

2022-09-06 LECTURE 07 – PROGRAM CONTROL – II

- contents
 - constants and literals
 - data types and their ranges
 - format specifiers
 - escape sequences
 - operators and their precedence
 - structured programming
 - nested control structures
 - while in while
 - for in for
 - concepts discussed in the lecture
 - exercises
- literals and constants
 - **literal** refer to fixed values that the program may not alter during its execution or a value that is expressed as itself
 - for example, the number 25 or the string "Hello World" are both literals
 - an integer is a numeric literal (associated with numbers) without any fractional or exponential part, there are three type of integer literals
 - **decimal** (number in base 10)
 - 0, -9, 22
 - **octal** (number in base 8)
 - 021 // this is equal to $2 * 8^1 + 1 * 8^0 = 16 + 1 = 17$ decimal
 - 077 // this is equal to $7 * 8^1 + 7 * 8^0 = 56 + 7 = 63$ decimal
 - 033 // this is equal to $3 * 8^1 + 3 * 8^0 = 24 + 3 = 27$ decimal
 - **hexadecimal** (number in base 16)
 - 0x7f // this is equal to $7 * 16^1 + 15 * 8^0 = 116 + 15 = 131$ decimal
 - 0x2a // this is equal to $2 * 16^1 + 10 * 8^0 = 32 + 10 = 42$ decimal
 - 0x521 // this is equal to $5 * 16^2 + 2 * 16^1 + 1 * 16^0 = 1280 + 32 + 1 = 1313$ decimal
 - a **floating-point literal** is a **numeric literal** that has either a fractional form or an exponent form, floating point literals are expressed as follows
 - -2.0
 - 0.0000234
 - -0.22E-5 // this means -0.22×10^{-5}
 - a **character literal** is created by enclosing a single character inside single quotation marks, character literals are expressed as follows
 - 'a', 'm', 'F', '2', '}'
 - a **string literal** is a sequence of characters enclosed in double-quote marks, string literals are expressed as follows
 - "good" //string constant
 - "" //null string constant

- " " //string constant of six white space
 - "x" //string constant having a single character.
 - "Earth is round\n" //prints string with a newline
- **constant**
 - to define a variable whose value cannot be changed during the execution of a program, it can be defined using the **const keyword** which creates a constant
 - const int a = 10;
 - const float PI = 3.14;
- **data types**
 - in C programming, data types are used for declarations of variables
 - it determines the type and size of data associated with variables
 - commonly used types in C programming include
 - **int**: integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 0, -5, 10
 - the size of int is usually 4 bytes (32 bits), and, it can take 2^{32} distinct states, it can hold the values in the range from -2147483648 to 2147483647
 - **float, double and long double**: float, double and long double are used to hold real numbers
 - the size of **float** is usually 4 bytes (32 bits), and, it can hold the values in the range from 3.4E-38 to 3.4E+38
 - the size of **double** is usually 8 bytes (64 bits), and, it can hold the values in the range from 1.7E-308 to 1.7E+308
 - the size of **long double** is usually 10 to 16 bytes (80 to 128 bits), and, it can hold the values in the range from 3.4E-4932 to 1.1E+4932
 - **short int, long int, and long long int**:
 - if you want to use, only a small integer you should use short, and, it can hold the values in the **range** from -32,767 to +32,767
 - if you want to use, only a long integer you should use long, and, it can hold the values in the range from -2,147,483,648 to 2,147,483,647
 - if you want to use, very long integer then you should use **long long int**, and, it can hold the values in the range from $-(2^{63})$ to $(2^{63})-1$
 - **signed and unsigned int**: in C, signed and unsigned are type modifiers, you can alter the data storage of a data type by using them:
 - **signed** - allows for storage of both positive and negative numbers
 - **unsigned** - allows for storage of only positive numbers
 - **unsigned short int**: it can hold the values in the range from 0 to 65535
 - **unsigned long int**: it can hold the values in the range from 0 to +4,294,967,295
 - **unsigned long long int**: it can hold the values in the range from 0 to 18,446,744,073,709,551,615
 - **char**: keyword char is used for declaring character type variables.
 - **signed char**:
 - it can hold the values in the range from -128 to 127
 - **unsigned char**:
 - it can hold the values in the range from 0 to 255
 - **example (using sizeof statements)**: a program that uses **sizeof** statements

// L07-C01

```

1. #include <stdio.h>
2. int main() {
3.     int a;
4.     char b;
5.     float c;
6.     short int d;
7.     unsigned int e;
8.     long int f;
9.     long long int g;
10.    unsigned long int h;
11.    unsigned long long int i;
12.    signed char j;
13.    unsigned char k;
14.    long double l;
15.    printf("size of  int           = %d bytes\n", sizeof(a));
16.    printf("size of  char          = %d bytes\n", sizeof(b));
17.    printf("size of  float         = %d bytes\n", sizeof(c));
18.    printf("size of  short int      = %d bytes\n", sizeof(d));
19.    printf("size of  unsigned int   = %d bytes\n", sizeof(e));
20.    printf("size of  long int        = %d bytes\n", sizeof(f));
21.    printf("size of  long long int    = %d bytes\n", sizeof(g));
22.    printf("size of  unsigned long int = %d bytes\n", sizeof(h));
23.    printf("size of  unsigned long long int = %d bytes\n", sizeof(i));
24.    printf("size of  signed char       = %d bytes\n", sizeof(j));
25.    printf("size of  unsigned char      = %d bytes\n", sizeof(k));
26.    printf("size of  long double          = %d bytes\n", sizeof(l));
27.    return 0;
28. }
```

▪ Output:

```

▪ size of  int           = 4 bytes
▪ size of  char          = 1 bytes
▪ size of  float         = 4 bytes
▪ size of  short int      = 2 bytes
▪ size of  unsigned int   = 4 bytes
▪ size of  long int        = 4 bytes
▪ size of  long long int    = 8 bytes
▪ size of  unsigned long int = 4 bytes
▪ size of  unsigned long long int = 8 bytes
▪ size of  signed char       = 1 bytes
▪ size of  unsigned char      = 1 bytes
▪ size of  double          = 8 bytes
▪ size of  long double          = 16 bytes
```

• format specifiers

- the format specifiers are used in C for input and output purposes

- using this concept the compiler can understand that what type of data is in a variable during taking input using the scanf() function and printing using printf() function
- following are different format specifier used in the scanf() and printf() functions

Data Type	Size (bytes)	Format Specifier
int	at least 2, usually 4	%d, %i
char	1	%c
float	4	%f
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%ld, %li
long long int	at least 8	%lld, %lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
signed char	1	%c
unsigned char	1	%c
double	8	%lf
long double	at least 10, usually 12 or 16	%Lf

- **example (using different format specifiers):** a program that uses **format specifiers**

// L07-C02

```

1. #include <stdio.h>
2. int main() {
3.     int a = 10;
4.     char b = 'A';
5.     float c = 2.25;
6.     short int d = -5;
7.     unsigned int e = 25;
8.     long int f = 25;
9.     long long int g = -125;
10.    unsigned long int h = 125;
11.    unsigned long long int i = 1250;
12.    signed char j = 'B';
13.    unsigned char k = 'C';
14.    double l = 12.50;
15.    long double m = -25.25;
16.    printf("int           = %d\n", a);
17.    printf("char          = %c\n", b);
18.    printf("float         = %f\n", c);
19.    printf("short int      = %hd\n", d);
20.    printf("unsigned int   = %u\n", e);
21.    printf("long int       = %ld\n", f);
22.    printf("long long int  = %lld\n", g);
23.    printf("unsigned long int = %lu\n", h);
24.    printf("unsigned long long int = %llu\n", i);
25.    printf("signed char    = %c\n", j);
26.    printf("unsigned char   = %c\n", k);
27.    printf("double         = %lf\n", l);

```

```

28. printf("long double          = %Lf\n", m);
29. return 0;
30. }

```

▪ **Output:**

```

▪ int          = 10
▪ char         = A
▪ float        = 2.250000
▪ short int    = -5
▪ unsigned int = 25
▪ long int     = 25
▪ long long int = -125
▪ unsigned long int = 125
▪ unsigned long long int = 1250
▪ signed char  = B
▪ unsigned char = C
▪ double       = 12.500000
▪ long double  = -25.250000

```

• **escape sequences**

- in C, all escape sequences consist of two or more characters, the first of which is the backslash, \ (called the "Escape character"); the remaining characters determine the interpretation of the escape sequence
- for example, \n is an escape sequence that denotes a newline character
- there are several types of escape sequence in C to achieve various purposes
 - \n (New line): used to shift the cursor control to the new line
 - \t (Horizontal tab): used to shift the cursor to a couple of spaces to the right in the same line
 - \a (Audible bell): used to generated a beep indicating the execution of the program to alert the user
 - \r (Carriage Return): used to position the cursor to the beginning of the current line
 - \\ (Backslash): used to display the backslash character
 - \' (Apostrophe or single quotation mark): used to display the single-quotation mark
 - \" (Double quotation mark): used to display the double-quotation mark
 - \0 (Null character): used to represent the termination of the string
 - \? (Question mark): used to display the question mark (?)
 - \nnn (Octal number): used to represent an octal number
 - \xhh (Hexadecimal number): used to represent a hexadecimal number
 - \v (Vertical tab): used to move curser vertically down
 - \b (Backspace): used to move curser on character back
 - \e (Escape character):
 - \f (Form Feed page break): used to eject current pages from the printer

- **operators:** an operator is a symbol that tells the compiler to perform specific mathematical or logical operations in C language

- **arithmetic operators:** assume A = 10 and B = 20

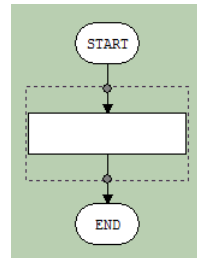
- + adds two operands. $A + B \rightarrow 30$
- - subtracts second operand from the first. $A - B \rightarrow -10$
- * multiplies both operands. $A * B \rightarrow 200$
- / divides numerator by denominator. $B / A \rightarrow 2$
- % remainder of an integer division. $B \% A \rightarrow 0$
- ++ increases the integer value by one. $A++ \rightarrow 11$
- -- decreases the integer value by one. $A-- \rightarrow 9$
- **relational operators:** assume $A = 10$ and $B = 20$
 - == checks if the values of two operands are equal or not. If yes, then the condition becomes true. $(A == B)$ is not true.
 - != checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true. $(A != B)$ is true.
 - > checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true. $(A > B)$ is not true.
 - < checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true. $(A < B)$ is true.
 - >= checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true. $(A >= B)$ is not true.
 - <= checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true. $(A <= B)$ is true.
- **logical operators:** assume $A = 1$ and $B = 0$
 - && called logical AND operator, if both the operands are non-zero, then the condition becomes true. $(A \&\& B)$ is false.
 - || called logical OR operator, if any of the two operands is non-zero, then the condition becomes true. $(A || B)$ is true.
 - ! called logical NOT operator, it is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. $!(A \&\& B)$ is true.
- **bitwise operators:** assume $A = 60$ and $B = 13$
 - & binary AND operator copies a bit to the result if it exists in both operands. $(A \& B) \rightarrow 12$, i.e., 0000 1100
 - | Binary OR Operator copies a bit if it exists in either operand. $(A | B) \rightarrow 61$, i.e., 0011 1101
 - ^ Binary XOR Operator copies the bit if it is set in one operand but not both. $(A \wedge B) \rightarrow 49$, i.e., 0011 0001
 - ~ Binary One's Complement Operator is unary and has the effect of 'flipping' bits. $(\sim A) \rightarrow \sim(60)$, i.e., -0111101
 - << Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand. $A \ll 2 \rightarrow 240$ i.e., 1111 0000
 - >> Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand. $A \gg 2 \rightarrow 15$ i.e., 0000 1111
- **assignment operators**
 - = assigns values from right side operands to left side operand $C = A + B$ will assign the value of $A + B$ to C

- += it adds the right operand to the left operand and assigns the result to the left operand. $C += A$ is equivalent to $C = C + A$
- -= it subtracts the right operand from the left operand and assigns the result to the left operand. $C -= A$ is equivalent to $C = C - A$
- *= it multiplies the right operand with the left operand and assigns the result to the left operand. $C *= A$ is equivalent to $C = C * A$
- /= it divides the left operand with the right operand and assigns the result to the left operand. $C /= A$ is equivalent to $C = C / A$
- %= it takes modulus using two operands and assigns the result to the left operand. $C \% = A$ is equivalent to $C = C \% A$
- <<= left shift AND assignment operator. $C <<= 2$ is same as $C = C << 2$
- >>= right shift AND assignment operator. $C >>= 2$ is same as $C = C >> 2$
- &= bitwise AND assignment operator. $C \&= 2$ is same as $C = C \& 2$
- ^= bitwise exclusive OR and assignment operator. $C \wedge = 2$ is same as $C = C \wedge 2$
- |= bitwise inclusive OR and assignment operator. $C |= 2$ is same as $C = C | 2$
- **miscellaneous operators**
 - sizeof() returns the size of a variable. sizeof(a), where a is integer, will return 4
 - & returns the address of a variable. &a; returns the actual address of the variable
 - * pointer to a variable, *a;
 - ?: conditional expression, if condition is true ? then X : otherwise Y
- **operator precedence**
 - operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated
 - certain operators have higher precedence than others;
 - for example, the multiplication operator has a higher precedence than the addition operator

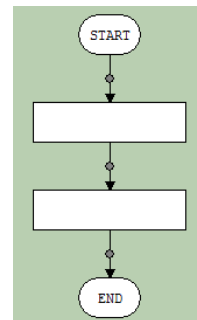
Category	Operator	Associativity
postfix	(), [], ->, ., ++, --	Left to right
unary	+, -, !, ~, ++, --, (type), *, &, sizeof	Right to left
multiplicative	*, /, %	Left to right
additive	+, -	Left to right
shift	<<, >>	Left to right
relational	<, <=, >, >=	Left to right
equality	==, !=	Left to right
bitwise AND	&	Left to right
bitwise XOR	^	Left to right
bitwise OR		Left to right
logical AND	&&	Left to right
logical OR		Left to right
conditional	?:	Right to left
assignment	=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, =	Right to left
comma	,	Left to right

- **structured programming**

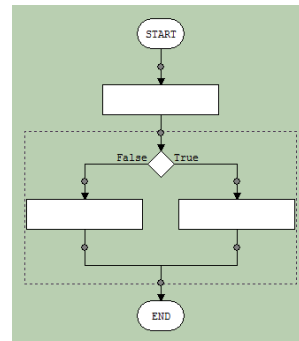
- composing programs as sequences of blocks with a single entry and exit points make them easier to understand
- this aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of
 - sequence
 - selection
 - repetition
- for simplicity above control statements could only be combined in two ways
 - **stacking**: joining control statements in a sequence one after another
 - **nesting**: embedding one control statement into another using single entry and exit principle
- to write a structured program following rules are used
 - **Rule 1**: begin with a simplest flow chart having a start, rectangle (action with single entry and single exit) and end symbols



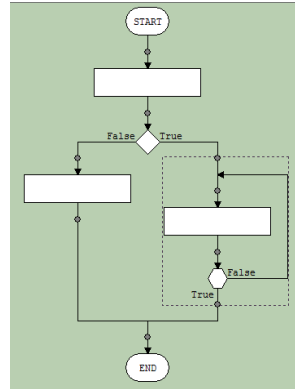
- **Rule 2**: stacking rule: any rectangle (action) can be replaced by two rectangles (actions) in sequence



- **Rule 3**: nesting rule: any rectangle (action) can be replaced by any control statement (sequence, if, if...else, switch, while, do...while or for)



- **Rule 4**: stacking and nesting may be applied as often as you like and in any order



- Rule 4 can help to design larger and more deeply nested structures
- because of the elimination of the goto statement, these building blocks never overlap one another
- in C language selection is implemented using three ways
 - if (single selection)
 - if . . . else (double selection)
 - switch (multiple selection)
 - it is straightforward to prove that simple if statement is sufficient to provide any form of selection
- in C language repetition is implemented using three ways
 - while
 - do . . . while
 - for
 - it is straightforward to prove that simple while statement is sufficient to provide any form of repetition
- hence, any form of control can be expressed in only the following three forms of controls
 - sequence
 - if
 - while
- similarly only following two forms of combining these controls can produce any structured program
 - stacking
 - nesting
- nested control structures
 - recall that a nested control structure means to embed one control structure into another control structure
 - for example, repetition control structure could be used inside another repetition control structure
 - while in while nesting
 - **example (using while nested control statements):** a program that prompts user to enter a number and displays its table at console, if user enters a **0** or a **negative number** than exit the program.

// L07-C03

1. int main (void) {

```

2.  int counter = 0, number;
3.  int sentinel = 1;
4.  while ( sentinel != 0 ) {
5.      printf("enter the a number to display its table (<= 0 to exit):\t");
6.      scanf("%d", &number);
7.      if ( number > 0 ) {
8.          counter = 1;
9.          while ( counter <= 10 ) {
10.             printf("%d\t*\t%d\t=\t%d\n", number, counter, number*counter);
11.             counter = counter + 1;
12.          } // end of inner while
13.      } // end of if
14.      else {
15.          sentinel = 0;
16.      } // end else
17.  } // end of outer while
18. }

```

▪ **output/input:** enter the a number to display its table (<= 0 to exit): 5

```

▪ output: 5 * 1 = 5
▪ output: 5 * 2 = 10
▪ output: 5 * 3 = 15
▪ output: 5 * 4 = 20
▪ output: 5 * 5 = 25
▪ output: 5 * 6 = 30
▪ output: 5 * 7 = 35
▪ output: 5 * 8 = 40
▪ output: 5 * 9 = 45
▪ output: 5 * 10 = 50

```

▪ **output/input:** enter the a number to display its table (<= 0 to exit): 0

- **Line 2:** define and initialize a counter and a variable to get user input named **counter** with value 0, and number to store user's entered value
- **Line 3:** define and initialize a sentinel variable named **sentinel** with value 1
- **Line 4:** defines a condition on the **sentinel** variable which would be true (1) if the value in **sentinel** variable is not equal to 0
- **Line 5&6:** prompt user to enter a number and get input from user and store in variable number
- **Line 7&8:** if the user's entered number is greater than zero then set counter variable equal to 1
- **Line 9-12:** run the statements in the while control from counter's value equal to 1 to counter's value is 10 and print table of the user's entered number at console using printf statement

- **Line 14-16:** if the user's entered number is equal or less than zero then the else block from lines 14-16 will set the value of the variable named **sentinel** equal to zero, which will terminate while condition at line 4
- for in for nesting
 - **example (using for nested control statements):** a program that prompts user to enter a number and displays a square of '*' at console having dimensions equal to users entered number
 - // L07-C04**
 - 1. int main (void) {
 - 2. **int number;**
 - 3. printf("enter a number:\t");
 - 4. scanf("%d", &number);
 - 5. **for (int i=0; i<number; i++) {**
 - 6. **for (int j=0; j<number; j++) {**
 - 7. printf(" * ");
 - 8. }**// end of inner for**
 - 9. printf("\n");
 - 10. }**// end of outer for**
 - 11. }
 - **output/input:** enter a number: 5
 - **output:** * * * * *
 - **output:** * * * * *
 - **output:** * * * * *
 - **output:** * * * * *
 - **output:** * * * * *
 - **Line 2:** define a variable to get user input named **number**
 - **Line 3&4:** prompt the user to enter a number and take user's input and store it into the variable number
 - **Line 5:** outer for repetition is started with for header where a counter variable named "i" is defined and initialized with value zero, a condition on the **variable i** is defined which would be true (1) if the value in "i" variable is less than the value of user's entered value (stored in the variable **number**)
 - **Line 6:** inner for repetition is started with for header where a counter variable named "j" is defined and initialized with value zero, a condition on the **variable j** is defined which would be true (1) if the value in "j" variable is less than the value of user's entered value (stored in the variable **number**)
 - **Line 6:** display * on console and as many times as the value entered by the user
 - **Line 7:** after the termination inner for repetition a new line character is displayed at console and the control is transferred back to header of the outer for repetition at line 5
- concepts discussed in the lecture
 - literal and constants

- decimal, octal, hexadecimal, floating-point literal, numeric literal, character literal, string literal, constant, const keyword,
- datatypes
 - int, float, double, long double, short int, long int, long long int, range, signed, unsigned, unsigned short int, unsigned long int, unsigned long long int, char, signed char, unsigned char,
- formatted I/O
 - sizeof, format specifiers, escape sequence, \n, \t, \a, \r, \\, \', \", \0, \?, \nnn, \xhh, \v, \b, \f
- operators
 - arithmetic operators, relational operators, logical operators, bitwise operators, assignment operators, miscellaneous operators, operator precedence, unary operators, shift operators, bitwise operators,
- structured programming
 - stacking, nesting, goto statement