      printf("Printing floating-point numbers with\n");
      printf("floating-point conversion specifiers.\n");
      printf("Compare the output with source code\n\n");
      printf("1.  %e\n", 1234567.89);
      printf("2.  %e\n", +1234567.89);
      printf("3.  %e\n", -1234567.89);
      printf("4.  %E\n", 1234567.89);
      printf("5.  %f\n", 1234567.89);
      printf("6.  %g\n", 1234567.89);
      printf("7.  %G\n", 1234567.89);
      system("pause");
}
```

Output:

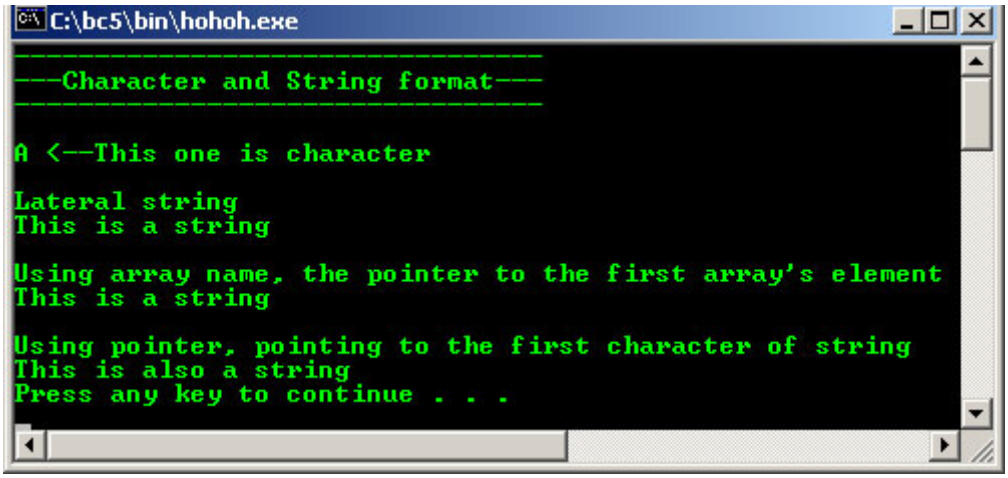


5.6 Printing Strings And Characters

- **c** and **s** conversion specifiers are used to print individual characters and strings respectively.
- Conversion specifier `c` requires a char argument and `s` requires a pointer to char as an argument.
- `s` causes characters to be printed until a terminating `NULL` (‘`\0`’) character is encountered.
- A program example.

```
//Printing strings and characters
#include <stdio.h>
#include <stdlib.h>

int main()
{
char character = 'A';
char string[]  =  "This is a string";
char *stringPtr = "This is also a string";
printf("---------------------------------\n");
printf("---Character and String format---\n");
printf("---------------------------------\n\n");
printf("%c <--This one is character\n", character);
printf("\nLateral string\n");
printf("%s\n", "This is a string");
printf("\nUsing array name, the pointer to the first array's element\n");
printf("%s\n", string);
printf("\nUsing pointer, pointing to the first character of string\n");
printf("%s\n", stringPtr);
system("pause");
return 0;
}
```

Output:



1.1 Other Conversion Specifiers

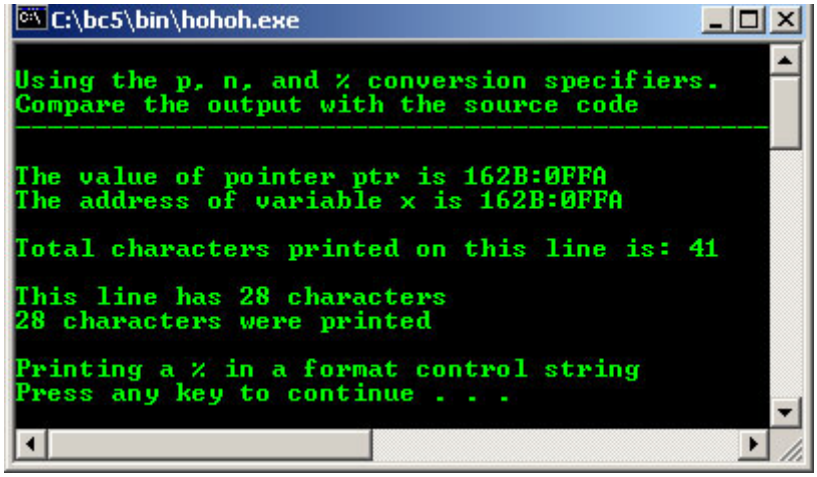
- p, n and % are summarized in table 5.3 followed by the program example.

Conversion specifier	Description
p	Display a pointer value (memory address) in an implementation defined manner.
n	Store the number of characters already output in the current `printf()` statement. A pointer to an integer is supplied as the corresponding argument. Nothing is displayed.
%	Display the percent character.

Table 5.3: Other conversion specifiers

```
//Using the p, n, and % conversion specifiers
#include <stdio.h>
#include <stdlib.h>

int main()
{
       int  *ptr;
       //pointer variable
       int  x  =  12345, y;

       ptr = &x;

       //assigning address of variable x to variable ptr
       printf("\nUsing the p, n, and %% conversion specifiers.\n");
       printf("Compare the output with the source code\n");
       printf("-----------------------------------------------\n\n");
       printf("The value of pointer ptr is %p\n", ptr);
       printf("The address of variable x is %p\n\n", &x);
       printf("Total characters printed on this line is:%n", &y);
       printf(" %d\n\n", y);

       y = printf("This line has 28 characters\n");

       printf("%d characters were printed\n\n", y);
       printf("Printing a %% in a format control string\n");
       system("pause");
   return 0;
}
```

Output:



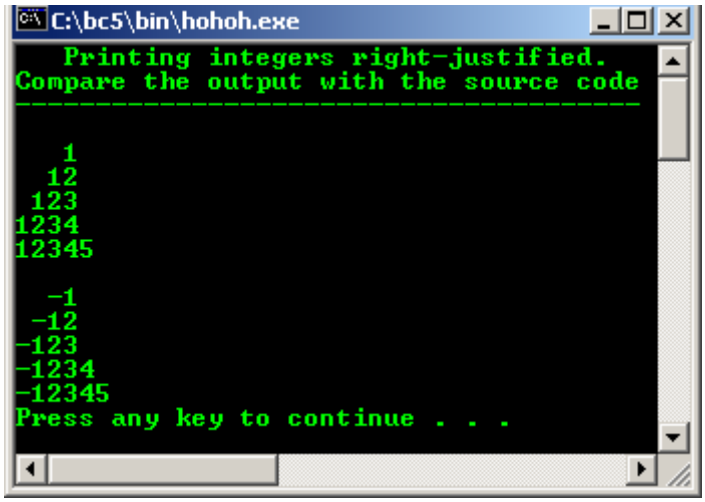
1.2 Printing With Field Widths And Precisions

- A field width determines the exact size of a field in which data is printed.
- If the field width is larger then the data being printed, the data will normally be right-justified within that field.
- An integer representing the field width is inserted between the percent sign (%) and the conversion specifier in the conversion specification.
- Function `printf()` also provides the ability to specify the precision with which data is printed.
- Precision has different meanings for different data types.
- A program example.

```
  //Printing integers right-justified
```

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
     printf("   Printing integers right-justified.\n");
     printf("Compare the output with the source code\n");
     printf("---------------------------------------\n\n");
     printf("%4d\n",  1);
     printf("%4d\n",  12);
     printf("%4d\n",  123);
     printf("%4d\n",  1234);
     printf("%4d\n\n",  12345);
     printf("%4d\n",   -1);
     printf("%4d\n",   -12);
     printf("%4d\n",   -123);
     printf("%4d\n",   -1234);
     printf("%4d\n",   -12345);
     system("pause");
     return 0;
}
```

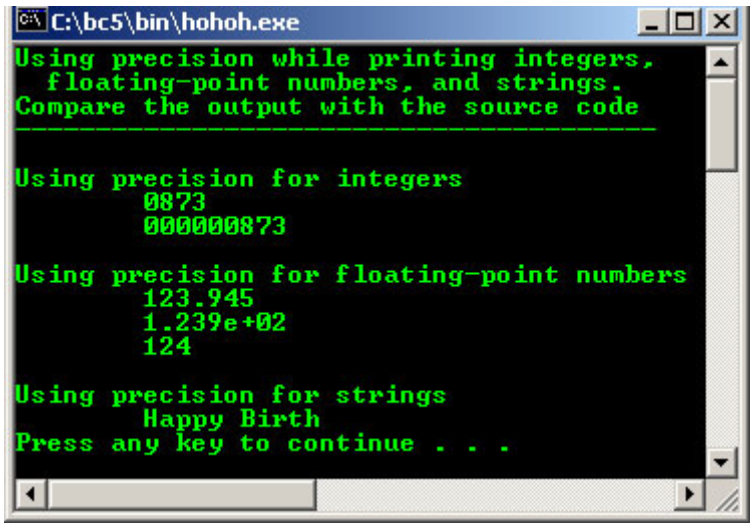
Output:



- Another example:

```
//Using precision while printing integers,
//floating-point numbers and strings
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int     i = 873;
    float   f =  123.94536;
    char s[] = "Happy Birthday";

    printf("Using precision while printing integers,\n");
    printf("  floating-point numbers, and strings.\n");
    printf("Compare the output with the source code\n");
    printf("---------------------------------------\n\n");
    printf("Using precision for integers\n");
    printf("\t%.4d\n\t%.9d\n\n", i, i);
    printf("Using precision for floating-point numbers\n");
    printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
    printf("Using precision for strings\n");
    printf("\t%.11s\n", s);
    system("pause");
    return 0;

}
```

Output:



- By using asterisk (*), it is also can be like this:

```
printf("%*.*f", 7, 2, 98.736)
```

- This statement uses 7 for the field width, 2 for the precision and will output the value 98.74 right-justified.

1.3 Using Flags In The `printf()` Format Control String

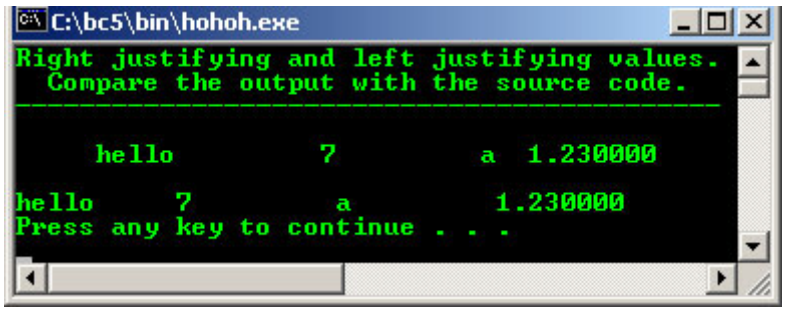
- Flags used to supplement its output formatting capabilities.
- Five flags are available to be used in format control string as shown in table 5.4 then followed by program examples.

Flag	Description
`-` (minus sign)	Left-justify the output within the specified field
`+` (plus sign)	Display a plus sign preceding positive values and a minus sign preceding negative values.
space	Print a space before a positive value not printed with the `+` flag.
#	Prefix `0` to the output value when used with the octal conversion specifier `o`. Prefix `0x` or `0X` to the output value when used with the hexadecimal conversion specifiers `x` or `X`. Force a decimal points for a floating-point number printed with `e`, `E`, `f`, `g`, or `G` that does not contain a fractional part. (Normally the decimal point is only printed if a digit follows it). For `g` and `G` specifiers, trailing zeros are not eliminated.
`0` (zero)	Pad a field with leading zeros.

Table 5.4: Format control string flags

```
//Right justifying and left justifying values
#include <stdio.h>
#include <stdlib.h>

int main()
{
        printf("Right justifying and left justifying values.\n");
        printf("  Compare the output with the source code.\n");
        printf("-----------------------------------------\n\n");
        printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
        printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
        system("pause");
        return 0;
}
```

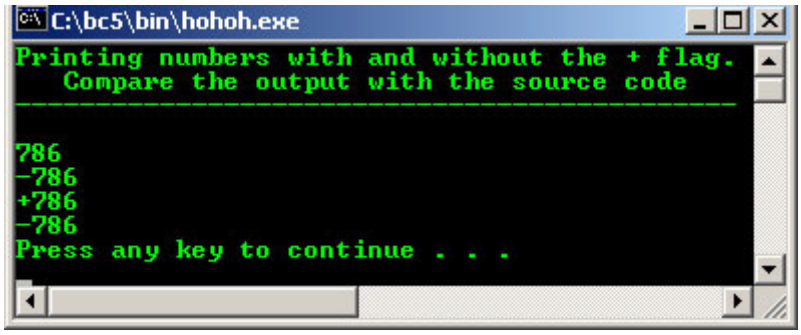
Output:



- Program example:

```
//Printing numbers with and without the + flag
#include <stdio.h>
#include <stdlib.h>

int main()
{
     printf("Printing numbers with and without the + flag.\n");
     printf("   Compare the output with the source code\n");
     printf("-----------------------------------------\n\n");
     printf("%d\n%d\n", 786, -786);
     printf("%+d\n%+d\n", 786, -786);
     system("pause");
     return 0;
}
```

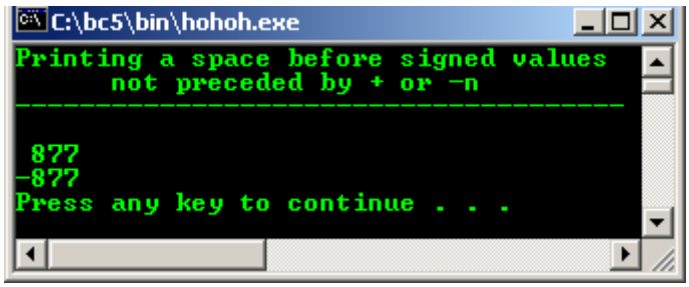
Output:



- Another example:

```
//Printing a space before signed values
//not preceded by + or -
#include <stdio.h>
#include <stdlib.h>

int main()
{
     printf("Printing a space before signed values\n");
     printf("      not preceded by + or -n\n");
     printf("--------------------------------------\n\n");
     printf("% d\n% d\n", 877,  -877);
     system("pause");
     return 0;
}
```

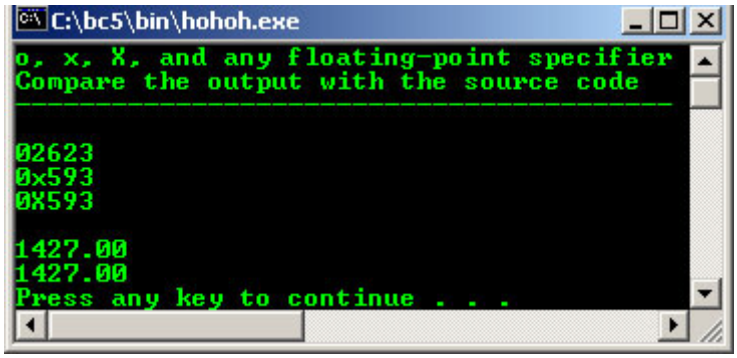
Output:



- Program example:

```
//o, x, X, and any floating-point specifier
#include <stdio.h >
#include <stdlib.h>

int main()
{
     int    c  = 1427;
     float  p  =  1427.0;

     printf("o, x, X, and any floating-point specifiers\n");
     printf("Compare the output with the source code\n");
     printf("----------------------------------------\n\n");
     printf("%#o\n", c);
     printf("%#x\n", c);
     printf("%#X\n", c);
     printf("\n%#g\n", p);
     printf("%#G\n", p);
     system("pause");
     return 0;
}
```

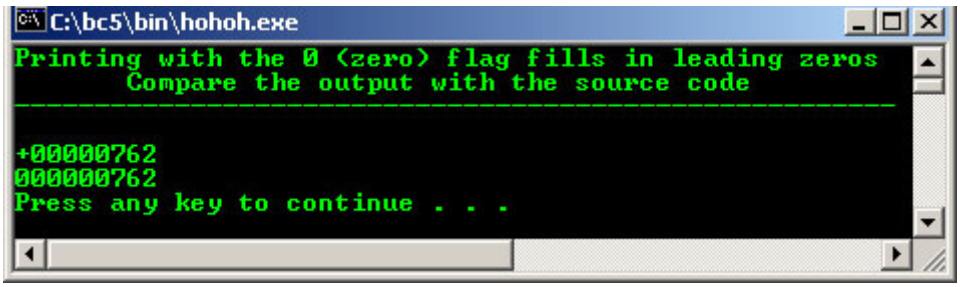
Output:



- Program example.

```
//Printing with the 0 (zero) flag fills in leading zeros
#include <stdio.h>
#include <stdlib.h>

int main()
{
       printf("Printing with the 0 (zero) flag fills in leading zeros\n");
       printf("       Compare the output with the source code\n");
       printf("-------------------------------------------------------\n\n");
       printf("%+09d\n", 762);
       printf("%09d", 762);
       printf("\n");
       system("pause");
       return 0;
}
```

Output:



1.4 Printing Literals And Escape Sequences

- Various control characters, such as newline and tab, must be represented by escape sequences.
- An escape sequence is represented by a backslash (\) followed by a particular escape character.
- Table 5.5 summarizes all the escape sequences and the actions they cause.

Escape sequence	Description
`\'`	Output the single quote (') character.
`\"`	Output the double quote (") character.
`\?`	Output the question mark (?) character.
`\\`	Output the backslash (\) character.
`\a`	Cause an audible (bell) or visual alert.
`\b`	Move the cursor back one position on the current line.
`\f`	Move the cursor to the start of the next logical page.
`\n`	Move the cursor to the beginning of the next line.
`\r`	Move the cursor to the beginning of the current line.
`\t`	Move the cursor to the next horizontal tab position.
`\v`	Move the cursor to the next vertical tab position.

Table 5.5: Escape sequences

- Other `printf()` family that you might find somewhere, sometime is listed in the following Table. It is important to note that some of these are not part of the standard library but are widely available. You have to check your compiler.

Function	Description & Prototype
`printf()`	Prints output on the standard output stream *stdout*. Include `stdio.h`.
	int printf(const char *restrict *format*, ...);
`fprintf()`	Prints output on the named output *stream*. Include `stdio.h`.
	int fprintf(FILE *restrict *stream*, const char *restrict *format*, ...);
`sprintf()`	Prints output followed by the null byte, `'\0'`, in consecutive bytes starting at `*s`; it is the user's responsibility to ensure that enough space is available. Include `stdio.h`.
	int sprintf(char *restrict *s*, const char *restrict *format*, ...);
`snprintf()`	Prints into a string with length checking. Equivalent to `sprintf()`, with the addition of the `n` argument which states the size of the buffer referred to by `s`. If `n` is zero, nothing shall be written and *s* may be a null pointer. Otherwise, output bytes beyond the `n-1`st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array. Include `stdio.h`.
	int snprintf(char *restrict *s*, size_t *n*, const char *restrict *format*, ...);
`vfprintf()`	Print to a `FILE` stream from a `va_arg` structure. Equivalent to `printf()`, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `stdarg.h`. Include `stdarg.h and stdio.h`.
	int vfprintf(FILE *restrict *stream*, const char *restrict *format*, va_list *ap*);
`vprintf()`	Prints to 'stdout' from a `va_arg` structure. Equivalent to `fprintf()` respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `stdarg.h`. Include `stdarg.h and stdio.h`.

	int vprintf(const char *restrict *format*, va_list *ap*);
`vsprintf()`	Prints to a string from a `va_arg` structure. Equivalent to `sprintf()` respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `stdarg.h`. Include `stdarg.h and stdio.h`.
	int vsprintf(char *restrict *s*, const char *restrict *format*, va_list *ap*);
`vsnprintf()`	Prints to a string with length checking from a `va_arg` structure. Equivalent to `snprintf()`, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `stdarg.h`. Include `stdarg.h and stdio.h`.
	int vsnprintf(char *restrict *s*, size_t *n*, const char *restrict *format*, va_list *ap*);

Table 5.6: `printf()` family

5.11 Formatting Input With `scanf()`

- Used for precise input formatting.
- Every `scanf()` function contains a format control string that describes the format of the data to be input.
- The format control string consists of conversion specifications and literal characters.
- Function `scanf()` has the following input formatting capabilities

    1. Inputting all types of data.
    2. Inputting specific characters from an input stream.
    3. Skipping specific characters in the input stream.

- It is written in the following form:

```
scanf (format-control-string, other-arguments);
```

- For example:

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

- The *format-control-string* describes the formats of the input, and the *other-arguments* are **pointers to variables** in which the input will be stored.
- Table 5.7 summarizes the conversion specifiers used to input all types of data.

Conversion specifier	Description
Integers	
`d`	Read an optionally signed decimal integer. The corresponding argument is a pointer to integer.
`i`	Read an optionally signed decimal, octal, or hexadecimal integer. The corresponding argument is a pointer to integer.
`o`	Read an octal integer. The corresponding argument is a pointer to unsigned integer.
`u`	Read an unsigned decimal integer. The corresponding argument is a pointer to unsigned integer.
`x` or `X`	Read a hexadecimal integer. The corresponding argument is a pointer to unsigned integer.
`h` or `l`	Place before any of the integer conversion specifiers to indicate that a `short` or `long` integer is to be input.
Floating-point numbers	
`e, E, f, g` or `G`	Read a floating-point value. The corresponding argument is a pointer to a floating-point variable
`l` or `L`	Place before any of the floating-point conversion specifiers to indicate that a `double` or `long double` value is to be input.
Characters and strings	
`c`	Read a character. The corresponding argument is a pointer to `char`, no null (`'\0'`) is added.
`s`	Read a string. The corresponding argument is a pointer to an array of type `char` that is large enough to hold the string and a terminating null (`'\0'`) character.

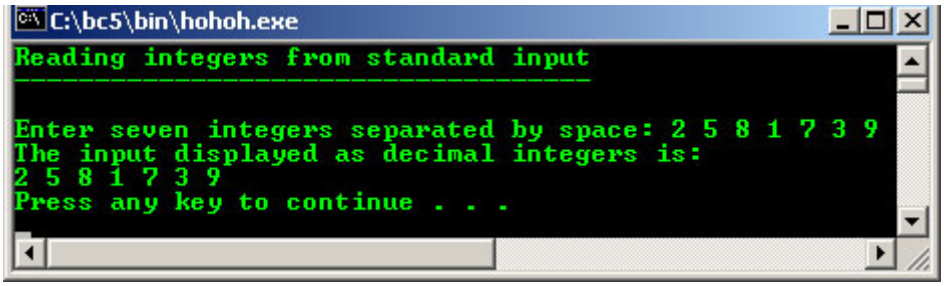
Scan set [*scan characters*]	Scan a string for a set of characters that are stored in an array.
Miscellaneous	
p	Read a pointer address of the same form produced when an address is output with %p in a `printf()` statement.
n	Store the number of characters input so far in this `scanf()`. The corresponding argument is a pointer to integer.
%	Skip a percent sign (%) in the input.

Table 5.7: Conversion specifiers for `scanf()`

- Let explore through program examples:

```
//Reading integers
#include <stdio.h>
#include <stdlib.h>

int main()
{
       int  a, b, c, d, e, f, g;

       printf("Reading integers from standard input\n");
       printf("-----------------------------------\n\n");
       printf("Enter seven integers separated by space: ");
       scanf("%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g);
       printf("The input displayed as decimal integers is: \n");
       printf("%d %d %d %d %d %d %d\n",  a, b, c, d, e, f, g);
       system("pause");
       return 0;
}
```

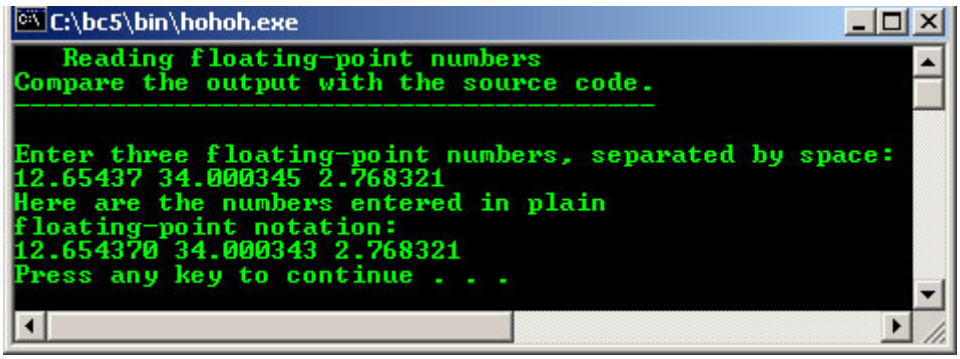
Output:



- Program example:

```
//Reading floating-point numbers
#include <stdio.h>
#include <stdlib.h>

int main()
{
  float   a,  b,  c;

  printf("   Reading floating-point numbers\n");
  printf("Compare the output with the source code.\n");
  printf("----------------------------------------\n\n");
  printf("Enter three floating-point numbers, separated by space: \n");
  scanf("%e%f%g", &a, &b, &c);
  printf("Here are the numbers entered in plain\n");
  printf("floating-point notation:\n");
  printf("%f %f %f\n", a, b, c);
  system("pause");
  return 0;
 }
```

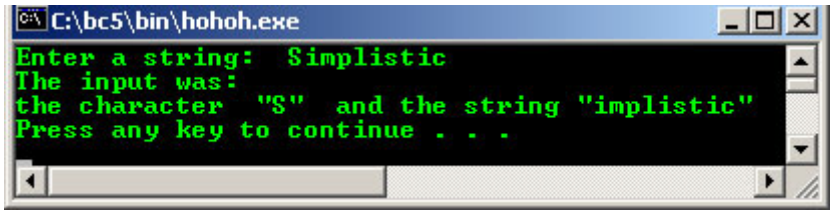
Output:



- Program example:

```
 //Reading characters and strings
#include <stdio.h>
#include <stdlib.h>

int main()
{
     char  x,  y[20];

     printf("Enter a string:  ");
     scanf("%c%s", &x, y);
     printf("The input was: \n");
     printf("the character  \"%c\"  ", x);
     printf("and the string \"%s\"\n", y);
     system("pause");
     return 0;
}
```

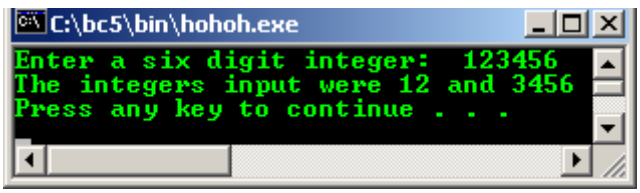
Output:



- Program example:

```
//input data with a field width
#include <stdio.h>
#include <stdlib.h>

int main()
{
       int    x,  y;

       printf("Enter a six digit integer:  ");
       scanf("%2d%d", &x, &y);
       printf("The integers input were %d and %d\n", x, y);
       system("pause");
       return 0;
}
```

Output:

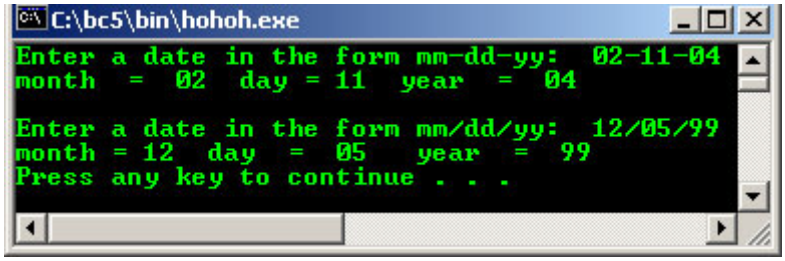


- Program example:

```
//Reading and discarding characters from the input stream
```

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
     int  month1, day1, year1, month2, day2, year2;

     printf("Enter a date in the form mm-dd-yy:  ");
     //pad 0 for two fields and discarding the - characters....
     scanf("%d%*c%d%*c%d",  &month1, &day1, &year1);
     printf("month  =  %02d  day = %02d  year  =  %02d\n\n", month1, day1,
year1);
     printf("Enter a date in the form mm/dd/yy:  ");
     //pad 0 for two fields and discarding the / characters...
     scanf("%d%*c%d%*c%d", &month2, &day2, &year2);
     printf("month = %02d  day  =  %02d   year  =  %02d\n", month2, day2,
year2);
     system("pause");
     return 0;
}
```

Output:



- Other `scanf()` family that you might find is listed in the following Table. It is important to note that some of these are not part of the standard library but are widely available. You have to check your compiler.

Function	Description & Prototype
`scanf()`	Read from the standard input stream *stdin*. Include `the stdio.h`.
	int scanf(const char *restrict *format*, ...);
`fscanf()`	Read from the named input *stream*. Include the `stdio.h`.
	int fscanf(FILE *restrict *stream*, const char *restrict *format*, ...);
`sscanf()`	Read from the string `s`. Include the `stdio.h`.
	int sscanf(const char *restrict *s*, const char *restrict *format*, ...);
`vscanf()`	Equivalent to the `scanf()` function except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `stdarg.h`. Include the `stdarg.h` and `stdio.h`.
	int vscanf(const char *restrict *format*, va_list *arg*);
`vsscanf()`	Equivalent to the `sscanf()` functions except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `stdarg.h`. Include the `stdarg.h` and `stdio.h`.
	int vsscanf(const char *restrict *s*, const char *restrict *format*, va_list *arg*);
`vfscanf()`	Equivalent to the `fscanf()` function except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `stdarg.h`. Include the `stdarg.h` and `stdio.h`.
	int vfscanf(FILE *restrict *stream*, const char *restrict *format*, va_list *arg*);

Table 5.8: `scanf()` family

- Be careful when using the `printf()` and `scanf()` families because improper use can generate buffer overflow problems. The buffer overflows phenomenon widely exploited by malicious, worm and virus codes. See everyday security updates and the Proof Of Concept (POC) at frsirt regarding the buffer overflow.

5.12 An Introduction To C++ Input/Output Streams

- C++ provides an alternative to `printf()` and `scanf()` function calls for handling input/output of the standard data types and strings. For example:

```
printf("Enter new tag:  ");
scanf("%d", &tag);
printf("The new tag is: %d\n", tag);
```

- Is written in C++ as:

```
cout<<"Enter new tag: ";
cin>>tag;
cout<<"The new tag is: "<<tag<<'\n';
```

- The first statement uses the standard output stream `cout` and the operator << (the stream insertion operator, synonym to "put to"). The statement is read:

  The string "*`Enter new tag`*" is put to the output stream `cout`.

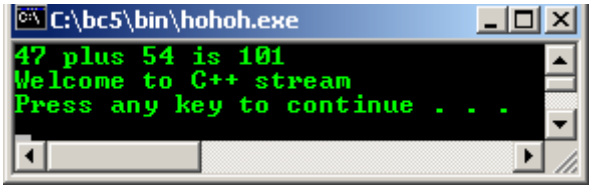
- The second statement uses the standard input stream `cin` and the operator >> (the stream extraction operator, synonym to "get from"). This statement is read something like:

  Get a value for tag from the input stream `cin`.

- The stream insertion and extraction operators, unlike `printf()` and `scanf()`, do not require format strings and conversion specifiers to indicate the data types being output or input.
- The operator & also not required as in `scanf()`. Mostly done automatically and you have to know how to use left shift, << and right shift, >> operators together with spaces and new line to do the formatting. As a conclusion, `cout` and `cin` are simpler and easier to use.
- You have to include the iostream.h header file to use these stream input/outputs.
- `iostream` library provide hundreds of I/O capabilities. The `iostream.h` contains `cin`, `cout`, `cerr` (unbuffered standard error device) and `clog` (buffered standard error device) objects for standard input stream, standard output stream and standard error stream respectively.
- Let try a simple program example.

```
//concatenating the << operator
#include <iostream.h>
#include <stdlib.h>

int main()
{
       cout<<"47 plus 54 is "<<(47 + 54)<<endl;
       cout<<"Welcome to C++ stream\n";
       system("pause");
       return 0;
}
```

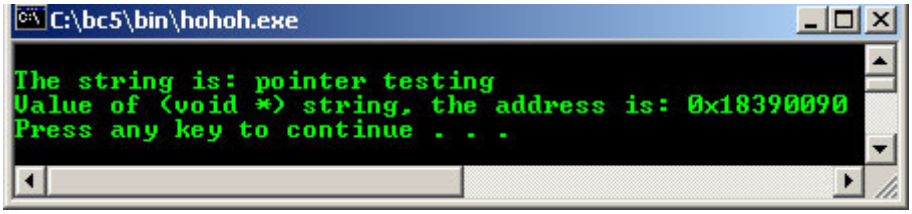
Output:




```
//printing the address stored in a char * variable
#include <iostream.h>
#include <stdlib.h>

int main()
{
       char * string = "pointer testing";

       cout<<"\nThe string is: "<<string
              <<"\nValue of (void *) string, the address is: "
              <<(void *)string <<endl;
       system("pause");
       return 0;
```

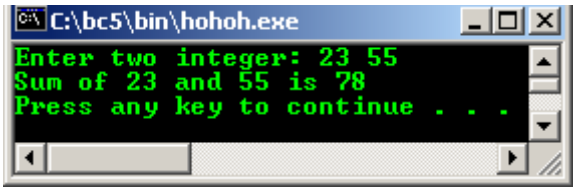
```
}
```

Output:



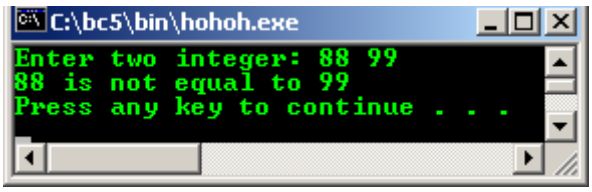

```
//stream extraction operator
//input from keyboard with cin
#include <iostream.h>
#include <stdlib.h>

int main()
{
	int x, y;

	cout<<"Enter two integers: ";
	cin>>x>>y;
	cout<<"Sum of "<<x<<" and "<<y<<" is "<<(x + y)<<endl;
	system("pause");
	return 0;
}
```

Output:




```
//stream extraction operator
//variation of cin
#include <iostream.h>
#include <stdlib.h>

int main()
{
	int x, y;

	cout<<"Enter two integers: ";
	cin>>x>>y;
	cout<<x<<(x == y?" is " : " is not ")<<"equal to "<<y<<endl;
	system("pause");
	return 0;
}
```

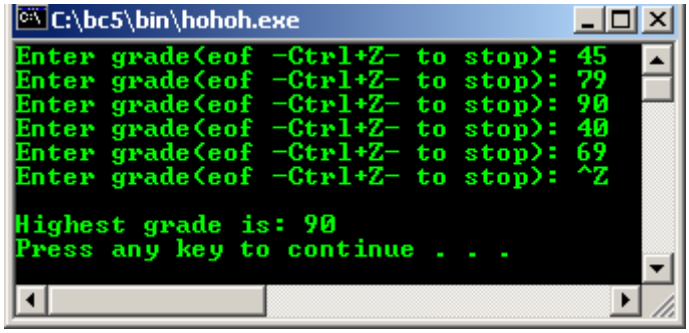
Output:




```
//stream extraction operator
//variation of cin
#include <iostream.h>
#include <stdlib.h>

int main()
{
	int mark, HighMark = -1;

	cout<<"Enter grade(eof -Ctrl+Z- to stop): ";
```

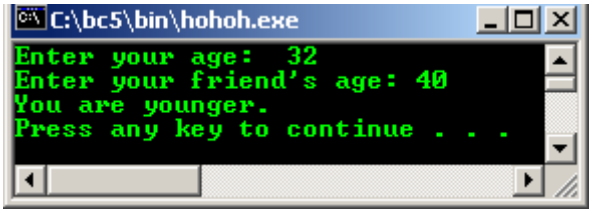
```
        while(cin>>mark)
        {
                if(mark>HighMark)
                HighMark = mark;
                cout<<"Enter grade(eof -Ctrl+Z- to stop): ";
        }
        cout<<"\nHighest grade is: "<<HighMark<<endl;
        system("pause");
        return 0;
}
```

Output:




```
//Simple stream input/output
#include <iostream.h>
#include <stdlib.h>

int main()
{
        cout<<"Enter your age:  ";

        int   myAge;
        cin>>myAge;
        cout<<"Enter your friend's age: ";

        int  friendAge;
        cin>>friendAge;

        if(myAge > friendAge)
                cout<<"You are older.\n";
        else
        if(myAge < friendAge)
                cout<<"You are younger.\n";
        else
        cout<<"You and your friend are the same age.\n";
        system("pause");
        return 0;
}
```

Output:

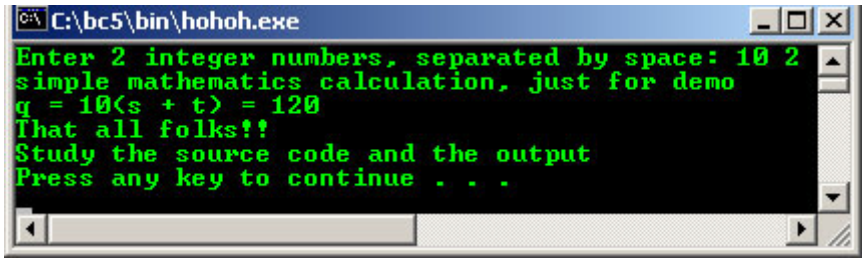


- Usage of the `'\n'`, `endl` with `cout` example.

```
//Some of the cout usage
#include <stdlib.h>
#include <iostream.h>

void main(void)
{
        int q, s = 0, t = 0;

        q = 10*(s + t);
```

```
        cout<<"Enter 2 integer numbers,"
              " separated by space: ";
        //using the " for breaking literal strings
        cin>>s>>t;
           q = 10*(s + t);
        cout<<"simple mathematics calculation, just for demo"<<'\n';
        //using '\n' for newline
        cout<<"q = 10(s + t) = "<<q<<endl;
        //using endl for new line
        cout<<"That all folks!!"<<"\n";
        cout<<"Study the source code and the output\n";
        system("pause");
}
```

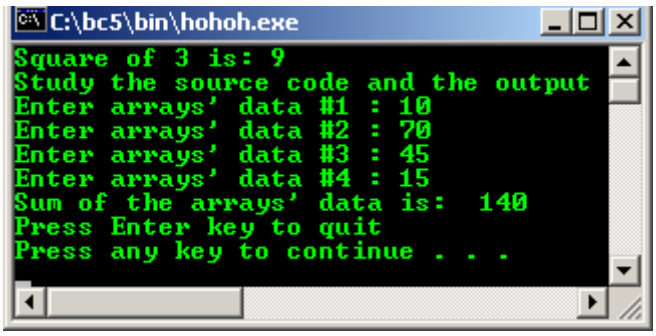
Output:



- `cout` and `cin` example for function call and array.

```
//cout and cin example for function call and array
#include <stdlib.h>
#include <iostream.h>

float simple_calc(float);

void main(void)
{
       float x = 3, y[4], sum=0;
       int i;

       cout<<"Square of 3 is: "<<simple_calc(x)<<'\n';
       //cout with function call

       cout<<"Study the source code and the output"<<endl;
              for (i=1; i<5; i++)
              {
                     cout<<"Enter arrays' data #"<<i<<" : ";
                     cin>>y[i];
                     sum = sum + y[i];
                     //array
              }
       cout<<"Sum of the arrays' data is:  "<<sum<<endl;
       cout<<"Press Enter key to quit\n";
       system("pause");
}

float simple_calc(float x)
{
       float   p;
       p = (x * x);
       return p;
}
```

Output:



- Other I/O header files include:

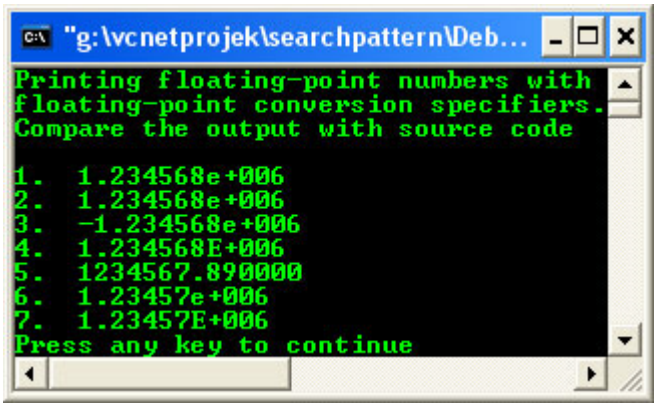
Header file	Description
`iomanip.h`	Used for performing formatted IO with so-called **parameterized stream manipulators**.
`strstream.h`	Used for performing in-memory formatting. This resembles file processing IO operation, normally performed to and from character arrays rather than files.
`stdiostream.h`	Used for programs those mix the C and C++ style of IO. This can be used for modifying existing C program or migrating C to C++.

Table 5.9: Other header files used for C++ formatted I/O

- The following program example compiled using VC++/VC++ .Net.

```
//Printing floating-point numbers with
//floating-point conversion specifiers
#include <cstdio>

void main()
{
      printf("Printing floating-point numbers with\n");
      printf("floating-point conversion specifiers.\n");
      printf("Compare the output with source code\n\n");
      printf("1.  %e\n", 1234567.89);
      printf("2.  %e\n", +1234567.89);
      printf("3.  %e\n", -1234567.89);
      printf("4.  %E\n", 1234567.89);
      printf("5.  %f\n", 1234567.89);
      printf("6.  %g\n", 1234567.89);
      printf("7.  %G\n", 1234567.89);
}
```

Output:



- Then, program example compiled using `gcc` on FeDora 3.

[bodo@bakawali ~]$ cat module5.c

```
/*Using the p, n, and % conversion specifiers*/
/*****************module5.c******************/
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
        int  *ptr;
        /*pointer variable*/
        int  x  =  12345, y;

        ptr = &x;

        /*assigning address of variable x to variable ptr*/
        printf("\nUsing the p, n, and %% conversion specifiers.\n");
        printf("Compare the output with the source code\n");
        printf("-----------------------------------------------\n\n");
        printf("The value of pointer ptr is %p\n", ptr);
        printf("The address of variable x is %p\n\n", &x);
        printf("Total characters printed on this line is:%n", &y);
        printf(" %d\n\n", y);

        y = printf("This line has 28 characters\n");

        printf("%d characters were printed\n\n", y);
        printf("Printing a %% in a format control string\n");
        return 0;
}
```

[bodo@bakawali ~]$ gcc module5.c -o module5
[bodo@bakawali ~]$ ./module5

```
Using the p, n, and % conversion specifiers.
Compare the output with the source code
-----------------------------------------------

The value of pointer ptr is 0xbff73840
The address of variable x is 0xbff73840

Total characters printed on this line is: 41

This line has 28 characters
28 characters were printed

Printing a % in a format control string
```
[bodo@bakawali ~]$

- Quite a complete discussion for other C++ formatted I/O will be discussed in Module 18.

----------o0o----------

Further reading and digging:

1. Check the best selling C/C++ books at Amazon.com.
2. The formatted I/O for wide character/Unicode is discussed HERE and the example of the implementation (Microsoft C) is presented HERE.