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

1 IMPERATIVE STATEMENTS

1.1 INTRODUCTION: MODULE 01 STAGE 02

Students should attempt this stage only after they have successfully completed their training using drill titled Module 01 Stage 01. The previous drill trains the students to work with a Unix system through a command-based shell interface. The students who continue to use computers in their future education and careers will learn sophisticated ways to use Unix shells.

Please ensure that grade for your successful completion of the previous stage has been correctly entered on the course records. The new drill that you are about to practice will teach you to write C programs using very simple constructs. These techniques will enable you to perform minor but meaningful computations.

In this drill, you will complete over a dozen practice exercises and programs included in this document. After you have completed the practice exercises, you will be assigned an assessment exercise by your tutor from the set of exercises designed for this stage. If you cannot complete the assessment exercise before the end of your lab session, a different assessment exercise will be assigned to you in your next lab session.

Only one stage is assessed in any one lab session. Successful completion of a stage is recorded in the course records after you demonstrate successful completion of the assigned assessment exercise to a tutor.

After the assessment, you continue trainning and learning using other assessment exercises in the set. This will help you in acquiring a better grasp of the topics covered in this stage. On the other hand, some students may start practice part of the next training stage in preparation for an assessment in a future lab session.

Please note that the tutor assessing you will not accept an exercise completion unless ALL criteria (NO EXCEPTION ALLOWED) listed in the checklist (listed later) are pass in a single demonstration to a tutor. Some of these checks are very easy to correct but you must make the necessary changes. A fresh full demonstration to a tutor will be scheduled after you have made the changes to meet the requirements of the checklist.

The tutors are busy and you may face delays in getting their attention for a repeat demonstration of your exercise. It is, therefore, a good idea to complete a self-check using the suggested checklist even before you seek a formal assessment from a tutor.

We acknowledge that some students have been trained in programming previously. Even if you have previous background in computer programming, please do not use the features of programming language C other than those suggested in this stage.

1.2 Remember: Learning is the Goal of these Practice Sessions

There is no time-limit or deadline for the learning tasks included in the practice drills. Each student practices the exercises in the drill to suit their learning preferences. Students will receive support and guidance from the tutors for their learning needs in all practice phases.

A student asks for an assessment when the student has completed the drill and the student is ready for an assessment. Again, assessment exercise does not impose any time limit except the end-time of the lab session. Unfinished assessment exercises do not continue over to the next lab sessions. Students will be given new assessment exercises in the following lab sessions. Obviously, the student will get only limited help from the tutors during the assessment phases. Contacts with others persons during the assessment phases is not permitted.

Temptation to use unfair means to complete assessment of a stage will not benefit the student much. The grades in the subject are determined primarily from the formal mid-semester and end-semester examinations. Students who progresses past a stage (or a module) without learning the topics are not likely to pass the examination exercises. Students are required to repeat the trainings for the modules that they do not pass in the mid-semester examination. End-semester examination also repudiates the benefits of progress past a module that the student has not learned well.

1.3 Learning aims of Module 01 Stage 02

- Explanatory and provenance comments in the programs;
- Variable name selection;
- Understand C types: `int` and `float`.
- Imperative view of C programs
- Printing program output on monitor screen
- Arithmetic expressions
- Common arithmetic operators and their precedence and associativity rules
- Assignment statements and sequential execution of the program statements.

2 General background for drill

A simple C program consists of a few standard `#include` directives and a single function `main()`. The function body of each program in this drill is made of a number of variable declarations, assignment statements and calls to standard input-output library functions to print program results on the computer screen. At this stage, we will only use a few very simple output functions and will not read any input data from the computer keyboard to the programs. (However, one example program does some data input for demonstration purpose.)

Listed below is a program that you can use to start learning about program codes.

2.1.1.1 PROGRAM 1

```
#include <stdio.h>

/* Your name and roll number */
/* Date and other relevant information */

int main (void)
{
  /* Declarations */
  int a;
  float b;

  /* Imperative actions */
  printf ("a = ?");            /* Prompt user for input */
  scanf ("%d", &a);            /* Read value from keyboard */
  b = a + 10;                  /* Compute b */
  printf ("b = %f \n", b);     /* Print value of b on screen */
  return (0);
}
```

The program structure above is very basic and uses only a few C programming constructs. However, you will learn through the exercises in this drill that these program constructs are enough to run a number of useful computations.

C belongs to a class of programming languages called imperative languages – statements in the programs are detailed step-by-step advices similar to the one you would give to your younger sibling to complete a maths homework. You will notice this imperative nature in all practice programs.

An elementary description of function `printf()is` helpful here. Function `printf()` prints the quoted string of characters included in the parentheses pair. Into this string, it adds an `int` value where `%d` is shown and inserts a `float` value where `%f` is written. You can see examples of `printf ()` call in PROGRAM 1 above.

Function `scanf()` is the counterpart of function `printf()` for reading the values into the programs. The values read are assigned to the variables specified by the programmers. Function `scanf` will be discussed in a later drill.

We will explain a few simple variations of function `printf()` that are enough to support practice exercises in this drill. Further details, however, are subject of the next drill.

3 TEST EXERCISE COMPLETION CHECKLIST

The tutors will use the following checklist to assess completion of the stage on the assessment exercise assigned to the students.

1. Does the program include appropriate comments to help understanding of 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 in understanding the program and the variables are used correctly and consistently to their purposes?
5. Are the variable type declarations right?
6. Is the program correctly indented and is it easy to read and understand?
7. Does the program run correctly?

3.1 Assessment Problem Description

This section describes the common background information and arrangements applicable to all assessment problems in this set (Module 1 Stage 2). Each of the other documents in this stage describes a problem for you to write a program. These are the problems/exercises created to assess students completing this stage.

Each problem description in these documents contain a guidance section to help the budding programmers to write the program. This help is needed for this stage as most students do not have past experiences in writing programs.

It should be noted that the examination questions will be similar to these assessment problems. However, examination problems will be without the guidance section or advisory comments. In the examinations, the students will be expected to understand the problem statement given to them, create a solution, and demonstrate the program solving the given problem to their examining tutors.

4 Some Programming Practice First

Before proceeding to an assessment exercise, this document helps you to learn the relevant programming skills.

A number of programs and program segments are listed in below. Please construct programs using these code fragments as the body of function `main()` on your computer.

Carefully read the codes of the programs to understand them before testing the programs on your computers. To support your learning and educational goals consult your class notes, CS101 textbook and other C programming books, your tutors, and even your friends (Note: you and your friend can discuss any problem during the training but you cannot do so while sitting on a computer desk. You must go to a separate area set for this purpose away from the computers). Two students cannot be together on any lab computer desk.

The first program given below, reads data from the computer keyboard. To run this program you must provide an input data (Suggested value to type: 100). As a rule, in all drills, any program titled as PROGRAM `nn` or PROGRAM CODE FRAGMENT `xx` should be created as a program by the student. The student should read the program code carefully to understand its nuances and run the program to verify their understanding as part of their drill tasks.

5 PRACTICE INPUT AND OUTPUT STATEMENTS AND SEQUENTIAL EXECUTION

In this section we practice some example programs that print output on the monitor screen.

The first program below is the same program that we listed previously – it features a typical behaviour of a program showing input, processing and output phases. The program reads data from your keyboard. To run this program you must provide an input data (Suggested input: 100).

The later programs are listed as code fragments. Each code fragment is body of function `main()`. You can construct a C program from these fragments by inserting a fragment in function `main ()` of your first program. The previous code in the function is not needed and must be deleted. (Note: some fragments have later been replaced by full program to ease the reading of the document)

As you read the codes, you notice that activities listed in the program statements occur in the order in which the statements are listed in the programs.

We do not practice input-output in full details till a later training stage. A brief description of four variants of function `printf()` are provided here for you to be able to use the function to do program outputs with ease during this practice and during the assessment phase of this drill.

To explain these four variants, for each variant, an example is listed first and then its effects explained.

```
printf ("prints this message as is on screen ");
```

The message in the quotation-pair is printed as is on computer screen. The quoted messages are called a string. If two print statements are run one after the other, their outputs appear on screen in order in which they are listed in the program and no additional character is added between the two output strings.

```
printf ("%d", variable_or_expression_here);
```

The value of the variable or expression shown after comma (,) is assumed to be an `int` value and this value is printed after any previously displayed output on the monitor screen.

```
printf ("%f", variable_or_expression_here);
```

The value of the variable or expression shown after comma (,) is assumed to be a floating-point value and the value is printed after any previous printing on the monitor screen.

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

Use of this variant of `printf ()` call moves the next output location on the monitor screen to the next line.

In a later stage, we will learn to combine these actions in a single `printf` action. Run the following programs and program code fragments to learn the use of the lessons related to printing of program output on computer monitor screens.

5.1.1.1 PROGRAM 1

```
#include <stdio.h>

/* Your name and roll number */
/* Date and other relevant information */

int main (void)
{
  /* Declarations */
  int a;
  float b;

  printf ("a = ?");              /* Prompt user for input */
  scanf ("%d", &a);              /* Read value from keyboard */
  b = a + 10;                    /* Compute b */
  printf ("b = %f \n", b);       /* Show value of b on screen*/
  return (0);
}
```

5.1.1.2 PROGRAM CODE FRAGMENT 2

```
  int i, j;
  int d;

  /* Assign a test value to variable i */
  i = 125670;
  j = i/100; /* Shift digits in I and lose 2 digits */
  d = j % 10;

  printf ("%d", d);  /* Print answer digit value */
  printf (" is at significance 100 in "); /* add message */
  printf ("%d", i); /* Print original number with message */
  printf ("\n");  /* Go to next line */
  return (0);
```

5.1.1.3 PROGRAM 3

```
#include <stdio.h>

int main(void)
{
    int paise = 123;
```

```
    int rupee, remaining;

    printf ("%d", paise);   /* Print amount is paise */
    printf (" is equal to ");  /* Print message */
    printf ("Rs");     /* Print symbol Rs */
    rupee = paise/100; /* 100P is Rs1 */
    printf ("%d", rupee);    /* Print rupee value */
    printf (".");     /* Print decimal point */
    remaining = paise % 100;
    printf ("%d", remaining);   /* Print paise value */
    printf ("P");   /* Print symbol P for paise */
    printf ("\n");  /* Move to next line on print screen */
    reurn (0);
}
```

PROGRAM 3 does not work well sometimes. For example, it prints `102P` as `Rs1.2P` instead of the correct output `Rs1.02P`.

The correct version is given as PROGRAM 4.

5.1.1.4 PROGRAM 4

```
#include <stdio.h>

int main(void)
{
    int paise = 12307;
    int rupee, remaining;

    printf ("%d", paise);   /* Print amount is paise */
    printf (" is equal to ");  /* Print message */
    printf ("Rs");     /* Print symbol Rs */
    rupee = paise/100; /* 100P is Rs1 */
    printf ("%d", rupee);    /* Print rupee value */
    printf (".");     /* Print decimal point */
    remaining = paise % 100;
    printf ("%d", remaining/10);  /* Print  10s of paises */
    printf ("%d", remaining%10);  /* Print 1 paises value */
    printf ("P");   /* Print symbol P for paise */
    printf ("\n");  /* Move to next line on print screen */
    return (0);
}
```

6 STATEMENTS AS IMPERATIVES

Variables in mathematics (Algebra) are used to denote a fixed but unknown or unspecified value. The order of equations listing does not matter in mathematical descriptions.

Variables in a programming language are not unknowns. They are locations in the computer memory that can hold values. In fact, using a variable that does not have a value is a poor programming practice that often causes programming errors.

The program variable can change its value through a new assignment of a value to the variable. Only one value assigned to a program variable is the value of the variable. This value is the last or the most recently assigned value to the variable.

In mathematics it is wrong to say:

```
X = 4;
X = 7;
```

Programs just replace the old value with the new one!

6.1.1.1 PROGRAM CODE FRAGMENT 5

```

    int y;
    int x;

    y = 1;
    x = 10;

    printf ("At first print y is ");
    printf ("%d", y);

    y = x;
    printf (". At the second print y is ");
    printf ("%d", y);
    printf ("\n");

    y = 1;
    printf ("At third print y is back to ");
    printf ("%d", y);
    printf (".");
    printf("\n");
    return (0);

```

6.1.1.2 PROGRAM CODE FRAGMENT 6

```
#include <stdio.h>

int main(void)
{
    int y;
    int x;

    y = 1;
    /* We no more do x = 10; */

    printf ("At first print y is ");
    printf ("%d", y);

```

```
    y = x;
    printf (". At the second print y is ");
    printf ("%d", y);
    printf ("\n");
    return (0);
}
```

Please carefully read te messages from your compiler after compiling this program.

7 CONDITIONAL EXPRESSION

Very often, we need to express computations as combines of two separate and disjoint segments. For example:

$$\text{Tax\_rate} = \begin{cases} 0\% & \text{if Income} < 500000 \\ 10\% & \text{Otherwise} \end{cases}$$

C has an expression to represent such cases,

```
condition?ifTrueValue:ifFalseValue
```

For example, we can write a code based on the above tax rate as follows:

```
float income = 600000.00;
float tax_rate;
tax_rate = income<500000?0.00:0.10;
```

Following programs provide some further practice in the use of conditional expressions.

7.1.1.1 PROGRAM 7

```
#include <stdio.h>

int main(void)
{
    /* Find largest of the three values */
    int a = 10, b = 240, c = 30;
    int part_max, max;

    printf ("Largest of numbers ");
    printf ("%d", a);
    printf (", ");
    printf ("%d", b);
    printf (", and ");
```

```
    printf ("%d", c);
    printf (" is ");
    part_max = a>b?a:b;  /* Find larger of a and b */
    max = part_max>c?part_max:c;
    printf ("%d", max);
    printf ("\n");
    return (0);
}
```

7.1.1.2 PROGRAM 8

```
#include <stdio.h>

int main(void)
{
    /* Find smallest of the three values */
    int a = 10, b = 240, c = 30;
    int part_min, min;

    printf ("Smallest of numbers ");
    printf ("%d", a);
    printf (", ");
    printf ("%d", b);
    part_min = a<b?a:b;  /* Find smaller of a and b */
    printf (", and ");
    printf ("%d", c);
    min = part_min<c?part_min:c;
    printf (" is ");
    printf ("%d", min);
    printf ("\n");
    return (0);
}
```

8 PRECEDENCE AND ASSOCIATIVITY OF COMMON ARITHMETIC OPERATORS

Arithmetic operators in C have precedence (which operator gets to perform its action first) and associativity (location of the operator in the text of the expression determining the order in which same precedence operators get to perform their actions.)

Important caution: C rules may not be same as those you learned in your Mathematics class.

Like Mathematics, C expressions also use parentheses-pairs to alter the order of application of the operators in the expressions determined by the precedence and associative rules.

There is an implicit operator that converts, values of type `int` to values of a `float` type when operands to these types come together in an operation. This conversion, however, is only applied if needed. Some of the examples below, will draw your attention to this issue.

The following program code fragments should all compute the same answers if programs were correct. Some code fragments, however, do not derive the correct programs. You must find the reasons for the differences in their outputs and make changes in the programs to correct mistakes.

8.1.1.1 PROGRAM 9

```
#include <stdio.h>

int main(void)
{
    float ans;
    ans = 6.0/2.0*1.0/4.0*3.0/5.0/2.0/2.0/2.0/2.0/2.0;
    printf ("Answer = ");
    printf ("%f", ans);
    printf ("\n");
    return (0);
}
```

8.1.1.2 PROGRAM CODE FRAGMENT 10

```
float numerator, denominator, two2power5;
numerator = 6.0*1.0*3.0;
denominator = 2.0*4.0*5.0;
two2power5 = 2*2*2*2*2;
printf ("Answer = ");
printf ("%f", numerator/denominator/two2power5);
printf ("\n");
return (0);
```

8.1.1.3 PROGRAM CODE FRAGMENT 11

```
float ans;

ans = 6.0*(1.0*3.0)/(2*4*5)/(2.0*2.0*2.0*2.0*2.0);
printf ("Answer = ");
printf ("%f", ans);
printf ("\n");
return (0);

```

8.1.1.4 PROGRAM CODE FRAGMENT 12

```
float ans, numerator, denominator;

/* But this does not print correct answer! */
numerator = 6.0*(1.0*3.0);
denominator = (2*4*5)/(2.0*2.0*2.0*2.0*2.0);
ans = numerator/denominator;
printf ("Not a correct answer = ");
printf ("%f\n", ans);
printf ("\n");
```

```
return (0);
```

8.1.1.5 PROGRAM CODE FRAGMENT 13

```
float ans, numerator, denominator;
/* This does print the right answer */
numerator = 6.0*(1.0*3.0);
denominator = (2*4*5)*(2.0*2.0*2.0*2.0*2.0);
ans = numerator/denominator;
printf ("Answer = ");
printf ("%f", ans);
printf ("\n");
return (0);
```

8.1.1.6 PROGRAM CODE FRAGMENT 14

```
float ans, numerator, denominator;
numerator = 6*1*3;
denominator = (2*4*5)*(2*2*2*2*2);
ans = numerator/denominator;
printf ("Answer = ");
printf ("%f", ans);
printf ("\n");
return (0);
```

8.1.1.7 PROGRAM 15

```
#include <stdio.h>

int main(void)
{
    float ans;
    /* This does not do the computation correctly */
    ans = 6*1*3/(2*4*5)*(2*2*2*2*2);
    printf ("Not a correct answer = ");
    printf ("%f\n", ans);
    printf ("\n");
    return (0);
}
```

There are many more operators but this practice ends now. The practice is sufficient to solve all exercises contained in the assessment exercises set.

9 SUGGESTIONS FOR IMPROVEMENTS

Please report mistakes you notice in this document to vmm@iitg.ernet.in. Further, we welcome your comments and suggestions to improve the document as a training instrument for our students.