CS110: Computer Programming Lab
Department of CSE
IIT, Guwahati Jan-May 2018

1 Declarations and Input-Output of Numerical Values

1.1 Introduction: Module 02 Stage 01

Students should attempt this stage after they have successfully trained themselves using drill *Module 01 Stage 02*. Please ensure that your successful training (as determined by a drill assessment exercise) for the previous stage has been recorded on the course records.

Students who do not have previous programming experiences may find this practice drill to be introducing a lot of new concepts. The drill, however, is foundational in many ways. Those with previous C programming experiences would complete this practice drill quickly.

To remind you of the drill procedures and arrangements, please read these dot-points:

- Use the practice session described in this document to learn the topics. Then ask for your assessment problem. You may also check your readiness for a formal assessment by trying one of the assessment exercises yourself.
- Only assessment on one stage is recorded in any one lab session.
- After the assessment, please stay in the lab and either do more assessment exercises from this stage for further learning or begin preparing for the next stage.
- Successful result for the assessment exercise requires pass for every criteria (NO EXCEPTION ALLOWED) listed in the checklist. All passes for the checklist items must be observed by a course tutor in a single assessment demonstration from the student.
- Even if you have previous education in the computer programming, please do not use the features of the programming language C other than those suggested for this stage.
- Students who neglect to prepare for the lab session before arriving in the lab will find it difficult to complete the drill assessment in a single lab session.

1.2 Remember: Learning is the Aim of these Practice Sessions

The students are reminded once again that

- There is no deadline for the learning tasks included in the drills. Tutors are available for help during the practice phase. Consult other sources for your learning as needed.

- Each student asks for a formal drill assessment when the student has completed the practice drill and the student is ready for the assessment. If student's assessment is incomplete at the end of a lab session, a fresh assessment problem will be assigned to the student in the next lab session. An assessment exercise does not continue over to the next session.
- The subject grading arrangements make progress past a stage inconsequential if the student goes past it without satisfactory learning.

1.3 Learning aims of Module 02 Stage 01

- Basic numerical values in C: `char`, `int`, `float`, `double`. Size qualifiers `long`, `short`. `signed` and `unsigned` values.
- Purpose and effects of the C declarations
- Header file: `limits.h` and `float.h`
- External (Human readable) representations of the basic numerical values
- Internal (computer) representations of the numerical values.
- Header file: `stdio.h`. Functions `printf()` and `scanf()`
- Conversions between the internal and external representations of the numerical values.

2 General background

In our early education, we learn to recognise symbols as alphabets and digits. We learn that digits can add. Alphabets cannot add to a new value. The skills and procedures for adding two single-digit values are by memorising the results. For example, 3 + 5 is 8. And, 5 + 6 is 11.

Later we learn a new set of rules. The new rules set helps us to add multi-digit numerical values. We learn a trick called *carry* that was not there in our previous skills. There is also a rule requiring that if two numerals have different number of digits, the numerals must be aligned on their rightmost digits before the digits in the numerals can add.

Our training on how to add numbers is not complete yet. We were introduced to the numbers with decimal points; these numbers have digits before and after a decimal point. Rules for addition are different now. The alignment is not pegged on the rightmost digits. Instead, the alignment is centred on the decimal points in the numbers.

Yet again we learn that there is one more add! This add is to add numbers written in a new style. To add numbers written in the scientific notation with a separate exponent, we first convert both numbers to have the same exponent value.

Many years of training, in the school mathematics, has helped us (but not all humans) to work through these different meaning of add with ease. However, remember different manifestations of add are used in the different circumstances based on how the values are expressed. Each manifestation has different sequences of steps to run to compute the result.

The single word *add* can mean any of the many ways of doing additions. Technical name for this phenomenon is *overloading* of the operator – that is, each manifestation of add runs different sequences of activities for the same goal depending on the representation of the values to be added. Further, the different forms or representations of the values are called their `type`. Algorithm (sequence of activities) chosen to compute the results depend on the types of the *operands* (values).

Computers are no different! All the jargon in the previous paragraph comes from the computer science books. Computers have electronic circuits to perform different kinds of add operations. The internal computer instruction to launch an add operation must clearly indicate which add circuit should be used (Computers also have circuits for dozens of other operators – subtract, multiply, divide, negate … and, so on).

Selecting the correct instruction to run inside the computer is one of the tasks performed by the compiler. The compiler needs to know the types of the values being used in each computation specified in the program.

Declarations in the programs provide type information to the compilers; the compilers need this information to select the correct instructions to do the operations specified in the C programs by the programmers.

3 DECLARATIONS

Declarations in a C program serve three essential purposes:

1. List the variables used in the programs.
2. Reserve and allocate locations in the computer memory for holding the values stored in the variables. You have already learned in a previous drill that a variable is a container to hold one value at any time.

   The memory allocation is done by creating two pieces of information in the compiler.

   a. The location (also called the *address*) of the memory allocated to the variable.
   b. Amount of memory space allocated to hold the value.

   The compiler needs to know the locations allocated to the variables, as generated instructions need to access these values from the locations containing them. Term access means either of these two uses of the memory: (a) to obtain the value from the memory to perform the computation, and (b) to deposit the result of the computation in the memory.

3. The third purpose of a declaration is to let the compiler know the form of the values the variable container holds. This is called *type* of the variable. Compiler remembers this information and uses it to choose the right internal instruction matching the computation described in the program. Type indicates how the pattern stored in the memory container should be interpreted.

A quick summary of a variable declaration in C is that it tells the compiler (a) what type of value the programmer holds in the variable; (b) how large is the allocated memory; and, (c) where in the computer memory the variable's value is stored.

4 NUMERIC VALUES AND COERCION OF NUMERIC VALUES

The set of numeric values in C are divided into two groups: Integers and Floating-points (There is a third group of numerical values for addresses with a specialised set of arithmetic operations that suit those address values. We will learn its details a few weeks from now. Until then you may ignore this value type except for a few brief mentions in this document.).

Our experiences and training from our schooldays arithmetic incline us to expect that the two groups have similar mathematical operators and properties. We have just explained that though the intrinsic meanings of these operations match our previous learnings, the algorithms to do the operations differ based on the types of the values involved. The difference is noticeable on the computers as the computational procedures (algorithms) are hard coded in the electrical circuits. You need to be aware of these differences. For example, 5.0/2.0 computes to 2.5. At the same time, 5/2 simplifies to an integer value 2.

The computer solutions of expressions may also be affected by the size of the container allocated to a variable for holding the values. The compiler cannot prejudge the actual value from a computation and therefore must choose the instructions to match the most unconstrained value for the variable possible. This is done to avoid the risks of errors during computations. This risk avoidance ensures that during a program execution, neither the final nor an intermediate result loses any part of a larger value due to incorrect choice of a container to hold the value or due to wrong choice of an instruction by the compiler. Hence, the programmers must declare the variable types carefully.

On the other hand, programmers may have a need to change a value of one type, into a value of another type. This is called type *coercion*. The compilers or the generated code automatically introduces some predictable coercions. Other coercions require the programmer to explicitly apply operators to change the type or size of the values.

For our immediate purposes, note the following example. In this example, program explicitly asks a `long` value to be coerced to an `int` value:

4.1.1.1 PROGRAM 1

```
#include <stdio.h>

/* Coercing a long int to int */

int main(void)
{
    long longOne, longTwo;
    int diff;
    longOne = 999999999;
    longTwo = 999999998;
```

```
        diff = (int) (longOne - longTwo);
        printf ("%d", diff);
        return (0);
    }
```

In PROGRAM 1, we determine the difference between two long integer values. We know that the difference is 1. However, the compiler cannot predict the value or its smaller size; the computed answer value 1 will emerge when the compiled program is run. So the compiler places the result in a temporary container that is suited for the large values (long declare the variable to be suitable for large integer values). We change that value to a normal size `int` value and print the value.

If you wonder why we coerced a `long` value to an `int` value, the answer is here: `printf()` as we have learned so far only knows how to print `int` values. We do not yet know how to print large size integer values (`long` values) correctly.

It is a good idea to run the code to see its behaviour on your computer.

4.2 OPERATORS TO FIND *LOCATION* AND *SIZE* OF A VARIABLE

Unlike other recent “safer and smarter” programming languages, C allows the programmers to know the *address* of the location allocated to a variable. Similarly, C also has an operator to find the size of the container given to a variable to store values.

We will practice these topics at a later stage. However, as part of our investigations into the variables representations on the computers, let us try a simple example below (See explanations later after the program):

4.2.1.1 PROGRAM 2

```
#include <stdio.h>

/* Find size (in number of bytes) for types */

int main(void)
{
    /* Try this with other types too.
       Replace double by (say) long or float */
    double one, two;
    int size;
    long longSize;
    int computedSize;
    size = sizeof (one);
    printf ("Size = ");
    printf ("%d", size);
    printf ("\n");
    longSize = (long)&two - (long)&one;
    computedSize = (int)longSize;
    printf ("Computed Size = ");
    printf ("%d", computedSize);
    printf ("\n");
    return (0);
```

```
}
```

Operator `sizeof` gives us the size of container used for holding the variable values. Operator `&` gives variable's address – the place in the memory where the container is located. However, arithmetic of addresses is unknown to us yet and we do not wish to explain it to you now. We use an alternate approach to explore the idea. We coerce the addresses to `long` integer values – type `long` is used to hold large values just in case an address is a large value. The difference of two `long` values obtained from the addresses is size of the container for variable `one`.

You must have noticed that we played a trick by declaring variables `one` and `two` next to each other. Then we used difference in their addresses to know the size of variable `one`. Such tricks are possible but are examples of bad programming! We did it just to make you aware that each variable, because of its declaration, is allocated space in the computer memory to hold values at a location in the memory. The distance between two locations is size of the space given to a variable.

Furthermore, note that the size of allocation computed by us and the operator `sizeof` matches when you run this code.

5 COMMON INTEGER TYPES

C defines many kinds of integer types. Some of these types are listed in the table below and you may wish to run PROGRAM 2 given in the previous section to determine sizes of each of these types. For this, you only need to change type-name in the declaration for variables `one` and `two` in the program.

Type	**Likely size**	**Common use**
`char`	1 byte	For small (<100) positive and negative values
`unsigned char`	1 byte	Small non-negative values. For values between 0 and 255.
`short int`	2 bytes	Positive and negative numbers in early 10-thousands. (<30000). Keyword `int` is optional.
`unsigned short int`	2 bytes	Positive values up to (say) 65000. Keyword `int` is optional.
`int`		Implementation dependent – either like `short` or like `long`.
`unsigned int`		
`long int`	4 bytes	Positive and negative 9 digit numbers. Keyword `int` is usually not written
`unsigned long int`	4 bytes	Positive integers of maximum value in early 10 digits. Keyword `int` is optional.

In the table, you might have noticed that a common C abbreviation practice is to skip keyword `int` in the declarations when used with qualifiers `long` or `short`.

Run PROGRAM 3 below to see the effects of qualifiers attached to the variable declarations. The declarations create the containers for the values and the values assigned to some of the variables exceed the capacity or interpretation of the values in them. Next section will explain the largest and the smallest values that a variable type can hold. Programs violating the constraints may compute incorrect results.

You will learn better if you type these programs yourself. Cut-and-paste methods reduce your exposure to the code and diminishes opportunities for you to learn.

5.1.1.1 PROGRAM 3

```
char charMinus128, char0, char127, char128, char255;
unsigned char uchar255;
charMinus128 = -128;
char0 = 0;
char127 = 127;
char128 =128;
char255 = 255;
uchar255 = 255;

printf ("char0 = ");
printf ("%d", char0);
printf ("\n");

printf ("char127 = ");
printf ("%d", char127);
printf ("\n");

printf ("charMinus128 = ");
printf ("%d", charMinus128);
printf ("\n");

printf ("char128 = ");
printf ("%d", char128);
printf ("\n");

printf ("char255 = ");
printf ("%d", char255);
printf ("\n");

printf ("uchar255 = ");
printf ("%d", uchar255);
printf ("\n");

return (0);
```


5.2 HEADER FILE LIMITS.H

Unlike most other programming languages, C is very closely tied to the computer where the code will be run. One area where we notice this dependence is in the integer value ranges that the different types can hold. The maximum and minimum values that various integer types can hold are defined in a header file named `limits.h`. There is a similar header file for floating-point types. The file is named `float.h` but we will not discuss the later header file any further.

Header file `limits.h` defines some macros (identifiers written in uppercase letters with a fixed computer-specific values) that give names to certain values (Just as we have `pi` or π to refer to value of 3.14159; or, `e` for 2.71828). Here are some example macros from this header file (if you need to use these macros, please add `#include <limits.h>` at the start of your program).

Purpose and meaning of each macro is easy to guess from the macro name.

INT_MIN	INT_MAX	LONG_MAX	SHRT_MIN
USHRT_MAX	CHAR_BIT	CHAR_MIN	UCHAR_MAX

6 HUMAN-READABLE AND IN-COMPUTER REPRESENTATIONS OF NUMERICAL VALUES

Humans represent numerical values in decimal notations (Using number 10 as the base). On the other hand, values are stored in the computers using binary formats (Number 2 is used as the base). Humans are rarely constrained by the limitation of space in writing their numbers. They are free to use unlimited amount of space to write their numbers.

Space available for values inside the computer is limited and fixed in size. Typical space is only enough for a few scores of binary digits called bits. Common sizes provided for storing values by the current generation computer hardware in number of bits are 8, 16, 32, 64 and 128. It is common in the modern computers to term 8 bits as a unit called byte – byte is the smallest collection of bits that internal instructions in the computer deal in. Dealing with individual bits inside the computers is possible but expensive on efforts.

For example, a char is made of CHAR_BIT bits. Which is enough for UCHAR_MAX different values. Write a program to print these values. (Remember header file `limits.h` if you are unable to recall how to print these values)

If a type `char` value is stored in a byte, there are 256 possible patterns that the byte can have. If each pattern is used to denote an integer value around 0, we have only enough patterns to cover values -128 to 0 to 127. One unique pattern of bits for each value.

If `int` is represented by 32 bits (4 bytes), it can cover range from -2147483648 to 0 to +2147483647. This is so because each pattern represents one different value and the neighbouring patterns represent values that are set 1 unit away from each other.

Floating-point representations use the same 4294967296 patterns differently to represent values in the range $1.2*10^{-38}$ to $3.4*10^{+38}$ and their negative versions $-1.2*10^{-38}$ to $-3.4*10^{+38}$. There is a separate special pattern to denote `0`. This wide range of values is possible because the distance between patterns for adjacent values is not fixed. Adjacent patterns for the small values differ only by a small value. Patterns associated with the larger values differ from each other by values much larger than 1. So by making the relative percentage differences similar (but not the same) across the full range, growth advantages of the geometric progression is used to support a larger range than the one supported by the integer types.

The integer types use arithmetic progression where nearest numbers are separated by 1 but every integer value in the range has a unique pattern to represent it. Floating-points support a huge range of values but many numbers in the range do not have representation on the computers. These numbers are approximated by a near value that can be represented in the scheme. The scheme, however, guarantees that the relative error is extremely small. The error is called truncation error – trailing bits necessary for a better representation of the number are lost due to the size limitation on the available bits in the allocated memory.

We can ignore the details of these representations. CS101 will cover the topic in greater details.

However, we do need ways to convert representations understandable to humans in the real world and the way these values are represented inside the computers. C provides functions in standard library `stdio.h` for this purpose.

7 INPUT-OUTPUT OF NUMBERS USING STANDARD LIBRARY STDIO

The library is not new to you as it was used in a previous drill. We also know that to use this library we must place `#include <stdio.h>` at the start of the program. Now we will study format specifiers for functions `printf()` and `scanf()` in some details.

7.1 FUNCTION PRINTF()

The `printf()` is one of the several functions included in the C standard library. The library is guaranteed to be available with every C implementation.

The function enables us to print the values available in the computer programs written in C, on the monitor screens. The official name for the `printf()` destination is `stdout` or standard output stream.

The function is designed such that it can take one or more arguments – these arguments are listed in the parentheses-pair that follows function-name `printf`. A comma is used to separate each argument.

The first argument is special for this function and is normally written as a quoted string – the quotes at the beginning and end of this string must be double quotes (`"`). This string is printed on the screen *as is* except for a few special substrings in it. These special substrings in the quoted string either indicate some special-effects or are place-holders for the values specified in the second and later arguments of the function. Since this string sets a pattern or format for what appears on the screen, we will also refer to this string as the format string or just format.

Often special substrings included in the format string stand for effect-causing rather than viewable symbols. These special substrings are made of 2 to 4 characters; the first of these characters is \ (backslash). Some commonly useful 2-symbol patterns are:

\a	Bell sound – a is for alert
\n	start of the next line
\\	Backslash --needed because a single \ has been given a special use
\?	Question mark – ? is also used for a special task
\’	single quote symbol ‘
\”	Double quote symbol – without \ in front it ends the format string!
%%	Symbol % -- Two % together in the format string means one % in print.

We will learn about longer phrases used to denote single symbols at a later stage.

Other, embedding in the format string argument of `printf()` begins with % and denotes a substitution. Substitution is a location in the format string where a value will be inserted at the time of printing it on the screen. The substitution marker (`%`) and a few related symbols after it are replaced by the value of the an argument of `printf()`. The matching of the arguments of the function and the substitution markers is done in a sequential left-to-right manner. First `%` in the format string matches with the first argument after the quoted-string. And, so on.

It is your responsibility, as a programmer, to make sure that there are exactly the same number of substituting `%` symbols in the format string as there are arguments after the format string. However, `%%` needs no matching argument as it does not denote a substitution; it represents a (single) symbol `%` in the on-screen display.

The substitution introduced by symbol `%` in the format string needs more information. This information is used to advice the function `printf` of your instructions about the displayed values.

Remember, each conversion has a matching argument which can be a variable, an expression or a value. These coding artefacts denote a value internal to the program (computer). The internal values are stored as binary value patterns which need to be interpreted using their types. A single binary pattern may represent different numerical value depending on its type. The programmer must qualify the substitution symbol `%` in format to indicate the kind of conversion they desire. If the value type and the programmer specified conversion do not match, the compiler may issue a warning message but will convert the binary pattern using the conversion the programmer specifies through the format string.

Example (run this program on your computer):

7.1.1.1 PROGRAM 4

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

```
/* Printing value through incorrect conversion */

int main(void)
{
    float f = 1.4E17;
    printf ("%d\n", f);
    printf ("%f\n", f);
    return (0);
}
```

You will notice that the compiler issues a warning to help you understand the problem. The values printed by two `printf()` statements are different. In fact, the printing from the first statement is unrelated to the assigned value – a floating-point value of type `double` being interpreted as an integer. The second value is close to the expected value but not the same. As said previously, not every floating point value has an internal representation. This should explain why the second value is not exactly what may be expected.

A careful reader would have noted that drill mentioned type `double` for the value being printed! Yes. It is right. Function `printf` coerces every `float` value to `double` silently before applying the conversion named by the programmer in the format string for display.

8 PRINTF CONVERSIONS FOR DECIMAL REPRESENTATIONS

There are several conversions specifications that one can write within a format string. However, in this section we will restrict ourselves to learn only the conversions related to the decimal representations that humans commonly use.

The conversion specifiers of an internal binary value to the matching on-screen decimal display of the value of interest to us in this drill are `d`, `i`, `f`, `e`, `E`, `g`, `G`, `u` and `c`. We have already used `%d` and `%f` in our previous programs. These characters are also called *conversion characters* as they specify the conversion algorithm to convert the internal binary representation to an on-screen decimal notation or symbols.

8.1 CONVERSIONS FOR INTEGER VALUES

Include `%d` in the format string to indicate that the value available at the corresponding argument in the `printf` function call should be treated as a value of type `int` and converted to a standard decimal value before displaying on the monitor screen. The decimal value replaces `%d` in the format string when the string is printed on the screen.

The value will be printed using just as many places on the screen as needed to print all useful digits in the number and a negative symbol, if the value is negative. This is most common act that a programmer desires! Conversion `%i` is an alternate way of specifying the same conversion actions. If you seek to treat the internal binary value as an unsigned `int` value, use conversion `%u` in the format string. The default actions for these conversions are useful and easy in the most circumstances.

However, sometimes we may seek to print the integer values in a different style. For example, we may want to print sign + before the value. Or we may seek to fill a fixed amount of space on the screen by placing blanks or 0s in the unfilled leading positions. How do we display `long int` values? Do we want value to be justified to the left edge or the right edge of the field? We will come to these advanced features after you have tested and understood the basic conversions `%d`, `%i` and `%u`.

Practice the conversions included in PROGRAM 4 below for different declared types for variables `a`, `b` and `c`.

8.1.1.1 PROGRAM 5

```
#include <stdio.h>

/* Conversions %d, %i and %u */

int main(void)
{
    short int a = 100;
    int b = -20000;
    unsigned int c = 2999999990;
    printf("short int a using %%d = %d\n", a);
    printf("int b using %%i = %i\n", b);
    printf("unsigned int c using %%d = %d\n", c);
    printf("unsigned int c using %%u = %u\n", c);
    return (0);
}
```

Try other values for the variables in the above program to learn how different conversions interpret the type of the value and use the conversion specified in the format string to create decimal value for display on the screen.

8.2 MODIFYING DEFAULT CONVERSION BEHAVIOUR

To alter the default behaviours of the conversion characters, modifying symbols are added between the character `%` and the conversion character in the format strings. Thus, full conversion specification in a format string begins with character `%` and ends in a conversion character. In between these two characters we include modifying flags, field-width specifier, and type-length specifier in this order.

8.2.1 Flags:

-	Left justified printing of the number in the field. See later for how to specify a field size.
+	Always print the sign of the number
Space	Leave a space for sign even if no sign is printed
0	Fill leading 0s in the field.

8.2.2 Minimum Field Width:

A numerical value after the flags denotes MINIMUM field width for the output value. The padding can be a space (default) or 0 if this was indicated through flags.

8.2.3 Type Length Modifier:

h To indicate a `short` or `unsigned short` is being converted.

l To indicate the value being converted is `long` or `unsigned long`.
It is lowercase letter L.

Note that not every combinations of the flags, field width and length modifier is a sensible combination. Use only meaningful modifications in the conversion specifications. Most compilers are sophisticate and warn the programmers about the bad combinations.

8.2.3.1 PROGRAM 6

```
#include <stdio.h>

/* Conversions %d, %i and %u with modifiers*/

int main(void)
{
    short int a = 100;
    printf("Using %%-7d = %-7d\n", a);
    printf("Using %%-07d = %+07d\n", a);
    printf("Using %%- 7d = %- 7d\n", a);
    printf("Using %% 7d = % 7d\n", a);
    printf("Using %%-+7d = %-+7d\n", a);
    printf("Using %%hd = %hd\n", a);
    printf("Using %%hd = %hd\n", 99999);
    return (0);
}
```

9 CONVERSIONS FOR FLOATING-POINT VALUES

Floating-point values are written with or without exponent when using a decimal notation. Format `%f` that we used previously has default style in which 6 digits are printed after the decimal dot symbol and has the general format `mmmmmm.dddddd` with a possible minus sign (−) in front, if necessary. All modifiers specified for the integer conversions previously can also be used with `%f`.

Field-width specification has one additional feature of interest when used for formatting a floating-point value. The field-width specification can include a decimal-point after the (minimum) width number followed by the precision of the on-screen display value. Precision of 0 would suppress printing of the decimal-point in the displayed value – in this case the printed value will be a whole-number.

9.1.1.1 PROGRAM 7

```
#include <stdio.h>

/* Conversions %f with modifiers*/
```

```
int main(void)
{
    float a = 100;
    printf("Default %f\n", a);
    printf("%%5f %5f\n", a);
    printf("%%10.0f %10.0f\n", a);
    printf("%%5.2f %5.2f\n", a);
    printf("%%07.1f %07.1f\n", a);
    return (0);
}
```

9.1.2 Questions for students to explore:

1. Is width field specification for the complete displayed value or just for the whole number part of the display?
2. Is the decimal point counted in the width field?
3. Another very important question for which you must find answer is the way your computer rounds the floating-point values it displays. (Though the right place for this information is in header `float.h`, students may print a few example values to get some view on this question. Header file `float.h` is too advanced for this class.)

Should you wish to display a `long double` value insert letter `L` before the conversion character `f` in `%f`. No additional modifier is needed for the other floating-point types as they are coerced automatically to an equivalent `double` value due to standard C protocols for the function arguments.

9.2 Display using Scientific Notation

Display of internal floating-point values in the normalised scientific notation is done using conversion characters `e` and `E`. These characters (`e` or `E`) in the displayed value separate normalized precision-value from the exponent value in the display. The normalised representation of the floating value has the form `m.ddddddE±xx` or `m.dddddde±xx.`

The modifiers applicable to conversion `%f` apply to `%e` and `%E` conversion specifications too. There is an interesting variation for the floating-point conversion; it is expressed as %g and %G. Some say that `g` or `G` denote good form! In this format the display will be either based on `%f` or `%e`/`%E` variant for the on-screen display based on what is a better display for the value.

Here is a practice program for you to try:

9.2.1.1 PROGRAM 8

```
#include <stdio.h>

/* Conversions %g with modifiers*/

int main(void)
```

```
{
    float a = 100999;
    printf("Default %g\n", a);
    printf("%%5g %5g\n", a);
    printf("%%10.0G %10.0G\n", a);
    printf("%%15.4G %5.4G\n", a);
    printf("%%07.1G %07.1G\n", a);
    return (0);
}
```

10 FORMATTED INPUT

Function `scanf` is counterpart of `printf` for input of data from the keyboard – actually, from `stdin` stream.

Once again format string defines the expected input. Blanks, tabs and other white spaces in the format string play a role to ease the input specification. White space specifiers in the format string are usually interchangeable and match any length white space in the input stream. Thus, a single space in the format string would allow the input stream to have as many spaces as the user typing the input inserts on the keyboard. A whitespace can even include newlines!

Conversion characters in the format string are written with a `%` preceding them just like those in the `printf ()` calls. The modifying flags, field width values and type size symbols are similar to those used for function `printf`. However, some differences need to be noted to account for the direction of data/value flow – when using `scanf` data flows from a human user to the computer.

One major difference between `printf` and `scanf` is about the arguments that are listed after the format string (2^{nd} and subsequent arguments) in function call `scanf()`. These arguments must be the destination addresses! We already know that this is easily done using operator `&` before the variable name!

It was explained previously that C variables have a fixed amount of space allocated to them. The value being send to a destination in the `scanf ()` call should meet this restriction. For example, a `long` value cannot fit in an `int` variable. Such a mismatch may cause a wrong value assigned to the variable by `scanf()` input.

Other difference between `printf` and `scanf` formats is the field width specification – `scanf` uses the specified size as the maximum width of the field in the input stream. `Printf` used the value as the minimum width of the field in display.

The type length modifiers determine the size of the type to which the value from the `stdin` stream is being converted to. It was previously mentioned that mismatch in the length modifier in the conversion specification and type of the destination variable may cause errors in the read value.

However, the convertors are smart to easily understand different versions of floating-point values keyed in decimal notations on the keyboard. Thus conversion characters `f`, e, `E`, `g`, and `G` behave in the same fashion.

Normally, first character that is not appropriate for a conversion stated in the format string marks the end of the input item being received by a convertor. The unprocessed input character will be handled as per the specifications in the format string after the terminated conversion.

Let us understand these issues using an example program.

10.1.1.1 PROGRAM 9

```
#include <stdio.h>

/* Reading values from stdin */

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

    printf("Three Inputs separated by \";\":");
    scanf ("%g ; %f ; %G", &a, &b, &c);
    printf("a in default %g\n", a);
    printf("b in %%5g %5g\n", b);
    printf("c in %%10.0G %10.0G\n", c);
    printf("a in %%15.4G %5.4G\n", a);
    printf("c in %%07.1G %07.1G\n", c);
    return (0);
}
```

The program seeks to read three floating-point values – unlike the case of `printf`, here the values should be suitable for a variable of type `float`. This is so because variables `a`, `b` and `c` are of type float.

There should be a semi-colon after each of the first two numbers. This is because of the semicolon (`;`) symbols appearing in the format string. The specified format string permits any amount of space to be inserted between the numbers and the separating semicolons. And also, after the semicolons. The space included in the format string can stretch to any size!

The version of PROGRAM 9 below will not read the values correctly if either of the first two keyed numbers do not end with semicolons.

10.1.1.2 PROGRAM 10

```
#include <stdio.h>

/* Reading values from stdin
   A correct sequence: 1; 2; 3
   AN incorrect sequence: 1.1 ; 2.2 ; 3
*/
```

```
int main(void)
{
    float a, b, c;
    printf("Three Inputs separated by \";\":");
    scanf ("%g; %f; %G", &a, &b, &c);
    printf("a in default %g\n", a);
    printf("b in %%5g %5g\n", b);
    printf("c in %%10.0G %10.0G\n", c);
    printf("a in %%15.4G %5.4G\n", a);
    printf("c in %%07.1G %07.1G\n", c);
    return (0);
}
```

Please note that in their normal operational modes, Unix system does not forward the typed characters of input to the program till the user presses return or enter key on the keyboard.

11 More on Type Char

We will practice a little more on integer values of type `char` before concluding this drill. You would recall that type `char` fits a single byte of memory. A byte has 8 binary digits (bits) which can hold 256 different bit patterns. `Char` is an integer type and as such is able to participate in arithmetic and other integer-based operations.

There is another interesting use of variables and constants of type `char`. The 256 different patterns of bits that a char value can hold, are also used to represent symbols – mostly printable characters but also many control characters are represented as `char`.

In C programs, a character symbol written within a pair of single quotes denotes the pattern associated with the character. For example, `'a'`, `'x'`, `'?'`, `'A'`, `'7'` are each different and distinct patterns with `unsigned` values between 1 and 255.

Please note that the pattern `'a'` is different from the pattern `'A'`. One can represent lowercase as well as uppercase letters without any mix up. There are patterns to represent digits. There are patterns to represent the punctuation marks.

Some patters are used for control purposes. For example, `'\n'` is used to denote a newline character; without this special specification arrangement it is awkward to specify a newline character in a format string.

A new programmer is sometimes confused to note that digits have bit patterns that are different from the bit patterns for their numerical values! Thus, the value for number 7 and character `'7'` are represented by different bit patterns. You should not be alarmed by this! We have already explained previously that what matters is not the pattern stored in the memory but how it is interpreted.

Function `printf` provides a very convenient way to explain the effects of various interpretations of a pattern. The conversion character to convert internal binary representation into an on-screen character representation is `c`. The program below uses this conversion character in the format string.

11.1.1.1 PROGRAM 11

```
#include <stdio.h>

/* Char constants, and their input-output */

int main(void)
{
    char chr1 = 'A'; /* Insert a bit pattern */
    char chr2 = 100; /* Insert a bit pattern */

    /* Print the bit patterns using different
       Conversion methods on computer screen
    */
    printf("chr1 has decimal value %d\n", chr1);
    printf("chr2 has decimal value %d\n", chr2);
    printf("chr1 pattern is character %c\n", chr1);
    printf("chr2 pattern is character %c\n", chr2);
    return (0);
}
```

The example below shows how to read a character from a computer keyboard:

11.1.1.2 PROGRAM 12

```
#include <stdio.h>

/* Char constants, and their input-output */

int main(void)
{
    char chr1;
    short s;

    printf ("Please type an alphanumeric character:");
    scanf(" %c", &chr1);
    printf("Character %c has code %u\n", chr1, chr1);

    printf ("\nPlease type a value between 33 and 126:");
    scanf("%hd", &s);
    chr1 = (char) s;
    printf("Code %u is for character %c\n", s, chr1);
    return (0);
}
```

Reading of characters requires some sophisticate understanding of some issues. It is generally helpful to place a space before `%c` in the format strings used in function `scanf`.

You will be taught about it in detail in CS101 classes later in the semester.

12 Test Exercise Completion Checklist

1. Does the program have appropriate comments that help in understanding the program code?
2. Is the amount of comments in the program appropriate? That is, the amount of comments is neither too little nor too much.
3. Is the name of the programmer and date of creation included in the demonstrated program?
4. Are the identifiers used as variables helpful and describe the variable use correctly?
5. Are all constants in the program from exercise statement? For every accepted program, the output must be computed from input to the program and the constants included in the program code from the exercise statement.
6. Are the variable type declarations appropriate?
7. Is the program correctly indented and it is easy to read and understand?
8. Does the program run correctly?

12.1 Test Problem Description

The topics covered under this stage are central to virtually every program that you will write. The problems in the assessment set are going to test you only on some of the skills that you must master.

A more comprehensive assessment would have been appropriate, but that would extend this drill and assessment beyond the 3 hours lab-time span we have. You should consider doing some more assessment exercises after the lab session to further improve your skills.

13 Welcoming Your Suggestions

I am not apologising for the lengths of my drill documents. The drills definitely demand substantial preparation before the lab sessions. They demand focus and attention in the labs. But, then they prepare the students to a level that the module could demand. Those who follow the drills with honesty, sincerely and diligence should have no fear of an exam time disappointment.

Please send comments and/or errors you notice in this document to vmm@iitg.ernet.in. Your messages will help us improve the quality of experience students have in CS110 labs.