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

1 Grouping Statements and Selecting Alternate Execution Actions

1.1 Introduction: Module 02 Stage 02

Students must refer to Module 02 Stage 01 for the protocol details related to the CS110 Lab sessions and assessment practices. The information is not being repeated here to save space on this document.

1.2 Learning aims of Module 02 Stage 02

- Grouping C declarations and statements in blocks;
- Execution flow control through category-specifying decision value ;
- Execution flow control using expressions with Boolean values.
- Boolean expressions.
    - Operators calculating binary results
    - Logical operations on binary values
- Minimal evaluation of logical expressions to determine expression values.
- Programming paradigm: Divide and Conquer.
- `Assert()` declarations as a program documentation and debugging apparatus.

1.3 Assessment Completion Checklist

The following items will be checked by the assessing tutor before recording successful completion of the stage by the student on CS110 course records.

1. Is the program appropriately commented? And, do the comments 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 all constants included in the program code from the exercise statement? The program output should only be computed from the program inputs and constants listed in the problem description.
5. Are the identifiers used as variables helpful and describe the variable use correctly?
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?
9. Student's code should have at least one `assert()` declaration. The student should demonstrate two tests. One test should fail the `assert()` condition and other clear the condition.

2 Learning Guide and Drill Instructions

A simple problem is described to provide a single exercise for all parts of this drill. We have two dates – each date is expressed as three integers to represent the day of the month (dd), the month (mm) and the year (yyyy). To aid discussions, we use the following six variables to represent the dates: `from_dd`, `from_mm`, `from_yyyy`, `to_dd`, `to_mm`, and `to_yyyy`. The assumed purpose of the program is to compute the number of days between the "*from date*" to the "*to date*". The drill will only cover some parts of this program; leaving the other parts as assessment exercises in the assessment set for this stage.

For convenience of writing the document, I will refer to two sets of three variables as date FROM and date TO (I will try to maintain the consistent order where word date appears first.).

We add a small additional requirement to the problem specification. The program must ensure that the two dates are valid. A bad date must terminate the program as it cannot compute the number of days correctly. To keep the number of days manageable we restrict the years to the range from 1700 to 2100. Roughly, the number of computed days will be less than 150000 days (about 366*401 days).

2.1 Programming Paradigm: Divide and Conquer

One foundational paradigm (way) of designing and developing programs is to divide the overall problem into smaller sub-problems. The problem above can be seen as made of four simpler problems:

```
1. Determine if date FROM related variables, as a combine,
   have a valid date value. Report problem and skip other
   tasks, if FROM is an invalid date.
2. Determine if date TO related variables, as a combine,
   have a valid date value. Report problem and skip other
   tasks, if TO is an invalid date.
3. Determine if date TO is after date FROM. This was not
   stated above, but is an implicit requirement in the
   specified problem.
4. If there was no problem in the date values then compute
   the days. And, print the answer.
```

2.2 Statement Blocks in C programs

A large book is always made of chapters. Chapters are made of sections with specific themes. Sections have subsections and so on. The arrangement helps us organise the

book in a way that matches our ability to understand the contents of the book. A book without an organisation would be impossible to read.

Programs have requirements that can easily exceed our abilities to comprehend all their details. Blocks are the artefacts that let us split the larger problem into units that we can understand. Blocks can be embedded inside the bigger blocks to multiple level of depth to match our abilities and needs to organise the problem solution as a combination of understandable parts.

The analogous unit to a chapter in a book is perhaps a function or procedure in the programming domain. We will learn about them later in our curriculum.

The programming paradigm Divide-and-conquer can be applied together with the features of programming language C to construct programs that are comprehensive to human intellect as well as have visible structures supporting easy understanding. Visual identification and recognition of the building blocks of the programs is important to understand the programs.

Details can be localised in the blocks – details that are important within a block may be a distraction outside the block. We may wish to avoid these details once we have understood the contents of a block. Only the external properties of the block are relevant to the program components outside the block.

The construction of a block in C is easy. Each block begins with an opening brace (`{`) and ends with a closing brace (`}`). To make the location and identification of these braces and the code contained in them obvious, certain C style conventions are in common use. The style convention we use for CS101 and CS110 is called Kernighan and Ritchie (K&R) style as it is the coding style used in their book.

If you ignored to follow the style while typing your program, do not be concerned. There is a Linux command `indent` that knows the convention and will make your program visually sanitised quickly. Thus, if your program is `myprog.c` the program is visually cleaned by running the following command:

```
indent -kr myprog.c
```

Many program editors are also likely to understand the C style conventions, and aid you in keeping your program visibly organised. Nonetheless, it is important for you to know the main features of K&R style.

1. Depending on the nested depth of the block, align all declarations, statements and comments to a fixed left margin.
2. Right shift the margin by a small fixed number of spaces as you begin a new nested block inside an outer block. Choose the amount of right-shaft carefully.
3. Place the closing brace of a block at the aligning left margin of the outer block. Let this brace be the single symbol on the line.

4. Since each level of block nesting requires the code inside to be shifted to the right, introduce the blocks sparingly. Deeply nested blocks will leave too much unused white-space on the left of your code.

It is perhaps best to adapt the style of the textbook that you are studying. Consistent use of a style is more important than the specific style version that you use.

Normally start of a big block is a good place to write comment(s) to explain the purpose of the block and/or its features that are important for understanding the code in the block.

Summary: Use blocks to group the code (declarations, statements and sub-blocks) that constitute a coherent unit to improve reading and understanding of the programs. A consistent coding style will ease the understanding of the program. An easy to understand program is less likely to suffer from the programming errors. It will also be easier to find the causes of the errors and correct your programs if the programs have identifiable building blocks with each block focused on one issue.

3 TESTING THE VALIDITY OF DATE **FROM**

Using our previous programming experiences we can quickly arrive at this structure of a yet-to-complete program for validating date FROM:

```
#include <stdio.h>

int main(void)
{
    int from_dd = 20;
    int from_mm = 11;
    int from_year = 2017;
    int invalid_from_date = 0;

    {
       /* Determine if date FROM related
          variables have valid values. Report
          problem and skip the other tasks,
          if a problem is found. */
    }
    return (0);
}
```

It is a standard practice in C to use an `int` value/variable to represent true-false, yes-no (and other binary values). Value of 0 denotes *false* or *no*. Non-zero integer values denote *true* or *yes*.

In the program block we are about to develop, we make an initial default assumption that the set of variables representing date `FROM` have a valid combination of values. This is reflected by initial value of variable `invalid_from_date` – a value 0 means that the date is not invalid. The code will test the validity of the date

components under different criteria and anytime a problem is noticed, the code will add 1 to variable `invalid_from_date`. Thus, recording the invalidity of the date.

`If`-statements in C provide a convenient way for checking the validity of a date. Let us test the month value first. As we know it should be a number from 1 to 12.

```
if (from_mm < 1)
      invalid_from_date = invalid_from_date + 1;
if (from_mm > 12)
      invalid_from_date = invalid_from_date + 1;
```

Let us arbitrarily impose a restriction on the year. We only wish to consider years from 1700 to 2100 as the useful years – outside the range are suspect years!

```
if (from_yyyy < 1700)
      invalid_from_date = invalid_from_date + 1;
if (from_yyyy > 2100)
      invalid_from_date = invalid_from_date + 1;
```

The test strategy can only work for the day of the month partially!

```
if (from_dd < 1)
      invalid_from_date = invalid_from_date + 1;
if (from_dd > 31)
      invalid_from_date = invalid_from_date + 1;
```

But, how about 31 April 2017? It is not a valid date! Even more difficult issue we face is to account for the leap years. 29 February 2016 is a valid date; but, 29 February 2011 is not a valid date.

Before we consider these, you may wish to key-in the following program and run it to see how simple `if-` and `if-else` flow-control constructs regulate the flow of execution paths in the programs.

3.1.1 PROGRAM 1

```
#include <stdio.h>

int main(void)
{
  int from_dd;
  int from_mm;
  int from_yyyy;
  int invalid_from_date = 0;

  printf("Please specify date FROM as yyyy-mm-dd ");
  scanf("%d-%d-%d", &from_yyyy, &from_mm, &from_dd);

  {
    /* Determine if date FROM related
```

```
     * variables have valid values. Report
     * problem and skip other tasks,
     * if a problem is found. */

    /* Check month validity */
    if (from_mm < 1)
      invalid_from_date = invalid_from_date + 1;
    if (from_mm > 12)
      invalid_from_date = invalid_from_date + 1;

    /* Check year validity */
    if (from_yyyy < 1700)
      invalid_from_date = invalid_from_date + 1;
    if (from_yyyy > 2100)
      invalid_from_date = invalid_from_date + 1;

    /* Partially check day validity */
    if (from_dd < 1)
      invalid_from_date = invalid_from_date + 1;
    if (from_dd > 31)
      invalid_from_date = invalid_from_date + 1;
  }

  if (invalid_from_date == 0) {
    printf("Date FROM %d-%d-%d is a valid date\n",
           from_dd, from_mm, from_yyyy);
    return (1);
  } else {
    printf("Date FROM %d-%d-%d is an invalid date\n",
           from_dd, from_mm, from_yyyy);
    return (0);
  }
}
```

The drill will discuss this code again later. The code needs many improvements. We delay those improvements for the moment and focus on the validity of `from_dd` value.

Also note that we are returning value 1 from function `main ()` when things were not right! Return of `0` for `main ()` is to say that all was well. Return of a non-zero value is a way to indicate trouble.

4 SWITCH STATEMENT

Using the two versions of `if-` statements, one can develop very sophisticated decision processes. However, sometimes the choice of activities may be based on a category -- a smart GPS device may determine the path to suggest based on if you are walking, riding a pedal-bike, a motorcycle, a private car, or a public transport

bus. This decision process is based on a category-specifying value stored in a decision variable.

A category-specifying value set is a small set of `int` values each of which corresponds to a category. For example, a month in a year denote a group of dates in that month – thus each year is made of 12 sets of dates. Validating `from_dd` is simply checking it against the valid value set associated with the categorising month.

It is convenient to use a `switch` statement to branch to the right segment of code to process the issues specific to that category.

4.1.1 PROGRAM 2

```
#include <stdio.h>

/* Days in the month */

int main(void)
{
    int month = 5;
    int year = 2017;
    int days_in_month;

    switch (month)
    {
    case 4:
    case 6:
    case 9:
    case 11:
        days_in_month = 30;
        break;
    case 2: /* To be corrected */
        days_in_month = 28;
        break;
    default:
        days_in_month = 31;
    }

    printf ("There are %d days in month %d of year %d\n",
            days_in_month, month, year);
    return (0);
}
```

You obviously note that the days in February are not yet correctly determined in the above program. Apart from that the things to note are:

1. The `switch` statement has a clearly marked body block enclosed in a pair of braces.
2. A number of cases with the same processing have been grouped together.
3. The program control jumps to the case category that matches the value of decision variable – `month` in the example above. And then continues

following the statements sequentially. This is why we could group similar cases together.

4. There is special imperative "`break;`" to indicate that no further processing inside the switch block is needed. Get out to the statement after the `switch` statement; to the statement outside the `switch` body block.
5. Cases not listed can be covered through `default` option. This default is permitted where the last case would have occurred. The default case is not mandatory in a `switch` statement.

5 Binary-valued Expressions – Boolean Expressions

The control expression in an `if`–statement is enclosed in a parentheses-pair and has two useful values. Conventionally we call them true and false. A non-zero value of the control expression is treated as true. The expression evaluating to 0 is false and it causes the else statement to be executed. True outcome causes the first (or the only) statement to be run.

`Else` part in an `if`–statement is optional.

Both these parts are single statements. However, we can easily replace any single statement in a C program by a block of statements. This change requires using a brace-pair (`{ }`) to define a block where the statement being replaced is located.

5.1 Relational Operators

C provides operators to create relational expressions. These operators compare two values and evaluate to a result value 1 (true) or 0 (false).

Following are relational operators in C: equal (==), unequal (`!=`), less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=).

Precedence of == and `!=` is lower than the other relational operators. The equality operators will be applied to the results of the other relational operators, if expression has these operators listed together.

CAUTION: It is a common mistake in C to use = (assignment) operator where the programmer intended to use relation operator == (equal to).

5.2 Boolean Operators

Two binary logical (also called Boolean) operators in C are: `&&` (AND) and `||` (OR). Boolean operation AND is applied first if the Boolean expression has both these logical operators. These logical operators have precedence lower than the precedence of the relational operators – logical operators will be evaluated after the relational operations have been evaluated.

Unary Boolean operator `!` (NOT) is also available and it has precedence even higher than those of the relational operators.

It is often a wise idea to use parentheses in the expressions to make the intensions clearer. However, sometimes too many parentheses make the expression too difficult to read.

CAUTION: There is a separate set of bitwise operators: `&`, `|`, ^, <<, >>, and ~. These operations are not logical operations. Specifically, do not confuse bitwise operator `&` for logical operator `&&`. Similarly, bitwise operator `|` is not same as logical operator `||`. Operator ~ is a bitwise not operation; Operator `!` is a logical negation operation.

5.3 Leap Year Problem

A year divisible by 400 is a leap year. *Of the remaining years*, a year divisible by 100 is not a leap year. *Again of the remaining years*, the one divisible by 4 is a leap year. All other years are non-leap years.

Thus, we can add the following code in `case 2:` of the `switch` statement in PROGRAM 2 to determine value for `days_in_month`.

```
case 2:
        if (year%400 == 0)      /* Year divisible by 400 */
            days_in_month = 29;
        else if (year%100 == 0) /* Of the remaining years */
            days_in_month = 28;
        else if (year%4 == 0) /* Test the non-century years */
            days_in_month = 29;
        else days_in_month = 28;
        break;
```

An alternate way to write the same with fewer `if`-statements is:

```
case 2:
    days_in_month = 28;
    /* Note the use of logical AND && and logical OR || */
    if ((year%400 == 0) || (year%100 != 0) && (year%4 == 0))
        days_in_month = 29;
   break;
```

5.4 Minimal Evaluation of Boolean Expressions

C compilers are smart and they evaluate the Boolean expressions containing logical operators `!` (NOT), `&&` (AND) and `||` (OR) from left to right just enough to know the truth value of the expressions.

Thus, for the code above, program will not evaluate `(year%100 == 0) && (year%4 == 0)` if variable year is 2000. Likewise, if year is 1900, it will not test (year%4 == 0).

5.5 A CONCISE PROGRAM TO VALIDATE **FROM** DATE

Here is a better version of PROGRAM 1 that we developed previously to validate a date information keyed-in by a user. The earlier program was verbose and also not doing all the checks. PROGRAM 3 below does all the checks. However, the new program is difficult to understand for a student programmer. We will describe some techniques to avoid mistakes in the next section.

The students should read the commentary given after PROGRAM 3 to understand the program. Reading – explaining your program code line by line to a friend – is among the most powerful ways to detect and remove errors from your programs. These techniques are often described under the titles *code inspection* or *walkthrough* in the software engineering books.

5.5.1 PROGRAM 3

```
#include <stdio.h>

int main(void)
{
    int from_dd;
    int from_mm;
    int from_yyyy;
    int invalid_from_date = 0;

    printf("Please specify date FROM as yyyy-mm-dd ");
    scanf("%d-%d-%d", &from_yyyy, &from_mm, &from_dd);

    {
        /* Determine if the "from" date related
           variables have valid values. */

        /* Check validity */
        if (from_mm < 1 || from_mm > 12 ||
                from_yyyy < 1700 || from_yyyy > 2100 ||
                from_dd < 1 || from_dd > 31)
            /* Failed to validate on common checks */
            invalid_from_date = 1;
        else if ((from_mm == 4 || from_mm == 6 ||
                  from_mm == 9 || from_mm == 11) &&
                 from_dd >30)
            /* Failed to validate date for 30 days months */
            invalid_from_date = 1;
        else if (from_mm == 2 && from_dd >29)
            /* Failed to validate maximum date check in Feb */
            invalid_from_date = 1;
        else if (from_dd == 29 &&
                 !((from_yyyy%400 == 0) ||
                   (from_yyyy%100 != 0)&& (from_yyyy%4 == 0)))
            /* Non-leap year Feb with 29 days */
            invalid_from_date = 1;
    }
```

```
    if (invalid_from_date == 0) {
        printf("Date FROM %d-%d-%d is a valid date\n",
               from_dd, from_mm, from_yyyy);
        return (0);
    } else {
        printf("Date FROM %d-%d-%d is an invalid date\n",
               from_dd, from_mm, from_yyyy);
        return (1);
    }
}
```

We now use only two values for variable `invalid_from_date`.

The validity of date `FROM` is checked in PROGRAM 3 through a series of checks. `FROM` values that are already known to be invalid are not tested any further! This is achieved by placing each new check in `else` part of the cascading `if-else` statements.

Thus, we are making tests related to months of February, April, June, September, and November days only when the years and month values are both valid. On the other hand, the tests on `from_dd` value are made with clear knowledge that the value is within the range 1 to 31 (both values inclusive).

Likewise, when leap year related check is made, we know that `from_mm` value is 2 and `from_dd` has value in the range 1 to 29. Thus, we only need to be sure that if `from_dd` is 29 then the year is a leap year. Once this check can be made, date FROM is valid.

6 Error Avoidance Through `assert()` Declarations

We have learned that some computations require careful control to ensure that the actions are executed only under the right conditions. We have learned three flow-control constructs (`if-`, `if-else`, and `switch`) that C programs use. If the actions require more than one statement, the statements can be grouped as a block.

Keeping track of the situation under which an action is executed is difficult and error-prone. For example, PROGRAM 3 in the previous section is difficult to understand because it has a number of `if` statements. Further, the statements are nested and/or cascaded to make the understanding error-prone.

Programming errors can be reduced by including `assert()` declarations in the programs. These are advisory and non-essential declarations. However, if included in the program, they are tested for their validity. If an assertion is not valid during the execution of the program, the program terminates abruptly after reporting the cause and location of the failed assertion. The programmer can then review the program and assertion to remove the error.

The program for validating date FROM variables is reproduced below. This time several assertions have been added to support our claim that only the invalid dates are being tagged as bad dates.

Note that the program now includes, a special library specified by statement `#include <assert.h>` at the start of the program. There are 4 sets of asserts added to the program to support our beliefs at different points in the program. These claims are explained after the program code below.

6.1.1 PROGRAM 4

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

int main(void)
{
   int from_dd;
   int from_mm;
   int from_yyyy;
   int invalid_from_date = 0;

   printf("Please specify date FROM as yyyy-mm-dd ");
   scanf("%d-%d-%d", &from_yyyy, &from_mm, &from_dd);

   {
      /* Determine if date FROM related
       * variables have valid values. */

      /* Check validity */
      if (from_mm < 1 || from_mm > 12 ||
          from_yyyy < 1700 || from_yyyy > 2100 ||
          from_dd < 1 || from_dd > 31)
         /* Failed to validate common checks */
      {
         /* See Explaination Item 1 */
         invalid_from_date = 1;
      } else
          if ((from_mm == 4 || from_mm == 6 || from_mm == 9
               || from_mm == 11) && from_dd > 30)
         /* Failed to validate dates for 30 days months */
      {
         /* See Explaination Item 2 */
         assert(from_mm != 1 && from_mm != 3 && from_mm != 5
                && from_mm != 7 && from_mm != 8
                && from_mm != 10 && from_mm != 12);
         assert(from_dd == 31);

         invalid_from_date = 1;
      } else if (from_mm == 2 && from_dd > 29)
         /* Failed to validatre gross day check for Feb */
      {
         /* See Explaination Item 3 */
```

```
            assert(from_mm == 2);;
            assert(from_dd == 30 || from_dd == 31);
            invalid_from_date = 1;
         } else if (from_dd == 29 &&
                   !((from_yyyy % 400 == 0) ||
                     (from_yyyy % 100 != 0)
                     && (from_yyyy % 4 == 0)))
            /* Non-leap year Feb with 29 days */
         {
            /* See Explaination Item 4 */
            assert(from_dd == 29 && from_mm == 2);
            assert(from_yyyy % 4 != 0 || from_yyyy % 100 == 0);
            assert(from_yyyy % 100 == 0
                   && from_yyyy % 400 != 0);

            invalid_from_date = 1;
         }
      }

      if (invalid_from_date == 0) {
         printf("Date FROM %d-%d-%d is a valid date\n",
                from_dd, from_mm, from_yyyy);
         return (0);
      } else {
         printf("Date FROM %d-%d-%d is an invalid date\n",
                from_dd, from_mm, from_yyyy);
         return (1);
      }
   }
```

As you read the following explanations, you will notice that there was a common principal mission guiding us. The principal mission in this example was not-to-mark a date invalid unless there is a clear reason to do so.

However, there is a separate mission of equal importance. This is to check every date for validity. That is, by using assert declarations as a second line of defense we have made sure that no valid date will be incorrectly marked as invalid. However, we have not created a second line of defense to not report an invalid date as valid.

This is a small and simple program. The checks can be made easily by some careful human efforts. In a more complex programs making such checks can become very difficult.

Let us now come back to PROGRAM 4 and explain how assert declarations help us in avoiding some errors. The headings in the following explanations match the phrases included in the comments of the program. Please read these explanations with the related program codes.

6.1.2 Explanation Item 1:

From the Boolean expression in the `if`-clause it is quite obvious that the `invalid_from_date` is set if any of the variable defining the date has value outside its most permissive range.

6.1.3 Explanation Item 2:

This being the `else` part of the first `if`-statement, we are assured that all the obvious checks on the variable values were valid. Specifically, `from_mm` has a value between 1 and 12. Boolean expression in the second `if`-statement has tested that `from_mm` is a month with 30 days but the `from_dd` is 31. So `FROM` is an invalid date. The `assert()` declarations we use captures the situation. The declarations say that the month is *not* a month with 31 days but `from_dd` is 31. Obviously a wrong combination of date values.

The program will report error if any of the listed `assert()` declarations fails. The message on the screen will give the line number of the failing assert declarations and the Boolean expression that was violated. Run the program by giving an input date to cause an error. For example, give input date of 2000-9-31.

6.1.4 Explanation Item 3:

This is testing if a February `from_dd` value is more than 29. Various `assert()` declarations take note of the fact that `from_dd` was already known to be a value between 1 and 31.

6.1.5 Explanation Item 4:

We will arrive at this location in the program only if all the `if`-conditions had returned false. This is only possible if `from_mm` is 2 and `from_dd` is 29. The date is invalid if `from_yyyy` is not a leap year. The condition in the `if`–statement and asserts within are written to match this conviction.

The assertions are not mandatory part of a C program but can be added to provide guidance to the reader of the program. Also since they are executed and tested in the generated code, assert () declarations can help eliminate obvious errors in the programs.

Validity check for the three values defining date `TO` are similar.

6.2 ADDING A SECOND LINE OF DEFENSE AGAINST ACCEPTING INVALID DATES

We expressed a small annoyance in the previous section that the assert declarations used there were not giving us a second line of defense against inadvertently failing to reject an invalid date.

Our next program is a variant of the previous program (PROGRAM 4). However, it uses the assert declarations to be sure that every accepted valid date meets our criteria set for a good date!

6.2.1 PROGRAM 5

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

int main(void)
{
   int from_dd;
   int from_mm;
   int from_yyyy;
   /* Left uninitialised to get coompiler caution */
   int invalid_from_date;

   printf("Please specify date FROM as yyyy-mm-dd ");
   scanf("%d-%d-%d", &from_yyyy, &from_mm, &from_dd);

   {
      /* Determine if the "from" date related
       *            variables have valid values. */

      /* Check date validity */
      if (from_mm < 1 || from_mm > 12 ||
          from_yyyy < 1700 || from_yyyy > 2100 ||
          from_dd < 1 || from_dd > 31)
         /* Failed to validate common checks */
         invalid_from_date = 1;
      else if ((from_mm == 4 || from_mm == 6 || from_mm == 9
                || from_mm == 11) && from_dd > 30)
         /* Failed to validate dates for 30 days months */
         invalid_from_date = 1;
      else if (from_mm == 2 && from_dd > 29)
         /* Failed to validatre gross day check for Feb */
         invalid_from_date = 1;
      else if (from_dd == 29 &&
               !((from_yyyy % 400 == 0) ||
                 (from_yyyy % 100 != 0)
                 && (from_yyyy % 4 == 0)))
         /* Non-leap year Feb with 29 days */
         invalid_from_date = 1;
      else {
         /* At this point we believe that we have taken
          * care of all cases that make a date invalid.
          *
          * We now make assertion declarations of our
          * success. If we are wrong program run will
          * fail on us.
          *
          */

         /* Year is with the range */
         assert(1700 <= from_yyyy && from_yyyy <= 2100);
         /* Month is with in the range */
```

```
632           assert(1 <= from_mm && from_mm <= 12);
633           /* Day of the month is with its gross range */
634           assert(1 <= from_dd && from_dd <= 31);
635           /* if date above 30 then it is not a shorter month */
636           assert(from_dd < 31
637                  || !(from_mm == 2 || from_mm == 4
638                       || from_mm == 6 || from_mm == 9
639                       || from_mm == 11));
640           /* February has even tighter condition on from_dd */
641           assert(from_dd < 30 || from_mm != 2);
642           /* Outside leap year February is even shorter */
643           assert(from_dd < 29 || from_mm != 2
644                  || from_yyyy % 400 == 0
645                  || (from_yyyy % 100 != 0
646                      && from_yyyy % 4 == 0));
647           /* NOw we can mark the date as valid */
648           invalid_from_date == 0;
649        }
650     }
651
652     if (invalid_from_date == 0){
653        printf("Date FROM %d-%d-%d is a valid date\n",
654               from_dd, from_mm, from_yyyy);
655        return (0);
656     } else {
657        printf("Date FROM %d-%d-%d is an invalid date\n",
658               from_dd, from_mm, from_yyyy);
659        return (1);
660     }
661  }
```

662 6.3 USING ASSERT DECLARATIONS TO EXCLUDE INVALID DATES

663 With the `assert()` as a powerful tool in our hand, the program to validate date
664 `FROM` can be simplified considerably. We only need to use a right set of asserts to
665 exclude every possible bad date. PROGRAM 6 below makes all checks through
666 `assert ()` declarations. The program needs no `if`-statement to test the date for
667 validity.

668 6.3.1 PROGRAM 6

```
669  #include <stdio.h>
670  #include <assert.h>
671
672  int main(void)
673  {
674     int from_dd;
675     int from_mm;
676     int from_yyyy;
677
678     printf("Please specify date FROM as yyyy-mm-dd ");
679     scanf("%d-%d-%d", &from_yyyy, &from_mm, &from_dd);
```

```
    assert(from_mm >= 1 && from_mm <= 12);
    assert(from_yyyy >= 1700 && from_yyyy <= 2100);
    assert(from_dd >= 1 || from_dd <= 31);
    /* Either from_dd < 31 or month is of 31 days */
    assert(from_dd <= 30 ||
           from_mm == 1 || from_mm == 3 || from_mm == 5 ||
           from_mm == 7 || from_mm == 8 || from_mm == 10 ||
           from_mm == 12);
    /* Either from_dd is less than 30 or it is not February */
    assert(from_dd <= 29 || from_mm != 2);
    /* Either it is Not February or
     * it is February with from_dd < 29
     * or from_dd is 29 in a leap year */
    assert(from_mm != 2 ||
           (from_mm == 2 && from_dd < 29) ||
           (from_dd == 29 &&
            /* Leap year check */
            (from_yyyy % 400 == 0 ||
             from_yyyy % 100 != 0 && from_yyyy % 4 == 0)));

    printf("Date FROM %d-%d-%d is a valid date\n",
           from_dd, from_mm, from_yyyy);
    return (0);
}
```

Alert students would note that the sets of `assert ()` declarations in PROGRAM 5 and PROGRAM 6 are the same accept for inconsequential differences in the way the Boolean expressions are written!

7 CHECKING THAT DATE FROM IS BEFORE DATE TO

The final check for ensuring that date given as TO is not before date given as FROM is easy.

7.1.1 PROGRAM 7

```
#include <stdio.h>

int main(void)
{
    int from_dd, to_dd;
    int from_mm, to_mm;
    int from_yyyy, to_yyyy;
    int invalid_from_date = 0;
    int invalid_to_date = 0;
    int from_before_to;

    printf("Please specify date FROM as yyyy-mm-dd ");
    scanf("%d-%d-%d", &from_yyyy, &from_mm, &from_dd);

```

```
    printf("Please specify date TO as yyyy-mm-dd ");
    scanf("%d-%d-%d", &to_yyyy, &to_mm, &to_dd);

    {
        /* Here include code to check the validity of
           three variables denoting from_date

           Code removed to save space */
    }

    {
        /* Here include code to check the validity of
           three variables denoting from_date

           Code removed to save space */
    }

    {
        /* Set from_before_to */
        if (from_yyyy < to_yyyy)
            from_before_to = 1;
        else if (from_yyyy == to_yyyy && from_mm < to_mm)
            from_before_to = 1;
        else if (from_yyyy == to_yyyy &&
                 from_mm == to_mm && from_dd <= to_dd)
            from_before_to = 1;
        else from_before_to = 0;
    }

    if (!invalid_from_date &&
            !invalid_to_date && from_before_to) {
        printf ("Date FROM %d-%d-%d is before or \
                same as date TO %d-%d-%d\n",
                from_dd, from_mm, from_yyyy,
                to_dd, to_mm, to_yyyy);
        return (0);
    } else {
        printf ("Invalid specification of dates \n");
        return (1);
    }
}
```

In the format string of `printf ()` in the last if-statement you notice a `\` added by my use of `indent` application. This is the standard way a string is split across multiple lines.

8 CONCLUDING REMARKS

In this drill you learned two important lessons. The first lesson is that when faced with a big programming problem, divide it into a set of smaller programming problems such that combination of their solutions is a solution to the main problem. The other lesson we learn is that a problem can have many alternate solutions. Programming is an art of selecting an elegant solution among them. This requires perseverance and practice with programs.

Code for computing the days is included as an assessment exercise. To compute the number of days, you will need to determine the exact number of leap days (29 February) that intervene the two dates.

Make sure that every program marked by Heading PROGRAM in this drill was run by you during the training and you made alterations to the code and ran multiple test cases to see the effects of your actions on the program execution.

Now is the time to ask for formal assessment of your drill.

9 ERROR REPORTING AND SUGGESTIONS FOR IMPROVEMENTS

My sincere apologies if the document has errors or mistakes. Please report errors in this document to vmm@iitg.ernet.in. Also, I welcome suggestions and advice to improve the quality of the document for the students of CS110.