Programming with C and C++

Meher Krishna Patel

Created on : Octorber, 2017

Last updated : May, 2021

Table of contents

No amount of prayer or meditation can do what helping others can do.

–Meher Baba

Chapter 1

First Program

1.1 Introduction

C language can be used in embedded design with Nios processor and SystemC. This tutorial contains various features of C language, which we will need for designing the SoPC (system on programmable chip) using NiosII processor. Further, the tutorials on 'SoPC design using C/C++' can be found at the website under the Section 'VHDL/Verilog tutorials'. In this chapter, simple 'Hello World' program is discussed along with it's execution. Further, the Table 1.1 shows the list of Keywords which are discussed which are discussed in this tutorial.

Table 1.1: Keywords in C and C++

C and C++			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	unsigned
const	float	short	union
continue	for	sizeof	while
default	goto	signed	void
do	if	static	volatile
Only C++			
asm	FALSE	public	try
bool	friend	protected	typeid
catch	inline	reinterpret_cast	typename
class	mutable	static_cast	using
const_cast	namespace	template	virtual
delete	new	this	wchar_t
dynamic_cast	operator	throw	
explicit	private	TRUE	
Bitwise (only C ++)			
and	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

1.2 Installing Compiler

We only need a C compiler to execute the programs. There are various C/C++ compiler available on Internet. Some of the C/C++ compilers are discussed below, so that we can compile both C and C++ codes.

Linux: To install the compiler on Linux, type following command to the terminal,

```
$ sudo apt-get install g++
```

Windows: Install 'mingw' compiler and add it to windows path; or 'code:block software' which contains the mingw compiler and set the mingw in the path after installation. Other options are also available e.g. Borland C or Turbo C etc.

1.3 Writing first code

Listing 1.1 shows the source code to print the 'Hello World' on the screen. To execute the code in the listing, open the terminal and go to the folder where program is saved and type following commands; which will generate the output on the terminal.

```
$ gcc -o out helloWorld.c
$ ./out                     (in Unix)
$ out                       (in Windows)
```

Explanation Listing 1.1

Note that, C is **case-sensitive** language. Each program contains one and only one 'main' function, which is the starting point of the code. In this listing, Line 6 has the 'main' function, which contains two other keywords i.e. 'int' and 'void'; here 'void' indicates that, the main 'function' does not have any input parameter, whereas 'int' represents that the return type of this function is 'integer'.

Keywords are nothing but predefined words in C, which can not be used for declaring **Identifiers** i.e. name of variables, functions and array etc. Table 1.1 shows the list of keywords available in C and C++ languages. Further, the terms 'function', 'parameters' and 'return type' are discussed in Chapter 4. Further, all the statements inside the brackets ({ }) (i.e. between Line 7 and 11) are belongs to function 'main'.

In Line 9, keyword 'printf' is used, which prints the characters on the screen; this 'prinf' keyword is defied in 'standard input/output (i.e. stdio.h)' library, which is imported to the current program using Line 4. The '.h' files are known as 'header files' and C-programs are saved with extension '.c'. In the tutorial, all the file names are shown at the top of the code (See Line 1).

Note that, multiple lines can be commented by using /* and */ as shown in Lines 11-14; whereas '//' is used for single line comments (e.g. Line 1). The comments make code more readable; e.g. in this listing, functions of all the lines are described as comments.

Also, it is the semicolon sign ';' which terminates the line, not the white spaces; e.g. Line 9 can be written in multiple lines as shown in Lines 11-14. If we uncomment these lines, then we will get the same output as Line 9.

Lastly, 'return 0 is used at Line 16, which means there is nothing to return. This line is required because we mention the return type as 'int' at Line 6.

Listing 1.1: Print Hello World

```
1   //helloWorld.c
2
3   //header file: stdio.h
4   #include <stdio.h> // required for printf, scanf etc.
5
6   int main(void) //return type: int;  input arguments: void i.e. none
7   {
8
9       printf("Hello World\n");
10
11      /* printf(
12          "Hello World\n"
13          );
```

(continues on next page)

(continued from previous page)

```
    */

    return 0; //return 0 i.e. int,
}

/* Outputs
Hello World
*/
```

1.4 Printing values in Binary, Hex, Octal and Decimal formats

In this section, we will learn to print the data in different formats i.e. Hexadecimal, Octal and Decimal. This can be done using identifiers '%d (int)', '%x (hex)' and '%o (oct)' in the 'printf' statements, as shown in Listing 1.2. For further understanding, see the comments and outputs of the code in the listing.

Listing 1.2: Printing numbers in different formats

```
//hecDec.c

#include <stdio.h>

int main(void){

    int x = 15;

    // hexadecimal number starts with 0x
    // h is defined as hex 0xb = 13(oct) = 11(dec)
    int h = 0xb;

    // octal number, starts with 0
    int o = 013;

    printf(" hex value of x = %x\n", x); // hex
    printf(" oct value of x = %o\n", x); // octal
    printf(" decimal value of x = %d\n\n", x); // decimal

    printf(" hex value of h = %x\n", h); // hex
    printf(" oct value of h = %o\n", h); // octal
    printf(" decimal value of h = %d\n\n", h); // decimal

    printf(" hex value of o = %x\n", o); // hex
    printf(" oct value of o = %o\n", o); // octal
    printf(" decimal value of o = %d\n\n", o); // decimal


    return 0;
}

/*
 hex value of x = f
 oct value of x = 17
 decimal value of x = 15

 hex value of h = b
 oct value of h = 13
 decimal value of h = 11

 hex value of o = b
```

(continues on next page)

(continued from previous page)

```
oct value of o = 13
decimal value of o = 11
*/
```

1.5 Conclusion

In this chapter, we see the list of compilers for C. Also, we wrote simple ‘Hello World’ programs, which illustrate the concept of ‘main function’, ‘return’ and ‘printing output on screen’. Further, we learn to print the numeric-outputs in different formats as well.

All know that the drop merges into the ocean, but few know that the ocean merges into the drop.

–Kabeer

Chapter 2

Data types

2.1 Introduction

C provides different kinds of data types to store the various values e.g. integer, decimal and strings etc. as shown in Listing 2.1. These data types can be categorized in two ways i.e. 'built-in data types' and 'user defined data types'; which can be further classified as 'variables' and 'constants'. In this chapter, these data types are discussed.

2.2 Variable example

Before discussing the data types, let us see one example of variables and their types. In Listing 2.1, three **variables** are defined i.e. x and z of integer type (Line 7) and y of float type (Line 8), using 'int' and 'float' keyword respectively. Note that, the 'float' and 'int' keywords are used to store 'decimal' and 'integer' values respectively.

Next, the values of 'x' and 'y' are set to 2 and 2.7 in Lines 10 and 11 respectively. Finally, these two values are added in Line 12, whose output is 4 (not 4.7). The decimal values is ignore in the output because 'z' is declared as 'int', which can store only 'integer' values, not the 'float' values.

Listing 2.1: Variable and data type

```
 1  // variableEx.c
 2
 3  #include <stdio.h>
 4
 5  int main(){
 6
 7          int x, z;
 8          float y;
 9
10          x = 2;
11          y = 2.7;
12          z = x + y; // z = 4 (not 4.7 because z has type integer)
13
14          printf("z = %d", z);
15  }
16
17  /* Outputs
18  z = 4 */
```

2.3 Built-in data types

Table 2.1 presents the 5 built-in data types, which are essential for design purposes. The data types in Table 2.1, can be further mixed with the other keywords i.e. 'signed', 'unsigned', 'short' and 'long', which will specify the range and required memory for the variables. Some of these combination are shown in Table 2.2 along with the required memory sizes and ranges. Lastly, the memory size can be checked for different data types with keyword 'sizeof' as shown in Listing 2.2,

Table 2.1: Built-in data types

Type	Keyword
Character	char
Integer	int
Floating point	float
Double	double
Boolean	bool

Table 2.2: Required memory and range for data types

Type	Size	Typical Range
char	1byte	-128 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-128 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647

Listing 2.2: Required memory for different data types

```c
//dataTypeEx.c

#include <stdio.h>

int main() {

   printf("Size of char : %d\n", sizeof(char));
   printf("Size of int : %d\n", sizeof(int));
   printf("Size of double : %d\n", sizeof(double));

   return 0;
}

/* Outputs
Size of char : 1
Size of int : 4
Size of double : 8 */
```

2.4 Enumerated data types

Custom data type can be defined using the keyword 'enum', which are known as 'enumerated data types'. Listing 2.3 presents the example of 'enumerated data type'. Note that, 'if statement' (at Line 16) is used in this example which is discussed in Section 3.3.1.

Explanation Listing 2.3

In the listing, 'enumerated data type' is defined at Lines 8-10. Here, the data type 'stateReg' has two possible values i.e. 'Win' and 'Loose'. Next, a variable 'currentState' of type 'stateReg' is defined at Line 12. Then, in Line 13, variable 'dice' is declared, with 'initial value = 6'.

In Line 16, ‘if statement’ is used, which is discussed in Section 3.3.1. Lines 16-21 indicates that if dice value is 6, then value of variable ‘currentState’ will be set as ‘Win’, else it will be set as ‘Loose’. Lastly, Lines 24-27 prints the message based on ‘currentState’ value i.e. if it is ‘Win’ then the message ‘You won the game’ will be printed i.e. Line 25, else Line 27 will be printed.

Listing 2.3: Enumerated data type

```
1  //enumDataEx.c
2
3  #include <stdio.h>
4
5  int main() {
6
7      //enumerated data-type `stateReg' with values Win and Loose
8      enum stateReg{
9          Win, Loose
10     };
11
12     stateReg currentState; // define variable 'currentState' of type 'stateReg'
13     int dice = 6;
14
15     // set status to win , if number on dice is 6
16     if (dice == 6){
17         currentState = Win;
18     }
19     else{ //otherwise loose
20             currentState = Loose;
21     }
22
23     //print message according the currentState
24     if(currentState == Win)
25         printf("You won the game...");
26     else
27         printf("You loose the game...");
28
29     return 0;
30 }
31
32 /* Outputs
33 You won the game...  */
```

2.5 Constants: define and const

Constants can be defined in two ways i.e. using ‘#define’ and ‘const’ keywords as shown in Listing 2.4.

Explanation Listing 2.4 In the listing, one new header is introduced i.e. ‘math.h’. This header contains various mathematical operations e.g. sin, abs, sqrt and pow etc. In the listing ‘pow (i.e. power)’ is used at Line 14, where area of the circle is calculated i.e. πr^2. Further, constants are defined using keywords ‘define’ (Line 6) and ‘const’ (Line 11). Then, these two constant values are used at Line 14 to calculate the area. Constant at Line 7 is used to show that a string can be defined as constant as well; which is used at Line 17 with ‘printf’ statement. Also, note that constants can not be modified e.g. if you uncomment the Lines 19 or 20, then error will be generated.

Listing 2.4: Constants: define and const

```
1  //defineConstantEx.c
2
3  #include <stdio.h>
4  #include <math.h>
```

(continues on next page)

(continued from previous page)

```c

#define pi 3.14
#define blog "PythonDSP"

int main(void)
{
    const int radius=2;
    float area;

    area = pi * pow(radius, 2); // pow(2, 3) = 2*2*2
    printf("Area of circle = %f\n", area);

    printf("Blog name: %s", blog);

    // pi = 3; // error as constant can not be modified
    // radius = 3; // error as constant can not be modified
    return 0;
}

/* Outputs
Area of circle = 12.560000
Blog name: PythonDSP */
```

2.6 Reading data from terminal with ‘scanf’

Listing 2.5 shows the usage of ‘scanf’ command for getting data from the users.

Explanation Listing 2.5

In the listing, at Line 7, variable ‘name’ is defined of type ‘char’ with maximum length 30 i.e. char[30]. When we run the code, after execution of Line 9, the message ‘Enter your name’ will be displayed. Note that, ‘t’ is used in Line 9, which puts the one-tab-space after the message. Next, compiler will stop at Line 10, to get the input value from the user for the variable ‘name’. Once, we provide the name e.g. Meher, the compiler will go to next line and print the message ‘Hello Meher’.

Further, ‘scanf’ command reads the input values from the user till a white space appear, i.e. if we type the value as ‘Meher Krishna’, the compiler will read the word ‘Meher’ only, as there is white space after that and the second part will be omitted from the result.

Note: Lastly, see Listing 2.6 for more ‘scanf’ commands; **also see the comments carefully in the listing**. It’s better to put one space before % sign, in every scanf-statement to avoid problems, as we did in Line 19 of Listing 2.6.

Listing 2.5: Getting data from user using ‘scanf’

```c
1  //scanfEx.c
2
3  #include <stdio.h>
4
5  int main(){
6
7     char name[30]; // variable 'name' of type character with max length 30
8
9     printf("Enter your name: \t"); // show message on the termainal
10    scanf("%s", name); // get the value of name from user
11
12    printf("Hello %s", name);  // Print Hello + name-provided-by-user
```

(continues on next page)

(continued from previous page)

```

   return 0;
}

/* Outputs
Enter your name:    Meher
Hello Meher */
```

Listing 2.6: More ‘scanf’ commands : look comments carefully

```
 1  //scanfEx2.c
 2
 3  #include <stdio.h>
 4
 5  int main(){
 6
 7      int int_value;
 8      float float_value;
 9      char char_value = 'W';   // stores only one character
10      char str_value[30] = "West"; // char-array of size 30 : can store 30 characters
11
12      printf("Enter integer: \t");
13      scanf("%d", &int_value);  // see & sign for reading values.
14
15      printf("Enter float: \t");
16      scanf("%f", &float_value);
17
18      printf("Enter character: \t");
19      scanf(" %c", &char_value);    // look for space at the beginning i.e. "space %c"
20
21      printf("Enter string: \t");
22      scanf("%s", &str_value[0]);  // or use scanf("%s", str_value)
23
24      printf("\nInteger: %d", int_value);  // print int
25      printf("\nFloat: %f", float_value);  // print float
26      printf("\nCharacter: %c", char_value);  // print char
27      printf("\nString: %s", str_value);  // print string
28     return 0;
29  }
30
31  /* Outputs
32  Enter integer:  32
33  Enter float:    123.4
34  Enter character:        M
35  Enter string:   Meher
36
37  Integer: 32
38  Float: 123.400002
39  Character: M
40  String: Meher */
```

Important: Note that, the ‘character (char a)’ is initialized within ‘single quote’ whereas the character-array (char a[ ]) is initialized within ‘double quote’ as shown in Lines 9 and 10 respectively of Listing 2.6.

2.7 Character array

In Listing 2.6, we saw that the datatype 'char' can store only 1 character; whereas character-array can store multiple character based on it's size. Character array can be defined in two ways e.g. character array 'a' can be defined as '**char** a[ ]' and '**const char** *a, as shown in Listing 2.7. Note that, 'char [a]' value can not be assigned directly in the program (Line 12), but it can be obtained using 'scanf' command (Line 14); whereas value can be assigned to 'const char *a' in the program (Line 17), but it can not be obtained by using 'scanf' command. Please uncomment these lines to see the errors.

Listing 2.7: Character array

```c
1  //charArrayEx.c
2  
3  #include <stdio.h>
4  
5  int main(){
6  
7      char a[30] = "Meher";  // character array with initialization
8      const char *b = "Patel"; // character array using pointer and with initialization
9  
10     printf("Initial name : %s %s\n", a,b);
11 
12     // a = "Krishna";  // not allowed
13     printf("Enter new name : ");
14     scanf("%s", a);
15 
16     // scanf("%s", b);  // not allowed
17     b = "New World";
18     printf("%s %s", a,b);
19 
20     return 0;
21 }
22 
23 /* Outputs
24 Initial name : Meher Patel
25 Enter new name : Hello
26 Hello New World */
```

2.8 Conclusion

In this chapter, we saw the basic data types i.e. 'in-built data types' and 'user-defined data types'. Further, these data types are used as 'constant' and 'variables'. In next chapter, we will see various operators and control structures to use these data types for performing various tasks.

It is not so much that you are within the cosmos, as that the cosmos is within you.

–Meher Baba

Chapter 3

Operators and control statements

3.1 Introduction

C provides various operators e.g. algebraic and boolean operators etc., which can be used to perform different mathematical operations. Further, various control structure of C can be used to perform these tasks repetitively or under suitable conditions. In this chapter, such operators and control structures are used to perform operations on various data-types.

3.2 Operators

Operators are the symbols which are used to perform certain operations on the data e.g. addition, subtraction and comparison etc. There are various types of operators in C, which are shown in this section; also, most of these operators are used with decision statements, therefore usage of these operators are shown in Section 3.3.

3.2.1 Arithmetic operators

Table 3.1 shows the list of arithmetic operators in C. These operations are used to perform various mathematical operations on 'integer', 'float' and 'double' data types. Functions of all the operators are straightforward except for '++' and '–', which are used to increment or decrement the value of the variable by '1' respectively. These operators can be used as 'i++ or i–' and '++i or –i'. The '++i' operation indicates that 'increase the value of by '1' and then use that value; whereas 'i++' indicates that 'use the value of i first' and then increment it by 1. These two operations are shown in Listing 3.1.

Table 3.1: Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder) (e.g. 3%2 = 1)
++	"Increment (e.g. if a = 1, then a++ = 2)"
–	"Decrement (e.g. if a = 1, then a– = 0)"
+= -= *= /=	short notation (e.g.'a += 3' indicates 'a = a+3'

Explanation Listing 3.1

In the listing, the variable 'i' is initialized as '1' at Line 7, and printed at Line 9. Next, 'i++' operation is printed at Line 12; note that, first '1' is printed by this line; and then the value is increased. This increase in value is verified by print 'i' again at Line 13, where the result is 2. Next, '++i' operation is used at Line 16, where the value is increase first and then printed, therefore the result is '3'. Finally, to check the value stored in 'i', it is printed again at Line 17.

Listing 3.1: Difference in 'i++' and '++i'

```
1   // incrementEx.c
2
3   #include <stdio.h>
4
5   int main(void)
6   {
7       int i = 1;
8
9       printf("i = %d\n\n", i);  // 1
10
11      // print first, then increment
12      printf("i++ = %d (i.e. access first and then increment): \n", i++); // 1
13      printf("i = %d\n\n", i); // 2 (i.e. i is incremented by above statement)
14
15      // increment first, then print
16      printf("++i = %d (i.e. increment first and then access):\n", ++i); // 3
17      printf("i = %d\n", i); // 3
18
19      return 0;
20  }
21
22  /* Outputs
23  i = 1
24
25  i++ = 1 (i.e. access first and then increment):
26  i = 2
27
28  ++i = 3 (i.e. increment first and then access):
29  i = 3
30  */
```

3.2.2 Relational operators

Relations operators are used of checking various equality conditions e.g 'greater than', 'equal' or 'less than' etc. These operators are shown in Table 3.2.

Table 3.2: Relational Operators

Condition	Symbol
equal	==
not equal	!=
greater	>
smaller	<
greater or equal	>=
smaller or equal	<=

3.2.3 Bitwise operators

Bitwise operators are used with boolean data type to calculate the results of boolean expressions or to perform shift operations on boolean data. These operators are shown in Table 3.3. Some of these operations are shown in Listing 3.2.

Table 3.3: Bitwise operators

Operators	Symbol
OR	|
AND	&
XOR	^
NOT	~
Left Shift	<<
Right Shift	>>

Explanation Listing 3.2

Most of the operations in the listing are straightforward i.e. right shift(Line 12), left shift (Line 13) and xor (Line 17). **But, look at the 'not operation' carefully, it may generate errors in the result in NIOS design.** Since, type 'int' is 32 bit long, therefore, a = 2 is saved as '0000-0002' and when 'not' operation is performed on it, then result is 'ffff-fffd' (not '0000-000d'), at Lines 20-21; therefore, we need to perform '& (and)' operation with the result, to suppress all the 'f' in the result (Line 24).

Listing 3.2: Bitwise operator

```
 1  // shiftOpEx.c
 2
 3  #include <stdio.h>
 4
 5  int main(){
 6      int a = 0x2;  // 0010
 7
 8      int b, c;
 9      int xor_op, not_op_bad, not_op_good;
10
11      // shift right by one bit and save to b
12      b = a >> 1;  // 0001 = 1
13      c = a << 2;  // 1000 = 8
14
15      printf("a = %x, b = %x, c = %x\n", a, b, c);
16
17      xor_op = a ^ b; // 0011 = 3
18      printf("xor_op = %x\n", xor_op);
19
20      not_op_bad = ~a; // fffffffd (expected output = 1101 = 13 = 'd')
21      printf("not_op_bad = %x\n", not_op_bad);
22
23      // suppress all 'f' (i.e. make them 0) by 'and-operation' of 'a' with '0000000f' = 15
24      not_op_good = ~a & 0xf; // or not_op_good = ~a & 15;
25      printf("not_op_good = %x\n", not_op_good);
26  }
27
28  /* Outputs
29  a = 2, b = 1, c = 8
30  xor_op = 3
31  not_op_bad = fffffffd
32  not_op_good = d
33  */
```

3.2.4 Logical operators

Logical operators are used to perform logic 'and', 'or' and 'not' operations, which are shown in Table 3.4. Unlike, 'bitwise logic operators' these operators do not check the second operator if the first operator gives the result e.g. in '0 && 1 = 0', the result does not depend on the value of second operand i.e. '1'; because first operand is '0', therefore the result will be zero, whether the second operand is '0' or '1'.

Table 3.4: Logical operators

Operator	Description
&&	Logical AND operator
||	Logical OR Operator
!	Logical NOT Operator

3.3 Decision statements

Control structure can be devided into two categories i.e. decision statements and loops. In this section, simple C codes are presented to explain decision statements available in the language; whereas loops are discussed in next section.

3.3.1 If-else statements

If-else statements are used to define different actions for different conditions. Symbols for such conditions are given in Table 3.2, which can be used as shown in Listing 3.3. Three examples are shown in this section for three types of statements i.e. **if**, **if-else** and **if- 'else if' -else**.

Explanation Listing 3.3

Listing 3.3 checks whether the number stored in variable 'x' is even or not.

In the listing, variable 'x' of type 'int' is defined at Line 8; also, this variable is initialize with value 3. In Lines 11 and 17, the '%' sign calculates the remainder of division x/2 and checks whether remainder is zero or not respectively. Then, Lines 12-13 will be printed if remainder is zero, else Line 18-19 will be printed.

Notice that, { } are used with if-statements e.g. at Lines 11 and 14. These brackets tell the compiler that, all the statements within them must be executed only when the condition in corresponding 'if' statement is satisfied i.e. 12-13 will be executed, when Line 11 is true. Further, if we remove the brackets from these line, then Line 12 will depend on the 'if' statement, whereas Line 13 will be printed all the time.

Listing 3.3: If statement

```c
1  // ifEx.c
2  // checks whether number is even or odd...
3
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int x=3;
9
10     // check if number is even
11     if(x % 2 == 0){
12         printf("Number is even.");
13         printf("Thank you...");
14     }
15
16     // check if number is odd
17     if(x % 2 != 0){
18         printf("Number is odd.");
19         printf("Thank you...");
20     }
21
22     return 0;
23 }
```

(continues on next page)

(continued from previous page)

```

/* Outputs
Number is odd.
Thank you...
*/
```

Explanation Listing 3.4

As we know that a number can not be even and odd at the same time. In such cases, we can use if-else statement. Listing 3.3 can be rewritten using 'if-else' statement as shown in Listing 3.4.

Here, only one condition is specified i.e. at Line 10. Compiler will check this line, if this is true, then Line 11-12 will be printed, otherwise compiler will go to 'else' statement at Line 14 and Lines 15-16 will be printed.

Listing 3.4: If-else statement

```
1  //ifElseEx.c
2  // checks whether number is even or odd...
3
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int x=3;
9
10     if (x % 2 == 0){
11         printf("Number is even.");
12         printf("Thank you...");
13     }
14     else{
15         printf("Number is odd.");
16         printf("Thank you...");
17     }
18
19     return 0;
20 }
21
22 /* Outputs
23 Number is odd.
24 Thank you...
25 */
```

- In previous case, there were only two conditions which were implemented using 'if-else statement'. If there are more than two conditions then 'else if' can be used as shown in this example. Further, 'if else-if else' block can contain any number of 'else-if' statement between one 'if' and one 'else' statement.

Note: Note that, the operator 'or' at Line 18 of Listing 3.5, will generate error, if we execute the code using 'gcc ElseIFEx.c -o out', as this operator for C++ compiler only. This code can be executed with command 'g++ ElseIFEx.c -o out'.

Explanation Listing 3.5

In this listing, Line 11 get the value of variable 'grade' from the user. The possible value of grades are A, a, B, b, C, c, D and d. If grade is 'A or a', then Line 15 will be printed. Note that, the 'or' operation is used in three ways, i.e. '||', '|' and 'or' at Lines 14, 16 and 18 respectively. If grade value provided by the user is 'B or b', then compiler will first check the Line 14, which is false, therefore it will go to next 'else-if' statement at Line 16, since it is true, therefore Line 17 will be printed; and compiler will leave the if-block. If we provide the wrong grade, e.g. Z, then compiler will check all the statements i.e. Lines 14, 16, 18 and 20; and finally reach to 'else' block and print the Line 23.

Listing 3.5: ‘If else-if else’ statement

```c
// ElseIfEx.c
// print message based on grades...

#include <stdio.h>

int main(void)
{
    char grade;

    printf("Enter the grade - A, B, C or D: \t");
    scanf(" %c", &grade);  // get the value from user

    // if grade is A or a
    if (grade == 'A' || grade == 'a')   // logical operator
        printf("Excellent Student");
    else if ((grade == 'B') | (grade == 'b'))   // bitwise operator
        printf("Very Good Student");
    else if (grade == 'C' or grade == 'c')  // logical 'or' operator
        printf("Good Student");
    else if (grade == 'D' or grade == 'd')   // or is C++ feature (not C)
        printf("Need improvements");
    else
        printf("Invalid grade");

    return 0;
}

/* Outputs
Enter the grade - A, B, C or D:     b
Very Good Student
*/
```

Explanation Listing 3.6

We can use various if statements inside the other if statements, which are known as nested statements. Listing 3.6 is the example of **nested loop**, where divisibility of the number with 2 and 3 is checked.

The listing checks whether the number is divisible by 2 and 3 using nested if-else statements. Nested statements are used for ‘if statement’ at Line 10. Here, line 10 checks whether the number is division by 2 or not; if divisible, then compiler goes to nested loop (at line 11) and checks whether number is divisible by 3 or not; if divisible, then Line 12 will be printed, else Line 15 and 16 will be printed. Note that, line 16 prints the value of remainder when number is divided by 3. Next, if number is not divisible by 2 (at Line 10), then compiler will directly go to the Line 19; and if number is divisible by 3, then Line 20 will be printed. If number is not divisible by 3 as well, then compiler will go to Line 22 (i.e. else statement) and finally Lines 23-25 will be printed.

Listing 3.6: ‘Nested If else-if else’ statement

```c
1  // ifElifElse.c
2  // Nested if-else statement to check the divisibility with 2 and 3.
3
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int x=4;
9
10     if (x % 2 == 0){
11         if(x % 3 == 0){
12             printf("Number is divisible by 2 and 3.");
```

(continues on next page)

(continued from previous page)

```
        }
        else{
            printf("Number is divisible by 2 only\n");
            printf("x%%3 = %d\n", x%3);  // note : two %% are used to print '%'
        }
    }
    else if (x % 3 == 0){
        printf("Number is divisible by 3 only.");
    }
    else{
        printf("Number is not divisible by 2 and 3\n");
        printf("x%%2 = %d\n", x%2);
        printf("x%%3 = %d\n", x%3);
    }

    return 0;
}

/* Outputs
Number is divisible by 2 only
x%3 = 1
*/
```

3.3.2 Switch-case-default statements

Similar to 'if statements', the switch-case statements can be used to perform certain operations, only if the specific conditions are met, as shown in Listing 3.7.

Explanation Listing 3.7

Operation of this listing is same as the Listing 3.5, but the condition is only checked for uppercase letter. We can add more case-statements for considering the lowercase letters as well. Here, the switch-statement depends on variable 'grade' (Line 10); and it is initialized with value 'B' at Line 8; and Lines 10-22 are the switch-statement, which contain various 'case-statements' (i.e. Lines 11, 14 and 17) and one 'default' statement at Line 20.

When we execute this code, the compiler will reach to 'switch statement (Line 10)' and then first go to Line 11 and compare the value of grade i.e. 'B' with 'A', since these are not equal therefore it will go to next 'case statement' at Line 15. Now, the condition is matched, therefore Line 15 will be printed out and then Line 16 will break the execution of switch statement, i.e. the complier will exit from the switch statement and will reach to line 23. '**Note that, unlike if-else-statements, the compiler does not automatically quit the switch-statement, a separate 'break' statement is required to exit the case. If we do not use 'break' statement, then compiler will check all the 'cases' and finally execute the 'default' block. More specifically, we use the 'break' statement to avoid the execution of 'default' block.**

Listing 3.7: Switch-case statement

```
1   //switchCase.c
2   // print message based on grade-value
3
4   #include <stdio.h>
5
6   int main(void)
7   {
8       char grade = 'B';
9
10      switch(grade){
11          case ('A'):
```

(continues on next page)

(continued from previous page)

```c
            printf("A: performance is excellent");
            break;
        case 'B' :
            printf("B: performance is good");
            break;
        case 'C' :
            printf("C: performance is not bad");
            break;
        default :
            printf("Please, work hard...");
    }

    return 0;
}

/* Outputs
B: performance is good
*/
```

3.3.3 Conditional operator (? :)

The conditional operator can be used as 'if-else' statements as shown Listing 3.8 and Listing 3.9. In this section, 'const char *' is used to define character-array, which is discussed in Section 2.7.

Explanation Listing 3.8

In conditional operator, the condition is specified before '?' and the 'true and false' results are seperated by ':'. For example, in this listing, condition '(a %2)== 0)' is specified before '?' at Line 14. Note that, the result before ':' (i.e. even) is assigned to variable 'result' if the condition is true, otherwise the result after ':' (i.e. odd) will be assigned to variable 'result'. Finally, Line 16 will display the message for even or odd number.

Listing 3.8: Condition operator as 'if-else' statement

```c
 1  // conditionalOperatorEx.c
 2  // check even and odd number
 3
 4  #include <stdio.h>
 5
 6  int main(void)
 7  {
 8      int a;
 9      const char *result;  // character array
10
11      printf("Enter a number:  ");
12      scanf("%d", &a);
13
14      result = (a%2 == 0 ) ? "even" : "odd";
15
16      printf("number is %s", result);
17
18      return 0;
19  }
20
21  /* Outputs
22  Enter a number:  4
23  number is even
24  */
```

Explanation Listing 3.9

Conditional operator can be used as 'if- else if- else' statement as well. For this, instead of assigning the value for 'false condition' after ':', we should put another condition-checking statement; e.g. in the listing, Line 14 ends with ':' and then at Line 15 starts with another condition. Now, if the condition at Line 14 (i.e. a % == 0) does not satisfy, then condition at Line 15 will be checked. Lastly, if none of the condition is matched, then Line 17 will be printed.

Listing 3.9: Condition operator as 'if-else if-else' statement

```
 1  // conditionalOperatorEx2.c
 2
 3  #include <stdio.h>
 4
 5  int main(void)
 6  {
 7      int a;
 8
 9      const char *result;
10
11      printf("Enter a number:  ");
12      scanf("%d", &a);
13
14      result = (a % 2 == 0 ) ? "number is divisible by 2" :
15               (a % 3 == 0) ? "number is divisible by 3" :
16               (a % 5 == 0) ? "Number is divisible by 5" :
17               "number is not divisible with 2, 3 and 5";
18
19      printf(result);
20
21      return 0;
22  }
23
24  /* Outputs
25  Enter a number:  9
26  number is divisible by 3
27
28  Enter a number:  29
29  number is not divisible with 2, 3 and 5
30  */
```

3.4 Loops

Loops are used to perform the operations repetitively. Three types of loop conditions are discussed in this section i.e. 'for loop', 'while loop' and 'do while loop'.

3.4.1 For loop

Note: Note that, 'int j' inside the 'for loop', i.e. Line 13 of Listing 3.10, will generate error, if we execute the code using 'gcc forloop.c -o out', as this operation is only allowed with the C99 or C11 compiler. This code can be executed with command 'gcc -std=c99 forLoop.c -o out' or 'g++ forloop.c -o out'.

Explanation Listing 3.10

In the listing, two 'for loops' are used i.e. at Line 13 and Line 25. In Line 13, variable 'i' is used which is declared at Line 8; and there is only statement inside this loop i.e. Line 14, which prints the value of 'i'. First, the loop will start with $i = 0$ at Line 13 and the statement inside the loop will be printed till the value of $i < 5$. And after executing the statement at Line 14, the value of the variable 'i' will be increased by 1 using 'i++' statement in Line 13. When the value of i becomes 5, then compiler will not execute the Line 14 and go outside the loop i.e at Line 16. Then Line 19 prints the current value of ‘i’, whose value is increased to 5, due to ‘i++’ statement in the ‘for loop’ at Line 13.

Loop at Line 25, is similar to Line 13, but ‘**int j = 0**’ is used here, instead of ‘**j = 0**’. This ‘int’ keyword will create a local variable ‘j’ for the loop and will be **independent** of the variable ‘j’ at Line 8. Hence, the value of ‘j’ printed by the Line 31 will be 2 (not 8). Value is not 8 because the variable ‘j’ which has the value 8 (i.e. ‘j’ at line Line 25) is local to ‘for loop’, and will be discarded by the compiler after coming out of the loop.

Further, the for loop can be executed in reverse direction as well using ‘i–’ operation, as shown in Listing 3.11.

Note: Note that, for(;;) is used for infinite loop; e.g. if we replace Line 13 with ‘for(;;)’ then the loop will continue forever. Use ‘ctrl+C’ to stop the loop.

Listing 3.10: For loop

```
 1  // forLoop.c
 2  // loops to print numbers
 3
 4  #include <stdio.h>
 5
 6  int main(void)
 7  {
 8      int i = 0, j = 2;
 9
10
11      // ************** LOOP 1 **************************
12      // print 0 to 4
13      for (i=0; i<5; i++){
14          printf("%d, ", i); //0, 1, 2, 3, 4,
15      }
16
17      // value of i is 5
18      printf("\n");
19      printf("Value of i = %d", i); // Value of i = 5
20
21      // ***************** LOOP 2 ************************
22      // 0 to 7
23      printf("\n");
24      // 'int j' creats a new variable 'j' for the loop
25      for (int j=0; j<8; j++){ // note 'int j' is used
26          printf("%d, ", j); // 0, 1, 2, 3, 4, 5, 6, 7,
27      }
28
29      // value of j is 2 (not 8)
30      printf("\n");
31      printf("Value of j = %d", j); // Value of j = 2
32      return 0;
33      // ************************************************
34  }
35
36  /* Outputs
37  0, 1, 2, 3, 4,
38  Value of i = 5
39  0, 1, 2, 3, 4, 5, 6, 7,
40  Value of j = 2
41  */
```

Listing 3.11: For loop in reverse direction

```c
// reverseOrder.c

#include <stdio.h>

int main(){
    int i;

    for (i=10; i>0; i--) // loop in reverse direction
        printf("%d, ", i);

    return 0;

    }

/* Outputs
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
*/
```

3.4.2 While loop

While loop uses a condition-check statement; and the loop continues till the condition is true, as shown in Listing 3.12.

Explanation Listing 3.12

> In the listing, the variable 'i' is initialized with 0 (Line 8). Then 'while loop' is used at Line 10, with condition '(i $<$ 5)', therefore the loop will continue till (i $<$ 5). Further, the loop contains two statements i.e. Line 11 which prints the value of 'i' and Line 12 which increments the value of i by 1. When the value of 'i' is reached to 5, then the compiler will exit the loop and reach to Line 14.

Note: Note that, while(1) is used for infinite loop; e.g. if we replace Line 10 with 'while(1)' then the loop will continue forever. Use 'ctrl+C' to stop the loop.

Listing 3.12: While loop

```c
1  // whileLoop.c
2  // while loop to print the numbers
3
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int i=0;
9
10     while(i < 5){
11         printf("%d,", i);
12         i = i + 1;
13     }
14
15     return 0;
16 }
17
18 /* Outputs
19 0, 1, 2, 3, 4,
20 */
```

3.4.3 Do-while loop

Do-while loop is not very popular method for creating loops, but can be used when we want to execute the statement at least once.

Explanation Listing 3.13

The listing is similar to 'while loop', except the condition is stated at the end of the loop i.e. with 'while statement' at Line 13. Since, condition is checked at the end of the loop, therefore it guarantees that the loop will execute at least one time.

Listing 3.13: Do-while loop

```
1  // doWhileLoop.c
2  // do-while loop to print the numbers
3
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int i=0;
9
10     do{
11         printf("%d, ", i);
12         i = i + 1;
13     }while(i < 5);
14
15     return 0;
16 }
17
18 /* Outputs
19 0, 1, 2, 3, 4,
20 */
```

3.5 Continue and Break statements

Continue and Break statements are used with loops, to skip or end the execution of the loop respectively for certain cases.

3.5.1 Continue

Continue statement is used to skip the loop for certain cases as shown in Listing 3.14. Here, 'continue' statement is used inside the 'if statement' (see Lines 12-14); therefore the **rest of the part** of the loop will be skipped whenever the result of 'i%2 is 0' i.e. Line 16 will be skipped. Hence, the values printed are the odd numbers only, as for even number the execution of the loop is skipped. **In the same way, we can use the 'continue statement' with other loops as well**.

Listing 3.14: Continue statement

```
1  // forContinueLoop.c
2  // print only odd numbers by skipping even numbers
3
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int i;
9
```

(continues on next page)

(continued from previous page)

```
    for (i=0; i<10; i++){

        if (i % 2 == 0){
            continue;
        }

        printf("%d, ", i); //0, 1, 2, 3, 4,
    }
}

/* Outputs
1, 3, 5, 7, 9,
*/
```

3.5.2 Break

Break statement exit the loop permanently, instead of skipping the loop for certain cases as shown in Listing 3.15. Here, break statement is used with ‘if statement’ at Lines 14-16. Therefore, the compiler will exit from the ‘while loop’ whenever the result of ‘i%7 is 0’. Hence, the value printed at the output are 0 to 6. **In the same way, we can use the ‘continue statement’ with other loops as well**.

Listing 3.15: Break statement

```
1  // whileBreakLoop.c
2  // exit the loop using 'break' statement
3
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int i=0;
9
10     while(i < 10){
11         printf("%d,", i);
12         i = i + 1;
13
14         if (i % 7 == 0){
15             break;
16         }
17     }
18
19     return 0;
20 }
21
22 /* Outputs
23 0, 1, 2, 3, 4, 5, 6,
24 */
```

3.6 Conclusion

In this chapter, various examples are shown to understand the operators, conditional statements and loop statements. Also, continue and break statements are discussed which are used to skip and break the loops respectively.

Things that are real are given and received in silence.

–Meher Baba

Chapter 4

Functions and Pointers

4.1 Introduction

If certain codes are used repeatedly in the program, then these codes can be written in the form of functions. Function is a block of code which can perform certain operations based on inputs provided to it. Further, writing code using functions make it more readable and manageable as discussed in this chapter. Also, scope of variables and the concept of pointers are discussed in this chapter.

4.2 Function Example

Let us understand the function with an example. Listing 4.1 shows three parts of a function i.e. 'function prototype (Line 7)', 'function definition (Lines 11-15)' and 'function call (Line 21)'.

Function prototype is used to indicate the name of function, which includes the **types** of input arguments along with the return type; e.g. Line 7 shows that the inputs are of 'int' and 'float' types, whereas return type is 'float'.

Function definition includes the functionality of the code along with the return value e.g. Line 13 add the two numbers and return the result from Line 14.

Function call is required to use the function; e.g. in Line 21 'add2Num' function is called with two values 'x' and 'y'. Note that, 'x' and 'y' are passed in the function call, which will be assigned as 'a' and 'b' in Line 11. Then, the result i.e. 'c' will be returned by Line 14, which will be stored in variable 'z' at Line 21. Finally, value of 'z' is printed by Line 23.

Listing 4.1: Function example

```
1  // functionEx.c
2  // defining and using functions to add numbers
3
4  #include <stdio.h>
5
6  // function prototype
7  float add2Num(int, float);  // use this or below line
8
9
10 // function definition
11 float add2Num(int a, float b){
12     float c;
13     c =  a + b;
14     return c;
15 }
16
17 int main(){
```

(continues on next page)

(continued from previous page)

```
    int x=2;
    float y=3.5, z;
    //function call
    z = add2Num(x, y);

    printf("z = %f\n", z);  // z = 5
}

/* Outputs
z = 5.500000
 */
```

4.3 Scope of variable

Variables can be accessed either by all the functions (i.e. global variable) or by a particular function only (i.e. local variable); this accessibility of the variable known as scope of the variable. Three types of scopes can be defined using following variables,

- Automatic variable
- Static variable
- Global variable

4.3.1 Automatic variable

Automatic variables are the variable which are accessible within the function, i.e. inside the function in which these variables are defined. All the variable which we studied till now, were the 'automatic variable'. Also, if these variable are not initialize, then they will contain garbage value as shown in Listing 4.2.

Explanation Listing 4.2

In the listing, there are three 'automatic variables' are defined i.e. x (Line 8), y (Line 13) and z (Line 17). Following are the important points to note about these variables,

- **The variable 'y' is defined inside the brackets at Lines 11 and 15.** Therefore it's scope is within these brackets and will not be accessible from outside; e.g. if we uncomment the Line 22, an error will be reported with message 'error: 'y' was not declared in this scope'.
- Variable 'z' is not initialized, therefore some garbage value will be stored in it, which can be seen by printing it's value, as shown in Line 24.
- Variable 'x' is defined in the main function (without any additional brackets), therefore it will be available for printing at Line 19.

Listing 4.2: Automatic variable

```
1  // autoEx.c
2  // scope of automatic variables
3
4  #include <stdio.h>
5
6  int main(){
7
8      int x=2;
9      printf("x = %d\n", x);
10
11     {   // y is not available outside the bracket i.e.
12         // scope is limited to  block
13         int y=2;
14         printf("y = %d\n", y);
```

(continues on next page)

(continued from previous page)

```
    }

    int z; // uninitialized, garbage value will be stored in it

    printf("printed at the end: x = %d\n", x);

    // // error y is not available here
    // printf("printed from outer block: y = %d\n", y);

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

    return 0;
}

/* Outputs
x = 2
y = 2
printed at the end: x = 2
z = 2893592
*/
```

4.3.2 Static variable

Keyword **'static'** is used to define static variable.Unlike automatic-variables, these variables are automatically initialized with 0. Further, these variables store their values within the function, even if the compiler leave the function, as shown in Listing 4.3.

Explanation Listing 4.3

In the listing, two similar functions are defined i.e. 'incFunc' and 'staticIncFunc' at Lines 10 and 18 respectively. The only difference in these function is that, in staticIncFunc, the variable j is defined as 'static' at Line 21; whereas the variable 'j' at Line 12 is 'automatic vairable', which makes the working of two functions different, as shown below,

- The values of 'automatic variable' is not stored after leaving the function. Therefore, each time the function 'incFunc' is called by Line 33 (which is in the for loop at Line 32), the variable 'j' is initialized with '0' and the output is always '1' at Line 13 and return back by Line 14. Hence, the Line 37 prints the output 'auto_output' as 1.
- The value of 'static variable' is stored within the function definition, and reused by next function call. Hence, the value of 'j' is kept increasing on each function call made by Line 34. Therefore, the final value of output 'static_output' is printed as '5' by Line 38.
- Also, we need not to initialize the variable 'j = 0' at Line 21, as by default, the static variables are initialized with 0.
- Lastly, similar to automatic variables, the static variables are available to their block only, i.e. static variable 'j' at Line 21 is available inside the 'staticIntFunc' only.

Listing 4.3: Static variable

```
1  // staticEx.c
2  // scope of static-variables
3
4  #include <stdio.h>
5
6  int incFunc(int);
7  int staticIncFunc(int);
8
9  // increment function with auto-variable j;
10 int incFunc(int i){
11     // j is initialize as 0 on each call
```

(continues on next page)

(continued from previous page)

```
    int j = 0;  // automatic variable
    j = j + i;
    return j;
}

// increment function with static variable j;
int staticIncFunc(int i){
    // last value is stored in static int
    // i.e. j is not initialize to 0 on each call
    static int j = 0; // by default initialize with 0
    j = j + i;
    return j;
}

int main(){

    int increment_value=1;
    int auto_output;
    int static_output;

    for(int j=0; j<5; j++){
        auto_output = incFunc(increment_value);
        static_output = staticIncFunc(increment_value);
    }

    printf(" output from incFunc: auto_output = %d\n", auto_output);
    printf(" output from staticIncFunc: static_output = %d\n", static_output);

    return 0;
}

/* Outputs
 output from incFunc: auto_output = 1
 output from staticIncFunc: static_output = 5
*/
```

4.3.3 Global variable

Global or External variables are defined outside the functions, which can be used by all the functions in the program, as shown in the Listing 4.4. Similar to static variables, by default, these variables are initialized with value ‘0’.

Explanation Listing 4.4

In the listing, two global variables are defined i.e. ‘g’ and ‘e’ at Lines 5 and 44 respectively. Please look at the below points to understand the differences between these variables i.e. ‘g’ and ‘e’,

- Global variable ‘g’ is defined at the top of the code, whereas the variable ‘e’ is defined at the bottom of the code. Since, ‘g’ is at the top, therefore it can be accessed by all the function without any further statements e.g. ‘g’ is used at Line 34 of function ‘main’ and Line 11 of function ‘g_incFunc’.
- Since, the global variable ‘e’ is defined at the bottom of the code, therefore it can not be find by compiler for other functions and needs further settings. Hence, the ‘extern’ command is added in the functions, to use the global variable ‘e’ as shown in Lines 17 and 25 of function ‘e_incFunc’ and ‘main’ respectively.
- Similarly, if the global variable are defined in different files, then ‘extern’ command must be used.

Listing 4.4: Global variable

```c
// globalEx.c
#include <stdio.h>

// global variable
int g;

int g_incFunc(int);
int e_incFunc(int);

int g_incFunc(int i){ // increment value of g
    g = g + i; // g is accessed in incFunc
    return g;
}

int e_incFunc(int i){  // increment value of `e'
    // extern keyword must be used if global varialbe is not at the top or in other file
    extern int e;
    e = e + i; // g is accessed in incFunc
    return e;
}

int main(){

    // extern keyword must be used if global varialbe is not at the top or in other file
    extern int e;
    int increment_value = 1;
    int g_inc, e_inc;

    for(int j=0; j<5; j++){
        g_inc = g_incFunc(increment_value);
        e_inc = e_incFunc(increment_value);
    }

    printf(" g = %d\n", g); // g is accessed in main()
    printf(" g_inc = %d\n", g_inc);

    printf(" e = %d\n", e); // e is accessed in main()
    printf(" e_inc = %d\n", e_inc);

    return 0;
}

// global variable at the end.
int e = 2;

/* Outputs
 g = 5
 g_inc = 5
 e = 7
 e_inc = 7
 */
```

4.4 Pointers

Pointers are the variables which store the memory location, which are shown in Listing 4.5. In this listing, following points are important for pointers' declarations and their usage,

- To declare a pointer, '*' is used before variable name along with it's data type (see Line 8).

- '&' sign with normal variable' represents the address of the variable (see Line 18).
- To store the address of a variable in pointer-variable, the '& sign' is used with normal variable (see Line 10); where address of variable 'x' is stored in pointer-variable 'y'.
- Name of the pointer variable returns the address stored in it i.e. Line 19.
- To get the values in the address stored by the pointer, * sign is used i.e. Line 20.
- Lastly, values can not be assigned to the pointer-variables e.g. if we uncomment the Lines 14-15, it will generate error.

Listing 4.5: Pointer example

```
1  // pointerEx.c
2  // define and use pointers
3
4  #include <stdio.h>
5
6  int main(){
7      int x = 2;  // variable declaration
8      int *y;    // declaration - pointer variable
9
10     y = &x;    // store address of variable x
11
12     // following lines will generate error
13     // as values can not be assigned to pointers
14     // y = 2;
15     // y = x;
16
17     printf("x =  %d\n", x); // value of x
18     printf("&x = %p\n", &x); // &x = address of x
19     printf("y =  %p\n", y); // y = address of x  (print in pointer-address)
20     printf("*y = %d\n", *y);  // *y = value store at address y = value of x
21
22     return 0;
23 }
24
25 /* Outputs
26 x =  2
27 &x = 0022ff18
28 y =  0022ff18
29 *y = 2
30 */
```

4.5 Pointer to pointer

'**' are used to create a pointer for the pointer as shown in Listing 4.6. Rest of the working for the code is same as Section ref{sec_PointersBasic} e.g. the variable 'z' is the pointer to a pointer i.e. y. Now, we know that, the '*' sign is used to get the value from the pointer; therefore to get the value stored in pointer-to-the-pointer (i.e. z here), we need to use two '*', because first star will return the value stored in the 'z' (i.e. value stored at '&y', or equivalently address of x) and then second '*' will return the value stored in 'y' (or value stored in address of x). For more understanding, please see all the 'cout' statements in the listing.

Listing 4.6: Pointer to pointer

```
// pointerToPointerEx.c
// pointer to pointer

#include <stdio.h>

int main(){
    int x = 4;
```

(continues on next page)

(continued from previous page)

```
    int *y;
    int **z;

    y = &x;
    z = &y;

    printf("x : values of x: %d\n", x);
    printf("*y : values of x: %d\n", *y);
    printf("**z : values of x: %d\n", **z);

    printf("&x: address of x: %p\n", &x);
    printf("y: address of x: %p\n", y);
    printf("*z: address of x: %p\n", *z);

    printf("&y: address of y: %p\n", &y);
    printf("z: address of y: %p\n", z);

    return 0;
}

/* Outputs
x : values of x: 4
*y : values of x: 4
**z : values of x: 4
&x: address of x: 0022ff18
y: address of x: 0022ff18
*z: address of x: 0022ff18
&y: address of y: 0022ff14
z: address of y: 0022ff14
*/
```

4.6 Passing parameters to functions

There are two ways to pass the parameters to the function i.e. pass by values or pass by reference (i.e. passing addresses using pointers). In this section, squares and cubes of the numbers are calculated with these two methods, to understand the differences.

4.6.1 Passing parameters by values

All the functions, which we learned till now were the example of 'parameters passed by values'. Listing 4.7 is another example of it. Here square and cube of the variable 'a' is calculated. **Since, only one value can be returned by the function, therefore two separate functions are written i.e. 'cube' and 'square' to calculate the values.**. This problem of writing two functions can be solved by using 'parameter passed by reference' method, as shown next.

Listing 4.7: Pass by value

```
// passValueEx.c

#include <stdio.h>

// two seperated functions are required to calculate
// square and cube, because one function can return only value
int cube (int );
int square (int );

int main(){
```

(continues on next page)

(continued from previous page)

```c

    int a = 2;
    int c; // for cube value of a
    int s; // for square value of a

    c = cube(a);
    s = square(a);

    printf("cube of %d = %d\n", a, c);
    printf("square of %d = %d\n", a, s);

    return 0;
}

// return cube value
int cube(int i){
    return i*i*i;
}

// return square value
int square(int i){
    return i*i;
}

/* Outputs
cube of 2  =  8
square of 2  =  4
*/
```

4.6.2 Passing parameters by references

If the variables are passed by references, then function does not make the copy of variables; instead work directly on them. This method is good for passing a large number of data to function, otherwise making a copy of variables on each function call, may result in memory issues. Further, it is a good way to calculate and retrieve multiple values from the same function. Here two examples are shown to understand the concept of 'pass by reference'.

Explanation Listing 4.8

In this listing, numbers are swapped using function 'swap', i.e. value of variable 'x' is saved in 'y' and vice versa. Note that, at Lines 6 and 20, 'int *' are used to pass the parameter by reference (instead of 'int' as in pass by values). Next, variables are passed along with '&' operator, in the function 'swap' at Line 13, i.e. address of the variables are passed, instead the variable itself.

Further, since the addresses are passed at Line 13, therefore the changes are made directly on the address, hence the return type for the function is 'void' at Line 6. Lastly, Lines 21-24 perform the number-swapping operation.

Listing 4.8: Pass values by references

```c
1  // passRefFunc.c
2  // pass parameters by reference
3
4  #include <stdio.h>
5
6  void swap (int *, int *);  // * for pass by reference
7
8  int main(){
9
10     int x=2, y=5;
11     printf("x = %d, y = %d\n", x, y); // x = 2, y = 5
```

(continues on next page)

(continued from previous page)

```

    swap(&x, &y);  // pass the address, instead of value
    printf("x = %d, y = %d\n", x, y); // x = 5, y = 2

    return 0;
}

// nothing is returned, as changes are made in address directly.
void swap(int *i , int *j){
    int temp;
    temp = *i; // store value of i in temp variable
    *i = *j; // store value of j in i
    *j = temp; // finally store value of i (i.e. temp) in j
}
```

Explanation Listing 4.9

Function of this listing is same as Listing 4.7, but pass by reference and pass by value methods are mixed together here. At Line 17, variable ‘c’ and ‘s’ are passed by reference, whereas variable ‘a’ is passed by value. Next, calculation of square and cube values are performed by one function only (as oppose to two in Listing 4.7). Since the values are passed by reference, therefore the calculated values of cube and square are stored at the addresses of variables ‘c’ and ‘s’ respectively; since nothing is returned by the function ‘cube_square in this case, therefore ‘void’ is used at Line 25.

Listing 4.9: Pass by reference and value (mixed)

```
1  // passValueEx.c
2  // pass by reference and pass by value together
3
4  #include <stdio.h>
5
6  // by using pass-by-reference, multiple value can be retrieved from same function
7  void cube_square(int *, int *, int);
8
9  int main(){
10
11     int a = 2;
12     int c; // for cube value of a
13     int s; // for square value of a
14
15     // note that a is passed by value
16     // whereas c and s are passed by reference
17     cube_square(&c, &s,  a); // pass by reference &c and &s
18
19     printf("cube of %d = %d\n", a, c);
20     printf("square of %d = %d\n", a, s);
21
22     return 0;
23 }
24
25 void cube_square(int *j, int *k, int i){
26     *j = i*i*i;
27     *k = i*i;
28 }
29
30 /* Outputs
31 cube of 2  =  8
32 square of 2  =  4
33 */
```

4.7 Pointer to function

In this section, we will apply a pointer to the function. These pointers can be very useful for writing handy codes. For example, in Listing Listing 4.10, we can perform 'cube' and 'square' operations with the help of 'single function in main()' by passing the 'operations as arguments'.

Explanation Listing 4.10

- First, note that the function Cube and Square have same 'input types' and 'return types'. **Therefore, we can create a function-pointer which can point to these functions'**.
- Function-pointer is created at Line 9. Please read comments at Line 8 to write the function-pointer.
- Now we can use this pointer as shown in Line 22. Here, the function 'perform_action' can take 'function pointer' as input parameter, and based on this value 'Cube' and 'Square' operation will performed.
- For example, at Line 31 'perform_action(a, cube)' is used, which will invoke the cube function, whereas Line 32 will invoke the square function.

Listing 4.10: Pointer to function

```
 1  // pointerFunc.c
 2
 3  #include <stdio.h>
 4
 5  // pointer to function
 6  // cube and square have save input and return type.
 7  // function point is defined based on these types as below,
 8  // function pointer: typedef return_type (*funcName)(input_type)
 9  typedef int (*operation)(int);
10
11  // return cube value
12  int cube(int i){
13      return i*i*i;
14  }
15
16  // return square value
17  int square(int i){
18      return i*i;
19  }
20
21  // perform_operation based on 'op' value
22  int perform_operation(int i, operation op){
23      return op(i);
24  }
25
26  int main(){
27
28      int a = 2;
29      int b;
30
31      b = perform_operation(a, cube); // cube
32      printf("cube of %d = %d\n", a, b);
33
34      b = perform_operation(a, square); // square
35      printf("square of %d = %d\n", a, b);
36
37      return 0;
38  }
39
40
41  /* Outputs
42  cube of 2  =  8
43  square of 2  =  4
44  */
```

4.8 Conclusion

In this chapter, functions and pointers are discussed. Further, the scope of various type of variables are shown with example. Lastly, it is shown that when the parameters are 'passed by values', then only one value can be retrieve from the function; whereas 'pass by reference' method can be used to calculate and retrieve multiple values from a function.

Only do not turn your freedom into an opportunity to gratify your flesh, but through love make it your habit to serve one another.

–Jesus Christ

Chapter 5

Arrays, string and pointers

5.1 Introduction

Array is collection of data of same type, and each element of the array can be accessed by index number. Also, array has contiguous memory location for it's elements; where first element has the lowest address of memory, and the last number has the highest address of memory . Note that, array and pointers are closely related to each other; in the other words, arrays are always passed by reference to a function. In this chapter, we will discuss the array in details along with it's relationship with pointer.

5.2 Array

Listing 5.1 shows two types of one-dimensional arrays i.e. 'uninitialized array' and 'initialized array' at Lines 8 and 9 respectively. Also, outputs are printed in formatted-manner at Lines 51-61, where outputs are nicely aligned. The '%2d' and '%11.2f' etc. are used for defining the formats (see comments to understand this), which specify the width for printing elements (with respect to previously printed element, see Lines 26 and 28), and right-aligned outputs are printed by these.

Explanation Listing 5.1:

In the listing, array are initialized in following four ways,

- At Line 8, array 'a' is defined of size 5 (i.e. 0 to 4), but it is not initialized; hence elements of array 'a' contains the garbage values (see Line 15 and 46). At Line 15, 'a[0]' is used to print the value of zeroth position of array 'a'. Here, parameter inside the [] is known as 'index' i.e. '0' is the index. Also, value to this array is assigned using 'for loop' at Lines 37-40. Further, the values can be assigned using index as well e.g. a[0]=20 etc.
- At line 9, array 'b' of size 3 is defined; further this array is initialized as well. Values of this array is printed using 'for loop' at Lines 26-29.
- At line 10, size of array 'c' is not defined, but it is initialized with 2 elements; therefore it's size will be set to 2 automatically.
- At line 11, the array 'd' has size 5 and is initialized with only 2 values. Note that, in this case, the rest of the values i.e. position 2-4 are set to 0 (not filled with garbage value), as shown in Lines 19-21.
- Finally, line 13 is commented, where array 'e' is defined without providing it's size and initialization; such arrays are not allowed in C and compiler will report error, if we uncomment this line. We need to define array with initialization or size or both.

Listing 5.1: Array example

```
1 // arrayEx.c
2 // creating and accessing the elements of array
```

(continues on next page)

(continued from previous page)

```

#include <stdio.h>

int main()
{
    int a[5]; // uninitialized array of size 5 i.e. a[0]-a[4]
    float b[3]={2, 4.5, 6}; // initialized array of size 3
    int c[] = {3, 5};
    int d[5] = {3, 5};  // d[2]-d[4] will have zero value (not garbage)

    // int e[]; // error, either provided size or initialize it

    printf("a[0] = %d\n", a[0]);  // uninitialized array has garbage value

    printf("c[0] = %d\n", c[0]); // 3

    printf("d[0] = %d\n", d[0]); // 3
    printf("d[2] = %d\n", d[2]); // 0 (not garbage)
    printf("d[4] = %d\n", d[4]); // 0 (not garbage)

    // print all values of array b
    // %10s create the width of 10 after 'element'
    // and 'value' will be printed as right-aligned e.g. see 4.5 in output
    printf("%s %10s\n", "element", "value");
    for (int i=0; i<3; i++){
      printf("%4d %12.1f\n", i, b[i]); // %12.1f = show minimum 12 integer & 1 decimal
    }

    // assign values of array a
    for (int i=0; i<5; i++){
      a[i] = 2*i;
    }

    // print values of array a
    printf("%s %10s\n", "element", "value");
    for (int i=0; i<5; i++){
      printf("%4d %12d\n", i, a[i]);  // %12.1f = show minimum 12 integer and 1 decimal place
    }
    return 0;
}

/* Outputs

a[0] = 1998543804
c[0] = 3
d[0] = 3
d[2] = 0
d[4] = 0

element      value
   0          2.0
   1          4.5
   2          6.0

element      value
   0            0
   1            2
   2            4
   3            6
   4            8
*/
```

5.3 Arrays are pointers

In previous section, we printed the values stored in the array using ‘index’. We can not print the values of array by print it’s name (as we do for normal variable), because ‘arrays are not variables, but pointer-variables’, therefore their names print the addresses instead of values, as shown in this section.

Explanation Listing 5.2:

> Note that, at Line 9, the name of the variable i.e. ‘x’ is used with ‘printf’ statement , which prints the address of the first element of array i.e. x[0]. In the other words, name of the array is a pointer-variable, which store the location of the first element of the array. Value of first element of array i.e. ‘x[0]’ can be retrieved using ‘*’ operation as shown in Line 10. Similarly, ‘x+1’ (Line 12), is the address of second element of array, whose value can be retrieved by command ‘*(x+1)’ (Line 13). Please note that the difference between ‘*x + 1’ and ‘*(x+1)’ at Lines 14 and 13 respectively.

Listing 5.2: Arrays are pointers

```
1  // arrayPointerEx.c
2  // arrays are pointer
3
4  #include <stdio.h>
5
6  int main(){
7      int x[4] = {2, 4, 6, 8};  // array x of size 4.
8
9     printf("x : address of x[0] = %p\n", x); // array points to address
10    printf("*x : value of x[0] = %d\n", *x); // 2
11
12    printf("x+1 : address of x[1] = %p\n", x+1); // array points to address
13    printf("*(x+1) : value of x[1] = %d\n", *(x+1)); // 4
14    printf("*x+1 : value of x[0] + 1  = %d\n", *x+1); // x[0] + 1 = 3
15
16    return 0;
17 }
18
19 /* Outputs
20 x : address of x[0] = 0x22ff00
21 *x : value of x[0] = 2
22 x+1 : address of x[1] = 0x22ff04
23 *(x+1) : value of x[1] = 4
24 *x+1 : value of x[0] + 1  = 3
25 */
```

5.4 Multi dimensional arrays

To define the multi dimensional array, multiple [] are used e.g. for 2 dimensional array, two [], i.e. x[3][2], are used as shown in Line 7. x[3][2] represents that the array ‘x’ has ‘3 rows’ and ‘2 columns’. Further, this array is initialized as well (Line 8-10). Individual element of this array can be accessed by using index at Lines 14-15. Further, all the elements of this array are printed using ‘for loop’ at Lines 18-23. Similarly, we can define higher dimensional array e.g. three dimensional array is defined as ‘y[4][3][6]’.

Listing 5.3: Multi dimensional arrays

```
1  // multiDimArray.c
2  // create multidimensional array
3
4  #include <stdio.h>
5
6  int main(){
```

(continues on next page)

(continued from previous page)

```
    int x[3][2] = {
                {5, 6},  // row 1
                {7, 9},  // row 2
                {1, 2}   // row 3
            };

    // print specific element of array
    printf("x[0][1] : %d\n",  x[0][1]); // 6
    printf("x[2][0] : %d\n", x[2][0]); // 1

    // print all elements of array
    for (int i=0; i<3; i++){        // select all rows
        for (int j=0; j<2; j++){     // select all columns
            printf("%d, ", x[i][j]); // print values of x[i][j]
        }
        printf("\n"); // line change for each row i.e. i
    }
    return 0;
}

/* Output
x[0][1] : 6
x[2][0] : 1
5, 6,
7, 9,
1, 2,
*/
```

5.5 Passing array to function

When array is passed to function, then pointer to first element is passed i.e. arrays are always passed by reference, as shown in Listing 5.4.

Explanation Listing 5.4:

> In the listing, array is passed to function 'addArray' (by using it's name i.e. 'a') at Line 23, which is similar to passing a normal variable to function. But the difference is in the prototype at Line 7, where 'int []' are used for array; also, we can use 'int *' to pass the array (instead of 'int []') . In the function (Lines 33-36), all the elements of array are increased by the value of variable 'addValue (Line 14). Note that return type for the function 'addArray' is 'void', but we are getting the new values of 'a' at Lines 26-27. This is happening, because arrays are passed by reference, hence all the changes are made directly on the addresses of the array.

Listing 5.4: Passing array to function

```
1   // arrayFuncEx.c
2   // arrays are always passed by reference
3
4   #include <stdio.h>
5
6   // use 'int []' or 'int *' to pass the array
7   void addArray(int [], int);
8
9   int main(){
10
11      // array 'a' is initialized with 3 elements;
12      // rest will be filled with zero.
13      int a[9] = {2, 4, 6}; // array size = 9 (i.e. 0 to 8)
14      int addValue = 4; // value to add with array-elements
```

(continues on next page)

(continued from previous page)

```

    printf("Initial values in array\n");
    for(int i=0; i<9; i++)
        printf("%d, ", a[i]);

    printf("\n\n"); // print two blank lines

    // arrays are always passed by reference
    addArray(a, addValue);

    printf("Final values in array\n");
    for(int i=0; i<9; i++)
        printf("%d, ", a[i]);

    return 0;
}

// arrays are always passed by reference
void addArray(int b[], int value){
    for (int i=0; i<9; i++)
        b[i] = b[i] + value;
}

/* Output
Initial values in array
2, 4, 6, 0, 0, 0, 0, 0, 0,

Final values in array
6, 8, 10, 4, 4, 4, 4, 4, 4,
*/
```

5.6 Using 'const' with pointer

In Listing 5.4, we pass the array to function and it's value is modified by the function. The '**const**' keyword can be used to pass the array to function in '**read only mode**'. In this case, the elements of the can be used inside the function, but can not be modified, as shown at Lines 7 and 22 of Listing 5.5. If we uncomment the Line 29, then error will be generated as values of array can not be modified. Similarly, variables and strings passed as 'const' can not be modified i.e. if we uncomment the Line 33, then error will be generated.

Listing 5.5: Passing parameters to function with 'const' keyword

```
1  // arrayConstFuncEx.c
2  // parameter passed as 'const' can not be modified
3  
4  #include <stdio.h>
5  
6  // use 'int []' or 'int *' to pass the array
7  void addArray(const int *, const int);
8  
9  int main(){
10 
11     // array 'a' is initialized with 3 elements;
12     int a[3] = {2, 4, 6}; // array size = 3 (i.e. 0 to 2)
13     int b = 2;
14 
15     // arrays are always passed by reference
16     addArray(a, b);
17 
```

(continues on next page)

(continued from previous page)

```c
    return 0;
}

// arrays are always passed by reference
void addArray(const int *a, const int b){
    int i;
    for (i=0; i<3; i++){
        printf("%d, ", a[i]);

        // below line will generate error as array is
        // passed with 'const' keyword.
        // a[i] = a[i] + 1;
    }

    // similarly b can not be modified
    // b = 3;
}

/* Output
2, 4, 6,
*/
```

5.7 Passing strings to function

Similar to array, the strings are also ‘passed by reference’ as shown in Listing 5.6. In the listing, char array ‘a’ is passed with ‘const’ keyword, therefore it’s value can not be modified by the function i.e. if we uncomment Line 30, then error will be generated.

Listing 5.6: Passing strings to function

```c
1   // stringConstFuncEx.c
2   // parameter passed as 'const' can not be modified
3
4   #include <stdio.h>
5
6   // use 'char []' or 'char *' to pass the string
7   void stringFunc(const char *, char []);
8
9   int main(){
10
11      char a[] = "String1"; // size = 8 = total elements (7) + Null character (1)
12      char b[] = "String2";
13
14      // strings are always passed by reference
15      printf("Size of a and b are %d and %d respectively\n", sizeof(a), sizeof(b));
16      stringFunc(a, b);
17
18      return 0;
19  }
20
21  // strings are always passed by reference
22  void stringFunc(const char *a, char *b){
23      int i;
24
25      for (i=0; i<7; i++){
26          printf("%c, ", a[i]);  // %s can not be used
27
28          // below line will generate error as string 'a' is
```

(continues on next page)

(continued from previous page)

```
29          // passed with 'const' keyword.
30          // a[i] = 'x';
31      }
32 
33      printf("\n");
34      for (i=0; i<7; i++){
35          printf("%c, ", b[i]);
36 
37          // double quote can not be used i.e. "x" is invalid
38          b[i] = 'x';
39      }
40 
41      printf("\n");
42      printf("%s, ", b);
43 
44  }
45 
46  /* Output
47  Size of a and b are 8 and 8 respectively
48  S, t, r, i, n, g, 1,
49  S, t, r, i, n, g, 2,
50  xxxxxxx,
51  */
```

5.8 Problem with string initialization and solution

Listing 5.7 is same as Listing 5.6, but size of character array is defined at Line 12 and 13. Now, look at the output of Line 39 at Line 51, where 'String1' is added after 'xxxx. . . ".

Note: Solution to this problem is shown in Listing 5.8, where **Null character** i.e. **\0** is used with 'for loop' at Lines 27 and 36 (Note the differences in these two lines as well). In this way, the loops execute for the 'length of the actual string i.e. 8' (instead of SIZE = 10); therefore does not store the unexpected result. Please read the comment at Line 12 as well.

Listing 5.7: Problem with string initialization

```
1   // stringConstFuncEx.c
2   // parameter passed as 'const' can not be modified
3 
4   #include <stdio.h>
5   const int SIZE = 10;
6 
7   // use 'char []' or 'char *' to pass the string
8   void stringFunc(const char *, char []);
9 
10  int main(){
11 
12      char a[SIZE] = "String1";
13      char b[SIZE] = "String2";
14 
15      // strings are always passed by reference
16      printf("Size of a and b are %d and %d respectively\n", sizeof(a), sizeof(b));
17      stringFunc(a, b);
18 
19      return 0;
20  }
```

(continues on next page)

(continued from previous page)

```c

// strings are always passed by reference
void stringFunc(const char *a, char *b){
    int i;

    for (i=0; i<SIZE; i++){
        printf("%c, ", a[i]);  // %s can not be used

        // below line will generate error as string 'a' is
        // passed with 'const' keyword.
        // a[i] = 'x';
    }

    printf("\n");
    for (i=0; i<SIZE; i++){
        printf("%c, ", b[i]);

        // double quote can not be used i.e. "x" is invalid
        b[i] = 'x';
    }

    printf("\n");
    printf("%s, ", b);

}

/* Output
Size of a and b are 10 and 10 respectively
S, t, r, i, n, g, 1,  ,  ,  ,
S, t, r, i, n, g, 2,  ,  ,  ,
xxxxxxxxxxString1,                    // unexpected result
*/
```

Listing 5.8: Solution of problem in Listing 5.7

```c
// stringInitSolution.c
// parameter passed as 'const' can not be modified

#include <stdio.h>
#define SIZE 10

// use 'char []' or 'char *' to pass the string
void stringFunc(const char *, char []);

int main(){

    // SIZE = 10, string-length = 8 = total elements (7) + Null character (1)
    char a[SIZE] = "String1";
    char b[SIZE] = "String2";

    // strings are always passed by reference
    printf("Size of a and b are %d and %d respectively\n", sizeof(a), sizeof(b));
    stringFunc(a, b);

    return 0;
}

// strings are always passed by reference
void stringFunc(const char *a, char *b){
    int i;

```

(continues on next page)

(continued from previous page)

```
    for (i=0; *a != '\0'; a++){
        printf("%c, ", a[i]);  // %s can not be used

        // below line will generate error as string 'a' is
        // passed with 'const' keyword.
        // a[i] = 'x';
    }

    printf("\n");
    for (i=0; b[i] != '\0'; i++){
        printf("%c, ", b[i]);

        // double quote can not be used i.e. "x" is invalid
        b[i] = 'x';
    }

    printf("\n");
    printf("%s, ", b);

}

/* Output
Size of a and b are 10 and 10 respectively
S, t, r, i, n, g, 1,
S, t, r, i, n, g, 2,
xxxxxxx,
*/
```

5.9 Conclusion

In this section, we saw that the name of the array is a pointer. Also, we defined multidimensional array. Further, it is shown that the arrays and strings are passed-by-reference in the functions. Lastly, use of 'const' keyword is discussed for passing variables, strings and arrays to functions.

Possessions and power are sought for the fulfillment of desires. Man is only partially satisfied in his attempt to have the fulfillment of his desires. And this partial satisfaction fans and increases the flame of craving instead of extinguishing it. So greed always finds an endless field of conquest, and leaves the man endlessly dissatisfied.

*–**Meher Baba***

Chapter 6

Structures

6.1 Introduction

Arrays are collection of data of similar type, whereas the Structures are the collections of variables, possibly of different data type. It is quite useful as data usually contains different types of elements. For example, radio can be specified using 'company name (string)', price ('float') and id ('int'); in such cases, structure can be used. In this chapter, we will see various examples of structures.

6.2 Structure

Structure is defined using keyword 'struct'. Elements of structure are called 'members'. In Listing 6.1, a structure with name 'item' is defined (lines 7-12) with three members i.e. 'id', 'company' and 'cost' (Line 9-11). Then, two variables of this structure are defined at Lines 14-15 i.e. 'radio' and 'oven'. Further, variable 'oven' is initialized as well; note that these values are assigned to members based on position e.g. '101', 'Sony' and '99.99' at Line 14, will be assigned to 'id', 'company' and 'cost' respectively. Lastly, the member functions of these variables can be accessed using '.' operator; for example Lines 18-20 print the values of 'radio'; whereas at Lines 22-23, the values are assigned to different members of variable 'oven'.

Listing 6.1: Structure example

```
1  // structEx.c
2  // create and access structure
3
4  #include <stdio.h>
5
6  int main(){
7      struct item
8      {
9          int id;
10         const char *company;  // char[30] will not work  at Line 23 for oven.company
11         float cost;
12     }; // semicolon at the end
13
14     struct item radio = {101, "Sony", 99.99};  // variable radio of type `item'
15     struct item oven; // variable oven of type `item'
16
17     printf("radio details: \n");
18     printf("id = %d\n", radio.id); // printing value using '.' operator
19     printf("company = %s\n", radio.company);
20     printf("price = %.2f\n", radio.cost);
21
22     oven.id = 102;
```

(continues on next page)

(continued from previous page)

```c
    oven.company = "Panasonic"; // assiging value using '.' operator
    printf("oven company: %s\n", oven.company);
}

/* Outputs
radio details:
id = 101
company = Sony
price = 99.99
oven company: Panasonic  */
```

Also, we can define structure-variables along with the structure-definitions as shown in Listing 6.2. Here, structure variables (radio and oven) are defined at Line 12, i.e. with the structure definition 'item'. Rest of the working of this listing is same as Listing 6.1.

Listing 6.2: Structure variable with definition

```c
1  // structEx2.c
2  // create and access structure
3  
4  #include <stdio.h>
5  
6  int main(){
7      struct item
8      {
9          int id;
10         const char *company;
11         float cost;
12     } radio = {101, "Sony", 99.99}, oven; // Define variable with definition
13 
14     printf("radio details: \n");
15     printf("id = %d\n", radio.id); // printing value using '.' operator
16     printf("company = %s\n", radio.company);
17     printf("price = %.2f\n", radio.cost);
18 
19     oven.id = 102;
20     oven.company = "Panasonic"; // assiging value using '.' operator
21     printf("oven company: %s\n", oven.company);
22 }
23 
24 /* Outputs
25 radio Details:
26 id = 101
27 company = Sony
28 price = 99.99
29 oven company: Panasonic  */
```

6.3 Defining structure outside the main function

It is better to define structure outside the main function for the purpose of clarity of the code, as shown in Listing 6.3. Note that, no memory is assigned to structure definition (i.e. Lines 6-11); memories are assigned to structure variables only (i.e. Line 15). Note that, structure is define outside the main function, but it **does not mean** that the structure-variables are global variable; these variables are still automatic variable and available within their scope.

Listing 6.3: Structure variable outside the main function

```c
// structMemoryEx.c
// pointer-variable of struct

#include <stdio.h>

struct item
{
    int id;
    const char *company;
    float cost;
}; // semicolon at the end

int main(){

    struct item radio = {101, "Sony", 99.99};  // variable radio of type `item'

    printf("id = %d\n", radio.id); // printing value using '.' operator
    printf("company = %s\n", radio.company);
}

/* Outputs
id = 101
company = Sony  */
```

6.4 Arrays of structure

Listing 6.4 creates the ‘array of structure’ at Line 13, where array of size 3 is created. Further, each member of these arrays can be accessed by ‘.’ operator along with index e.g. a[0], as shown in Lines 15-16.

Listing 6.4: Array of structure

```c
1  // structArrayEx.cpp
2  // array of structure
3  #include <stdio.h>
4
5  int main(){
6      struct item
7      {
8          int id;
9          const char *company;
10         float cost;
11     }; // semicolon at the end
12
13     struct item a[3];  // `a' is the array of structure `item'
14
15     a[0].id = 102;
16     a[0].company = "Panasonic"; // assiging value using '.' operator
17     printf("a[0] company: %s", a[0].company);
18 }
19
20 /* Outputs
21 a[0] company: Panasonic */
```

6.5 Pointer-variable of structure

Pointer variable can be created for a structure, and to access this variable '->' operator is used instead of '.' operator, as shown in Line 21, 26 and 28 of Listing 6.5. Here, Line 21 and 28 print the values which are stored at the address of pointer-variable i.e. 'radioPointer', whereas Line 26 is used to modify the content using pointer variable.

Listing 6.5: Pointer-variable of structure

```
1  // structPointerEx.c
2  // pointer-variable of struct
3  
4  #include <stdio.h>
5  
6  int main(){
7      struct item
8      {
9          int id;
10         const char *company;
11         float cost;
12     }; // semicolon at the end
13 
14     struct item radio = {101, "Sony", 99.99};  // variable radio of type `item'
15     struct item *radioPointer; // pointer variable of type 'item'
16 
17     radioPointer = &radio; // assign address of variable 'radio' to pointer
18 
19     printf("id using variable = %d\n", radio.id); // printing value using '.' operator
20     printf("id using pointer-variable = %d\n",
21                 radioPointer->id); // printing value using '->' operator
22 
23     printf("initial company name using variable = %s\n",
24                 radio.company); // printing value using '.' operator
25 
26     radioPointer->company = "Panasonic";  // modify company name using pointer
27     printf("modified company name using pointer variable = %s\n",
28             radioPointer->company); // printing value using '->' operator
29 
30 }
31 
32 /* Outputs
33 id using variable = 101
34 id using pointer-variable = 101
35 
36 initial company name using variable = Sony
37 modified company name using pointer variable = Panasonic */
```

6.6 Passing structure to function

Structure can be passed by values or by reference. When structure is passed by value, then only one value can be returned (or modified); whereas the pass by reference method can modify the complete structure using one function, as shown in Listing 6.6. Working of the listing is same as normal functions, the only difference is in the function-prototypes and definition , where 'struct item' are used i.e. to create the prototype (Line 15 and 16) and function-definition (Lines 32 and 37), we need to use keyword 'struct' and the name of the structure i.e. 'item'.

Warning: Note that, if we define the structure inside the main function, then it can not be passed to the functions; i.e. Lines 7-12 of Listing 6.6 can not be defined inside the main function, as it will generate error.

Listing 6.6: Passing structure to function

```
1  // structFuncEx.c
2  // pointer-variable of struct
3
4  #include <stdio.h>
5
6  // define struct at the top, then typedef and then function prototype
7  struct item
8  {
9      int id;
10     const char *company;
11     float cost;
12 }; // semicolon at the end
13
14 // function prototype
15 int structValueFucn(struct item); // pass by value
16 void structRefFucn(struct item *);  // pass by reference
17
18 int main(){
19     struct item radio, oven; // variable radio and oven of type `item'
20
21     // passing structure by value
22     radio.id = structValueFucn(radio);
23     printf("radio id = %d\n\n", radio.id);
24
25     // passing structure by reference
26     structRefFucn(&oven);
27     printf("oven id = %d\n", oven.id);
28     printf("oven company = %s\n", oven.company);
29     printf("oven cost = %f", oven.cost);
30 }
31
32 int structValueFucn(struct item i){
33     i.id = 111;
34     return i.id;
35 }
36
37 void structRefFucn(struct item *i){
38     i->id = 112;
39     i->company = "Sony";
40     i->cost = 99.99;
41 }
42
43 /* Outputs
44 radio id = 111
45
46 oven id = 112
47 oven company = Sony
48 oven cost = 99.989998  */
```

6.7 typedef

The keyword ‘typedef’ is used to create the ‘synonym (or alias)’ of a data type. For example in Listing 6.6, ‘struct item’ is used at Lines 15 and 16; this name can be shorten to ‘Item (or some other name)’ using ‘typedef command as shown in Listing 6.7. Here, alias is created for structure at Line 13; and then in the function-prototype (Line 15), ‘Item’ is used, instead of ‘struct item’.

Note: To avoid errors, define ‘constants’ at the top, next structures, then typedef, next function-prototypes and finally write the function definitions.

Listing 6.7: typedef example

```
1  // typedef.c
2  // typedef to create alias
3  
4  #include <stdio.h>
5  
6  // define struct at the top, then typedef and then function prototype
7  struct item
8  {
9      int id;
10     const char *company;
11     float cost;
12 }; // semicolon at the end
13 typedef struct item Item; // short name of 'struct item' = Item
14 
15 void structRefFucn(Item *);  // pass by reference
16 
17 int main(){
18     struct item oven;
19 
20     // passing structure by reference
21     structRefFucn(&oven);
22     printf("oven id = %d\n", oven.id);
23     printf("oven company = %s\n", oven.company);
24     printf("oven cost = %f", oven.cost);
25 }
26 
27 void structRefFucn(struct item *i){
28     i->id = 112;
29     i->company = "Sony";
30     i->cost = 99.99;
31 }
32 
33 /* Outputs
34 oven id = 112
35 oven company = Sony
36 oven cost = 99.989998 */
```

6.8 Define structure using typedef

The structure can be defined using 'typedef' as well. In this way, we need not to create the structure variable using 'struct' keyword. The keyword 'typedef' can be used in two different ways for creating the alias for the structures, as shown in this section. In :numref:sec_aliasSameName', the 'struct item' is renamed as 'item', i.e. names are same; whereas in :numref:sec_aliasDiffName', 'struct item' is renamed as 'Item' (Uppercase 'I') i.e. names are different.

6.8.1 Rename 'struct item' to 'item'

In Listing 6.7, the variable 'oven' of structure 'item' is created using '**struct item oven**' statement at Line 19; but with the help of 'typedef' we can create it using '**item oven**' statement. Further, we can omit the 'typedef' at Line 13 of Listing 6.7, as shown in Listing 6.8, where Line 13 is commented. Here, 'typedef' is used to define structure at Line 7; **note that, the name of the structure is now at the end of the structure definition i.e. at Line 12 (instead of after the keyword 'struct' at Line 7)**.

Listing 6.8: Rename ‘struct item’ to ‘item’

```c
// typedefEx2.c
// typedef to create structure

#include <stdio.h>

// define struct at the top, then typedef and then function prototype
typedef struct
{
    int id;
    const char *company;
    float cost;
} item; // semicolon at the end
// typedef struct item Item; // no need of this line now...

void structRefFucn(item *);  // pass by reference

int main(){
    item oven;

    // passing structure by reference
    structRefFucn(&oven);
    printf("oven id = %d\n", oven.id);
    printf("oven company = %s\n", oven.company);
    printf("oven cost = %f", oven.cost);
}

void structRefFucn(item *i){
    i->id = 112;
    i->company = "Sony";
    i->cost = 99.99;
}

/* Outputs
oven id = 112
oven company = Sony
oven cost = 99.989998 */
```

6.8.2 Rename ‘struct item’ to ‘Item’

For providing the different name to structure, we have to provide the name of the structure with ‘typedef’ i.e. ‘typedef struct item’ is added at Line 7 of Listing 6.9; and alternative name i.e. ‘Item’ should be added before the semicolon (Line 12). Now we can use ‘Item’ instead of ‘struct item’ as shown in Lines 15, 18 and 27.

Listing 6.9: Rename ‘struct item’ to ‘Item’

```c
1  // typedefEx3.c
2  // typedef to create structure
3
4  #include <stdio.h>
5
6  // define struct at the top, then typedef and then function prototype
7  typedef struct item
8  {
9      int id;
10     const char *company;
11     float cost;
12 } Item; // semicolon at the end
13 // typedef struct item Item; // no need of this line now...
```

(continues on next page)

(continued from previous page)

```

void structRefFucn(Item *);  // pass by reference

int main(){
    Item oven;

    // passing structure by reference
    structRefFucn(&oven);
    printf("oven id = %d\n", oven.id);
    printf("oven company = %s\n", oven.company);
    printf("oven cost = %f", oven.cost);
}

void structRefFucn(Item *i){
    i->id = 112;
    i->company = "Sony";
    i->cost = 99.99;
}

/* Outputs
oven id = 112
oven company = Sony
oven cost = 99.989998 */
```

6.9 Conclusion

In this chapter, structure and ‘array of structure’ are created. Then, we learn to pass the structure to the function using ‘pass by value’ and ‘pass by reference’ methods. Finally, we used ‘typedef’ to provide the alternate name to the structure.

%Note that various topics, such as ‘pointer as structure member’, ‘structure as structure member’ and ‘passing structure to function’ etc. are not discussed here, as we will not need these in the embedded design.

But I say to you who listen: Love your enemies, do what is good to those who hate you, bless those who curse you, pray for those who mistreat you.

–Jesus Christ

Chapter 7

File operations in C

7.1 Introduction

In previous chapters, we used variable, arrays and structures to store the data; but such data do not store permanently and lost after the termination of the program. Data can be stored to file using keyword ‘fwrite’; and can be read from the file using keyword ‘fscanf’, as shown in this chapter.

7.2 Write data to file

For writing or reading data from a file, we need to create one ‘FILE pointer’ as shown in Line 10 of Listing 7.1. Next, we need to open the file, which is done at Line 13; this line prints the message at Line 14, if the file can not be open. Note that, the file ‘data.txt’ file is saved inside the ‘data’ folder, therefore we need to create the ‘data’ folder first, as compiler can not create the folder. If file is open successfully, the compiler will reach to ‘else’ block at Line 16; where it will read the data from the terminal (Line 19), and then save it to file using ‘fprintf’ command at Line 22. The ‘while loop’ is used at Line 21, which reads the data from terminal until ‘end of file command’ is given i.e. ‘ctrl-Z’ or ‘crtl-C’, as shown in Fig. 7.1. The data saved by the listing is shown in Fig. 7.2.

Listing 7.1: Write data to file

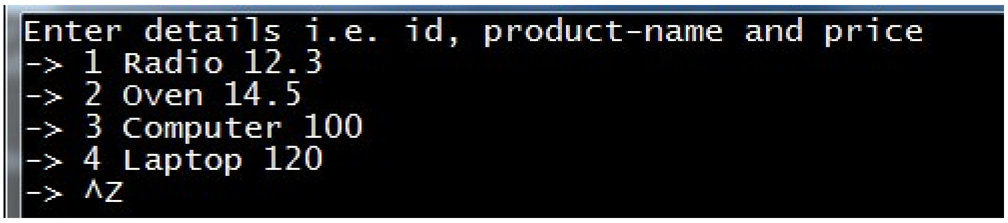
```
1  // writeDataEx.c
2
3  #include <stdio.h>
4
5  int main (void){
6      int id;
7      char product[30];
8      float price;
9
10     FILE *fPtr; // file pointer
11
12     // create 'data' folder first...
13     if ((fPtr = fopen ("data/data.txt", "w")) == NULL){  // "w" = write mode
14         printf("File can not be open\n");
15     }
16     else {
17         printf("Enter details i.e. id, product-name and price\n");
18         printf("-> ");
19         scanf("%d %s %f", &id, product, &price);
20
21         while(!feof(stdin)){
22             fprintf(fPtr, "%d %s %f\n", id, product, price); // write to file
23             printf("-> ");
```

(continues on next page)

(continued from previous page)

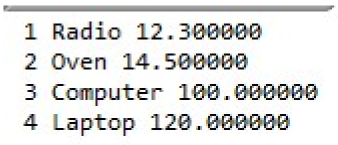
```c
            scanf("%d %s %f", &id, product, &price);
        }
    }

    fclose (fPtr);
    return 0;
}
```



```
Enter details i.e. id, product-name and price
-> 1 Radio 12.3
-> 2 Oven 14.5
-> 3 Computer 100
-> 4 Laptop 120
-> ^Z
```

Fig. 7.1: Reading data from terminal



```
1 Radio 12.300000
2 Oven 14.500000
3 Computer 100.000000
4 Laptop 120.000000
```

Fig. 7.2: Data in 'data.txt' file

7.3 Read data from file

Reading operation is similar to writing operation. The 'fscanf' command is used to read the data from the file, as shown in Line 16 and 20 of Listing 7.2.

Listing 7.2: Read data from file

```c
1  // readDataEx.c
2
3  #include <stdio.h>
4
5  int main (void){
6      int id;
7      char product[30];
8      float price;
9
10     FILE *fPtr; // file pointer
11
12     if ((fPtr = fopen ("data/data.txt", "r")) == NULL){ // "r" = read mode
13         printf("File can not be open\n");
14     }
15     else {
16         fscanf(fPtr, "%d %s %f", &id, product, &price);
17
18         while(!feof(fPtr)){
19             printf("%5d %10s %5.2f\n", id, product, price);
20             fscanf(fPtr, "%d %s %f", &id, product, &price); // read from file
21         }
22     }
23
24     fclose (fPtr);
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

7.4 Conclusion

In this chapter, we learn to read and write data to file. We can permanently store the data to file using ‘fwrite command’; and then this data can be processed and analyzed by some other software like Python and R etc.

The only way of not being upset by the blame is to be detached from the praise also; it is only through complete detachment that a person can keep unmoved by the opposites of praise and blame.

–Meher Baba

Chapter 8

Separating codes in multiple files

8.1 Introduction

In this chapter, we will learn to split the program in multiple files. In this way, the code-files are more manageable.

8.2 Program to separate

Listing 8.1 is the code, which we want to write in separate files. In this code, value of constant PI is printed at Line 18. Loops at Line 20 calculate the sum of all elements of array 'global_array'. Also, two functions are defined i.e. 'add2Num' and 'diff2Num' to calculate the sum and difference of two numbers.

Listing 8.1: Program to separate in multiple files

```
1  // main-to-split.c
2
3  #include <stdio.h>
4
5  #define PI 3.14   // constant
6  int global_array[] = {2, 4, 6};  // global array
7
8  void add2Num(int *, int *, int *); // pass by reference
9   int diff2Num(int, int); // pass by value
10
11 int main(){
12
13     int sum_array; // store sum of elements in global_array[]
14
15     int num1=4, num2=1;
16     int sum, diff;
17
18     printf("value of PI = %f\n", PI);
19
20     for (int i=0; i<3; i++)  // loop to find sum of global_array
21         sum_array += global_array[i];
22     printf("sum_array = %d\n", sum_array);
23
24     // add two numbers
25     add2Num(&num1, &num2, &sum); // pass by reference
26     printf("sum = %d\n", sum);
27
28     // subtract two numbers
29     diff = diff2Num(num1, num2); // pass by value
```

(continues on next page)

(continued from previous page)

```
    printf("difference = %d\n", diff);
}

// pass by reference
void add2Num (int *a, int *b, int *c){
    *c = *a + *b;
}

// pass by value
int diff2Num(int a, int b){
    return (a-b);
}
```

8.3 Separating the main function

In this section, we will write the main function in one file (Listing 8.2), constants in second file (Listing 8.3) and rest of the code in the third file (Listing 8.4). Since, we have splited the code in two different files, therefore we need to include these files in the main project as shown in Lines 5-6 of Listing 8.2.

Important: Note that, each file contains 'stdio.h' at the top which is enclosed between <...>, whereas files e.g. 'my_header.h' is enclosed between "...".

When we use "...", then compiler look for the file in the current project director, whereas <...> looks for the files in the standard library locations.

Listing 8.2: Program to separate in multiple files

```
1   // main1.c
2
3   #include <stdio.h>
4   #include "my_constants.c"
5   #include "my_header.h"
6
7   int main(){
8
9       int sum_array; // store sum of elements in global_array[]
10
11      int num1=4, num2=1;
12      int sum, diff;
13
14      printf("value of PI = %f\n", PI);
15
16      for (int i =0; i<3; i++)
17          sum_array += global_array[i];
18      printf("sum_array = %d\n", sum_array);
19
20      add2Num(&num1, &num2, &sum); // pass by reference
21      printf("sum = %d\n", sum);
22
23      diff = diff2Num(num1, num2); // pass by value
24      printf("difference = %d\n", diff);
25  }
```

Listing 8.3: Contants and global variable in 'my_constants.c' file

```c
// my_constants.h

#include <stdio.h>

#define PI 3.14
int global_array[] = {2, 4, 6};
```

Listing 8.4: Functions prototype and definition

```c
// my_header.h

#include <stdio.h>

void add2Num(int *, int *, int *); // pass by reference
int diff2Num(int, int); // pass by value

 void add2Num (int *a, int *b, int *c){
    *c = *a + *b;
}

int diff2Num(int a, int b){
    return (a-b);
}
```

8.4 Separating function-prototype and function-definition

In this section, we will further split the file 'my_header.h' in Listing 8.4, as shown in Listing 8.5 and Listing 8.6. Following two points are important here,

- If we uncomment the Line 4 of Listing 8.6, then we can compile the Listing 8.7 by using command 'g++ main2.c -o out' as the 'my_func_def.c' file is included to main2.c via 'my_header2.h'; but this is not the case if we comment this Line.
- If Line 4 of Listing 8.6 is commented, then there is no way to compile it directly using command in above point. We need to separately compile it using 'g++ main2.c my_func_def.c -o out'. Now, the code will compile successfully.

Listing 8.5: Functions prototypes

```c
// my_func_def.h

#include <stdio.h>

void add2Num (int *a, int *b, int *c){
    *c = *a + *b;
}

int diff2Num(int a, int b){
    return (a-b);
}
```

Listing 8.6: Functions definition

```c
1  // my_header2.h
2
3  #include <stdio.h>
4  // #include "my_func_def.c"
5
```

(continues on next page)

(continued from previous page)

```c
void add2Num(int *, int *, int *); // pass by reference
int diff2Num(int, int); // pass by value
```

Listing 8.7: Main function

```c
// main2.c

#include <stdio.h>
#include "my_constants.c"
#include "my_header2.h"

int main(){

    int sum_array; // store sum of elements in global_array[]

    int num1=4, num2=1;
    int sum, diff;

    printf("value of PI = %f\n", PI);

    for (int i =0; i<3; i++)
        sum_array += global_array[i];
    printf("sum_array = %d\n", sum_array);

    add2Num(&num1, &num2, &sum); // pass by reference
    printf("sum = %d\n", sum);

    diff = diff2Num(num1, num2); // pass by value
    printf("difference = %d\n", diff);
}
```

8.5 Conclusion

In this chapter, we learn to separate the program into multiple files. Also, we saw that we need to compile multiple files, for the cases where function-definitions-file is not included to function-header-file (i.e. file that contains function prototypes).

The only way of not being upset by the blame is to be detached from the praise also; it is only through complete detachment that a person can keep unmoved by the opposites of praise and blame.

–Meher Baba

Chapter 9

Formatted Inputs and Outputs

9.1 Introduction

In Listing 5.1, we used '%4d' and '%21.5' etc. to print the outputs in readable format. In this chapter, we will some more commands for formatting input and outputs i.e. 'scanf' and 'printf' commands respectively.

9.2 Printing Integer and Floating point values

Table 9.1 shows the list of 'conversion specifier' e.g. 'd' and 'f' etc. for printing 'Integer' and 'Floating point' values. Further, these specifiers are used to print the values in different formats in Listing 9.1.

Table 9.1: Print numbers in various formats

Integer Format	**Description**
d or i	Signed decimal integer
o	Unsigned Octal integer
u	Unsigned decimal integer
x or X	Unsigned Hexadecimal integer
Floating Point Format	**Description**
f	Fixed notation (e.g. 102.300003)
e or E	Exponential notation (e.g. 1.023000e+002)
g or G	Display in format 'f' or 'e' based on value

Listing 9.1: Different formats Integer and Float values

```c
// printIntFloat.c

#include <stdio.h>

int main()
{
    int i = 10, j = -12;

    float x = 102.3, y=-23.49;

    printf("----- Integer format ----- \n");
    printf("%d, %i, %o, %u, %x \n", i, i, i, i, i);
    // hex and oct : -ve numbers are displayed as "2's complement"
    printf("%d, %i, %o, %x\n\n", j, j, j, j);

    printf("----- Floating point format ----- \n");
```

(continues on next page)

(continued from previous page)

```
    printf("%e, %f, %g \n", x, x, x);
    printf("%e, %f, %g \n", y, y, y);

    // hex and oct : -ve numbers are displayed as "2's complement"
    // printf("%d, %i, %o, %x ", y, y, y, y);

    return 0;
}

/* Outputs
----- Integer format -----
10, 10, 12, 10, a
-12, -12, 37777777764, fffffff4

----- Floating point format -----
1.023000e+002, 102.300003, 102.3
-2.349000e+001, -23.490000, -23.49

*/
```

9.3 Modify 'printf' format

Table 9.2 shows various flags which can be used to format the outputs for the 'printf' statement. Listing 9.2 and Listing 9.3 show the usage of these flags. Please read the comments under these listings.

Table 9.2: 'printf' formats

Flag	Description
+	Right justified
-	Left justified
#	Prefix 'o' and 'x' for octal and hex outputs respectively
space	Prints space before positive values
0	Pad leading zeros

Listing 9.2: 'Left & right justified' and 'signs'

```
// printfFlag.c

#include <stdio.h>

int main()
{
    int i1 = 210, i2=-15;
    float j1 = -123.34, j2=12.22;
    char c1[] = "Meher", c2[] = "Krishna";

    // 10d : 10 is the minimum width for integer
    // 10.4f : 10 and 4 are minimum width for integer part and float part respectively
    printf("---------- Right justified ---------- \n");
    printf("%10d %10.4f %10s\n", i1, j1, c1);
    printf("%10d %10.4f %10s\n\n", i2, j2, c2);

    // +10d to add `+ sign' with numeric values
    printf("%+10d %+10.4f %10s\n", i1, j1, c1);
    printf("%+10d %+10.4f %10s\n\n", i2, j2, c2);

    // 0 is used for paddign zeros
```

(continues on next page)

(continued from previous page)

```
    // 0 can not be used with 'left justified'
    printf("%010d %010.4f %10s\n", i1, j1, c1);
    printf("%010d %010.4f %10s\n\n", i2, j2, c2);

    // - for left justified
    printf("---------- Left justified ---------- \n");
    printf("%-10d %-10.4f %-10s\n", i1, j1, c1);
    printf("%-10d %-10.4f %-10s\n\n", i2, j2, c2);

    // - for left justified and then + to add `+ sign' with numeric values
    printf("%-+10d %-+10.4f %-10s\n", i1, j1, c1);
    printf("%-+10d %-+10.4f %-10s\n\n", i2, j2, c2);

    // - for left justified and then 'space' to add `space' for +ve values
    printf("%- 10d %- 10.4f %-10s\n", i1, j1, c1);
    printf("%- 10d %- 10.4f %-10s\n", i2, j2, c2);

    return 0;
}

/* Outputs
---------- Right justified ----------
       210  -123.3400      Meher
       -15    12.2200    Krishna

      +210  -123.3400      Meher
       -15   +12.2200    Krishna

0000000210 -0123.3400      Meher
-000000015 00012.2200    Krishna

---------- Left justified ----------
210        -123.3400  Meher
-15        12.2200    Krishna

+210       -123.3400  Meher
-15        +12.2200   Krishna

 210       -123.3400  Meher
-15         12.2200   Krishna
*/
```

Listing 9.3: '#' to print '0x' and '0' before hex and oct numbers respectively

```
// printfPrefix.c

#include <stdio.h>

int main()
{
    int i1=10;

    // # to print '0x' and '0' before hex and oct numbers respectively
    printf("---------- Right justified ---------- \n");
    printf("%10d %10x %10o\n", i1, i1, i1);
    printf("%10d %#10x %#10o\n\n", i1, i1, i1);

    printf("---------- Left justified ---------- \n");
    printf("%-10d %-10x %-10o\n", i1, i1, i1);
```

(continues on next page)

(continued from previous page)

```
    printf("%-10d %-#10x %-#10o\n\n", i1, i1, i1);

    return 0;
}

/* Outputs
---------- Right justified ----------
        10           a          12
        10         0xa         012

---------- Left justified ----------
10         a          12
10         0xa        012
*/
```

9.4 Formatted input using 'Scanf'

In this section, we will see various methods to get inputs from user with 'scanf' command.

9.4.1 Format specifiers

We can use all the formats specified in Table 9.1 with 'scanf' as well. The difference is in 'd' and 'i' specifiers i.e. 'd' can only read integer format, where 'i' can read 'Octal' and 'Hexadecimal' formats as well, as shown in Listing 9.4.

Listing 9.4: Difference in 'd' and 'i'

```
// scanfEx.c

#include <stdio.h>

int main()
{
    int a, b, c, d;
    unsigned int e; // for hexadecimal values (int will give warning)

    // %d can not read Oct and Hex values
    // %i can read Oct and Hex values
    printf("Enter 5 numbers numbers : \n");
    scanf("%d%i%i%i%x", &a, &b, &c, &d, &e);

    printf("Outputs\n");
    printf("%d %d %d %d %d", a, b, c, d, e);
    return 0;
}

/* Outputs
Enter 5 numbers numbers :
-12 -12 014 0xc 0xc         014 : Oct input; 0xc : Hex input
Outputs
-12 -12 12 12 12
*/
```

9.4.2 Scan and rejection set

We can select or reject data from the input provided by user, as shown in Listing 9.5 and Listing 9.6 respectively. The '[abc]' is used for creating the scan-set, whereas *âbc* is used for rejection-set. Please see comments in the listing for better understanding.

Listing 9.5: Scan set

```c
// scanSetEx.c

#include <stdio.h>

int main()
{
    char a[20];

    printf("Write Yes, No, Yyes, YNo : ");
    // read the string starts from 'y, Y, n or N'
    // and select string till only these letters appear
    scanf("%[yYnN]", a);

    printf("Your answer is : %s\n", a);
    return 0;
}

/* Outputs
Write Yes, No, Yyes, YNo :  Yes
Your answer is : Y

Write Yes, No, Yyes, YNo : YNo
Your answer is : YN
*/
```

Listing 9.6: Inverted scan set (reject set)

```c
// invertedScanSetEx.c

#include <stdio.h>

int main()
{
    char a[20];

    printf("Data will be read till Y or Y appears \n");
    printf("Enter string :  ");
    // read the string till  'y or Y' appear
    scanf("%[^yY]", a);

    printf("You entered : %s\n", a);
    return 0;
}

/* Outputs
Data will be read till Y or Y appears
Enter string :  Meher Krishna y
You entered : Meher Krishna

Data will be read till Y or Y appears
Enter string :  Meher Krishna y Bye Bye
You entered : Meher Krishna
*/
```

9.4.3 Suppression character '*'

Suppression character can be used for removing certain characters while reading input e.g. while reading date in 'dd-mm-yyyy', we can skip the '-' as shown in Listing 9.7, where two methods are shown for suppressing the characters.

Listing 9.7: Suppression character

```c
// suppressCharEx.c

#include <stdio.h>

int main()
{
    int date, month, year;

    // define specific skip character
    printf("Enter data in dd-mm+yyyy format : ");
    scanf("%d-%d+%d", &date, &month, &year); // skip ' - ' and +
    printf("Date = %d, Month = %d, Year = %d\n\n", date, month, year);

    // define general skip character
    printf("Enter data in dd-mm-yyyy format : ");
    // %*c read character and skip it
    scanf("%d%*c%d%*c%d", &date, &month, &year);
    printf("Date = %d, Month = %d, Year = %d\n", date, month, year);


    return 0;
}

/* Outputs
Enter data in dd-mm+yyyy format : 12-12+2012
Date = 12, Month = 12, Year = 2012

Enter data in dd-mm-yyyy format : 12*12/2012
Date = 12, Month = 12, Year = 2012

*/
```

9.5 Conclusion

In this section, we saw various methods to format the input and output data using 'scanf' and 'printf' command respectively.

When a man is, through his desires, confronted with great suffering , he understands their true nature; so when such suffering comes, it should be welcome. Suffering may come in order to eliminate further suffering. Suffering has to come, when it is of use in purging the soul from it's desires.

–Meher Baba

Chapter 10

More Examples

10.1 Introduction

In previous chapters, we saw various useful features of C language with some examples. In this chapter, more examples are added for better understanding of the language. Only new and/or important parts of the codes are discussed in this chapter.

10.2 Basic Examples

In this section, some simple examples are shown to learn the language effectively.

10.2.1 Print number in reverse order

The 'for loop' can be executed in reverse direction using 'i–' operator as shown at Line 8 of Listing 10.1.

Listing 10.1: Print number in reverse order

```
 1  // reverseOrder.c
 2
 3  #include <stdio.h>
 4
 5  int main(){
 6      int i;
 7
 8      for (i=10; i>0; i--)  // loop in reverse direction
 9          printf("%d, ", i);
10      return 0;
11  }
12
13  /* Outputs
14  10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
15  */
```

10.2.2 Factorial

Listing 10.2 calculates the factorial of numbers. In this listing, 'factorial *= i' is the sort notation for 'factorial = factorial * i'. Similarly, we can use 'a += 3' for 'a = a + 3' etc.

Listing 10.2: Factorial

```c
// factorialEx.c

#include <stdio.h>

int main(){
    int i, num, factorial = 1, flag = 0;

    printf("Enter the number : ");
    scanf("%d", &num);

    if (num <= 0){
        flag = 1;
        printf("Number should be greater than 0.");
    }
    else{
        for (i=1; i<=num; i++)
            factorial *= i;  // factorial = factorial * i;
    }

    if (flag != 1)
        printf("Factorial = %d",  factorial);

    return 0;
}

/* Outputs
Enter the number : 4
Factorial = 24
*/
```

10.2.3 Fibonacci series

Listing 10.3 prints the Fibonacci series. In this listing, ‘infinite for loop’ is used at Line 15, which is terminated at Line 24, when the next Fibonacci number is greater than the ‘limit’ provided by the user at Lines 10-11.

Listing 10.3: Fibonacci series

```c
 1 // Fibonacci.c
 2 
 3 #include <stdio.h>
 4 
 5 int main(){
 6     int limit;
 7     int num1 = 0, num2 = 1, temp;
 8 
 9     // print Fibonacci series upto limit
10     printf("Enter the limit for series : ");
11     scanf("%d", &limit);
12 
13     // print Fibonacci series
14     printf("%d, %d, ", num1, num2);
15     for (;;){        // infinite loop
16         temp = num1;
17         num1 = num2;
18         num2 = num1 + temp;
19 
20         if (num2 <= limit)
21             printf("%d, ", num2);
```

(continues on next page)

(continued from previous page)

```
        else
            break;
    }
    return 0;
}

/* Outputs
Enter the limit for series : 50
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
*/
```

10.2.4 Print values of characters e.g. EOF, Space and Enter etc.

'Spaces', 'EOF (End of file)', 'Commas' and 'Enter (i.e. line-change) can be considered as the end of data e.g. in CSV file, the comma is the delimiter which distinguish one data from other. Therefore, we need to identity these values while working with data. Further, The symbolic constant 'EOF (End of file)' is an integer and defined in 'stdio.h'. The '**getchar**' command returns different values for different characters.

Explanation Listing 10.4

In the listing, Line 8 prints the value 'EOF' i.e. -1. Next , 'getchar' command is used (Line 14) to read the values from terminal and then corresponding 'integer' values are printed using 'printf' command. Here, infinite-while-loop is used to read the values from the terminal (Line 13) and 'E (uppercase E)' is used to break the loop. **Note that, 'int c' is used (not 'char c'), to store the larger values and 'EOF', which is not possible with 'char c'**. In the outputs, we can see the different values for different characters; some of those values are listed below,

- t = 116
- space = 32
- enter = 10
- EOF (i.e. ctrl+z) = -1

Listing 10.4: Print values of characters e.g. EOF, Space and Enter etc.

```
1  // EOF_ex.c
2
3  #include <stdio.h>
4
5  int main(void){
6      int c;
7
8      printf("EOF = %d\n", EOF);  // EOF = -1
9
10     printf("Enter characters\n");
11
12     // ctrl+z = -1, Enter = 10; space = 32
13     while (1){
14         c = getchar();
15         printf("%c = %d\n", c, c);
16
17         if (c == 'E') // type 'E' to exit the program
18           break;
19      }
20      return 0;
21 }
22
23 /* Outputs
24 EOF = -1
```

(continues on next page)

(continued from previous page)

```

Enter characters

tit tat       // below are the values of individual character
t = 116
i = 105
t = 116
  = 32         // space
t = 116
a = 97
t = 116
              // enter
 = 10

this that
t = 116
h = 104
i = 105
s = 115
  = 32
t = 116
h = 104
a = 97
t = 116

 = 10

^Z             // ctrl+z = end of file
  = -1

E              // terminate the execution
E = 69
*/
```

10.2.5 Read and write until EOF

In Section 10.2.4, we saw that 'EOF' is the constant, which is stored in the 'stdio.h' header file; and infinite loop was terminated with the help of character 'E' in 'if loop'. Now, In Listing 10.5, we will use the 'EOF' to terminate the program. Also, '**putchar**' command is used (Line 13) to print the character, instead of 'printf'.

Listing 10.5: read and write until EOF

```
1  // getPutEOF.c
2
3  #include <stdio.h>
4
5  int main(void){
6      int c;
7
8      printf("Enter characters\n");
9
10     c = getchar();
11
12     while (c != EOF){  // read and write till EOF
13         putchar(c);   // print the characters stored in 'c'
14         c = getchar();  // read next character
15       }
16
17     return 0;
```

(continues on next page)

(continued from previous page)

```
}

/* Outputs
Enter characters
Meher       // read from terminal
Meher       // printed by 'putchar'

Krishna
Krishna
^Z
*/
```

10.2.6 Counting characters

Listing 10.6, counts the total number of characters entered before EOF operation. Here, 'getchar()' is used inside the 'while loop' i.e. character will be read from the terminal until the compiler gets the 'ctrl+Z' command; also, we do no not need 'char c' for storing the character, as we did in Listing 10.5.

Listing 10.6: Counting characters

```c
// countChar.c

#include <stdio.h>

int main(void){
  int count = 0;

  printf("Enter characters\n");

  while ( getchar() != EOF ){  // read and write till EOF
    count += 1;    // increase count by 1
  }

  printf("Total words = %d\n", count);

  return 0;
}

/* Outputs
Enter characters    // run 1
This                // 4 characters + 1 enter = 5
^Z
Total words = 5

Enter characters    // run 2
This
That
^Z
Total words = 10
^Z
*/
```

10.2.7 Counting spaces, words and lines

Listing 10.7 counts the spaces, words and spaces. Note that, single quotes (not double) are used to define space i.e. ' '. Also, word counts are based on space and enters, therefore 'word_count' is used in both the 'if statements' i.e. Lines 13 and 18.

Listing 10.7: Counting characters

```c
// countLine.c

#include <stdio.h>

int main(void){
  int c;
  int space_count = 0, word_count = 0, line_count = 0;

  printf("Enter characters\n");

  while ( (c = getchar()) != EOF ){  // read and write till EOF

    if (c == ' '){  // space
      space_count += 1;   // increase space_count by 1
      word_count += 1;    // increase word_count by 1
    }

    if (c == '\n'){  // Enter
        line_count += 1;    // increase line_count by 1
        word_count += 1;    // increase word_count by 1
    }
  }

  printf("Total spaces = %d\n", space_count);
  printf("Total words = %d\n", word_count);
  printf("Total lines = %d\n", line_count);

  return 0;
}

/* Outputs
  Enter characters
  This is cat.
  That is cow.
  ^Z
  Total spaces = 4
  Total word = 6
  Total lines = 2
*/
```

10.2.8 Print longest line

Listing 10.8 prints the longest line entered by the user. Here, function 'getLine' (Line 8), checks the length of the line; which is called by 'main' function through Line 37'. Note that, the condition for 'while loop' at Line 38 is '> 0'; since the 'EOF' command returns '0' therefore, the loop will be terminated by 'ctrl+z' command. Also, automatic-variable 'k' (Line 46) is local to 'if statement' at Line 39 (see comments for further details). Lastly, 'MAX_CHAR - 2' is used at Line 13, to store the 'NULL character i.e. \ 0' at the end of the string.

Listing 10.8: Longest line

```c
1  // largest_line.c
2  #include <stdio.h>
3
4  const int MAX_CHAR = 15;  // display only MAX_CHAR characters of largest line
5
6  int getLine(char []);  // function prototype
7
```

(continues on next page)

(continued from previous page)

```
 8 // store 'MAX_CHAR' to line if number of characters are greater than MAX_CHAR
 9 int getLine(char line[]){
10     int i, j = 0, c;
11     for (i = 0; (c = getchar()) != EOF && c != '\n'; i++){
12         // Null character ( \0) is special string character to terminate the string
13         if (i < MAX_CHAR - 2){  // -2 to store the last character as "Null character"
14             line[j] = c;
15             j++;
16         }
17 
18         line[j] = '\0';
19     }
20     return i;
21 
22 }
23 int main(){
24     int i = 1, j = 0;
25     // int k = 0;
26     int current_len;  // number of characters in current line
27     int max_length = 0;  // maximum numbers of characters till now
28 
29 
30     char line[MAX_CHAR];  // current input line with maximum character "MAX_CHAR"
31     char longest_line[MAX_CHAR];
32 
33     // exit loop if characters are above MAX_CHAR
34     // getline : returns the number of character in current Line
35     // use 'crtl + z' to exit the loop; as it returns '0'
36     printf("Line %d : ", i);
37     while ((current_len = getLine(line)) > 0){
38         printf("Length =  %2d\n", current_len);
39         if (max_length < current_len){
40             max_length = current_len;
41             j = i;
42 
43             // note that K is local to 'if' statement
44             // if it is defined at Line 25, then we need to reset the 'k'
45             // i.e. k = 0, after exiting the 'while loop' (Line 49)
46             int k=0;
47             while((longest_line[k] = line[k]) != '\0') //copy 'line' to 'longest_line'
48                 ++k;
49             // k = 0;
50         }
51 
52         printf("Line %d : ", ++i);
53     }
54 
55     printf("\nLargest line is %d, which has %d characters \n", j, max_length);
56     printf("Content = %s", longest_line);
57 
58     return 0;
59 }
60 
61 /* Outputs
62 ####################################
63 Line 1 : Krishna
64 Length =   7
65 Line 2 : Patel
66 Length =   5
67 Line 3 : ^Z
68 
```

(continues on next page)

(continued from previous page)

```
Largest line is 1, which has 7 characters
Content = Krishna
####################################

Line 1 : My name is Meher Krishna Patel
Length =  30
Line 2 : Meher
Length =   5
Line 3 : ^Z

Largest line is 1, which has 30 characters
Content = My name is Me       // only 15 characters (i.e. MAX_CHAR) are displayed
*/
```

10.2.9 Store and print values of array

In Listing 10.9, the array size is received from terminal (Line 10), and then values are stored in the array (Lines 14-15). Finally, these values are printed by Lines 20-21.

Listing 10.9: Store and print value of array

```
 1  // storeArray.c
 2
 3  #include <stdio.h>
 4
 5  int main(){
 6      int i, arr_size, arr[25];
 7
 8      // get array size
 9      printf("Enter the size of array (max = 25): ");
10      scanf("%d", &arr_size);
11
12      // get values for array
13      for (i=0; i<arr_size; i++){
14          printf("element [%d] ->  ", i);
15          scanf("%d", &arr[i]);
16      }
17
18      // print values in array
19      printf("You entered following values, \n");
20      for (i=0; i<arr_size; i++){
21          printf("%d, ", arr[i]);
22      }
23
24      return 0;
25  }
26  /* Outputs
27  Enter the size of array (max = 25): 4
28  element [0] ->  2
29  element [1] ->  3
30  element [2] ->  5
31  element [3] ->  8
32
33  You entered following values,
34  2, 3, 5, 8,
35  */
```

10.2.10 Minimum and maximum values of array

Lines 23-28 of Listing 10.10 find the minimum and maximum values of the array.

Listing 10.10: Minimum and maximum values of array

```
1  // small_large_array.c
2  
3  #include <stdio.h>
4  
5  int main(){
6      int i;
7      int arr_size, arr[25];
8      int min_value, max_value;
9  
10     // get array size
11     printf("Enter the size of array (max = 25): ");
12     scanf("%d", &arr_size);
13 
14     // get values for array
15     for (i=0; i<arr_size; i++){
16         printf("element [%d] ->  ", i);
17         scanf("%d", &arr[i]);
18     }
19 
20     min_value = max_value = arr[0];  // initialize values
21 
22     // find minimum and maximum value
23     for (i=0; i<arr_size; i++){
24         if (min_value > arr[i])   // if min_value > current value
25             min_value = arr[i];   // then replace value
26 
27         if (max_value < arr[i])   // if max_value < current value
28             max_value = arr[i];   // then replace value
29     }
30 
31     // print minimum and maximum value
32     printf("Minimum value = %d \nMaximum value = %d", min_value, max_value);
33 
34     return 0;
35 }
36 
37 /* Outputs
38 Enter the size of array (max = 25): 6
39 element [0] ->  3
40 element [1] ->  67          // maximum
41 element [2] ->  12
42 element [3] ->  1
43 element [4] ->  0
44 element [5] ->  -13         // minimum
45 
46 Minimum value = -13
47 Maximum value = 67
48 */
```

10.2.11 Find duplicate elements in array

Listing 10.11 check the duplicates in the array, and prints the location of duplicate elements.

Listing 10.11: Find duplicate elements in array

```c
// find_duplicate.c

#include <stdio.h>

int main(){
    int i, j, flag;
    int arr_size, arr[25];

    // get array size
    printf("Enter the size of array (max = 25): ");
    scanf("%d", &arr_size);

    // get values for array
    for (i=0; i<arr_size; i++){
        printf("element [%d] ->  ", i);
        scanf("%d", &arr[i]);
    }

    // find duplicates
    for (i=0; i<arr_size-1; i++){  // 0 to arr_size - 2
        for (j=i+1; j<arr_size; j++){ // i+1 to arr_size - 1
            if ( arr[i] == arr[j] ){
                flag = 1;               // duplicate found
                printf("Duplicate found at locations %d and %d\n", i, j);
            }
        }
    }

    // if no duplicates i.e. flag == 0
    if (flag != 1)
        printf("No duplicates found");

    return 0;
}

/* Outputs
Enter the size of array (max = 25): 5
element [0] ->  5
element [1] ->  3
element [2] ->  2
element [3] ->  5
element [4] ->  3

Duplicate found at locations 0 and 3
Duplicate found at locations 1 and 4
*/
```

10.2.12 Function to sort the numbers in array

In Listing 10.12, array is sorted by the function 'sortArray' at Line 8. This sorting algorithm is known as 'bubble sort'.

Listing 10.12: Sort the numbers in array

```c
// sortArray.c

#include <stdio.h>

```

(continues on next page)

(continued from previous page)

```
void sortArray(int [], int);  // function prototype

// function for sorting array : bubble sort
void sortArray(int a[], int size){
    int i, j, temp;
    for (i = 0; i < size; i++){
        for (j = 0; j < size-i-1; j++){
            if ( a[j] > a[j+1]){
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}

int main(){
    int i;
    int arr_size, arr[25];

    // get array size
    printf("Enter the size of array (max = 25): ");
    scanf("%d", &arr_size);

    // get values for array
    for (i=0; i<arr_size; i++){
        printf("element [%d] ->  ", i);
        scanf("%d", &arr[i]);
    }

    // function to sort array
    sortArray(arr, arr_size);

    // print the sorted array
    for (i=0; i<arr_size; i++)
        printf("%d, ", arr[i]);

    return 0;
}

/* Outputs
Enter the size of array (max = 25): 6
element [0] ->  32
element [1] ->  43
element [2] ->  23
element [3] ->  1
element [4] ->  32
element [5] ->  455
1, 23, 32, 32, 43, 455,
*/
```

10.3 Number conversion

In this section, the numbers are converted into different formats.

10.3.1 Binary to decimal conversion

Listing 10.13 converts the binary number to decimal number.

Listing 10.13: Binary to decimal conversion

```c
// Binary_to_Decimal.c

#include <stdio.h>

void Binary_to_Decimal(int *, int *);

// function to convert binary number to decimal
void Binary_to_Decimal(int *binary_val, int *decimal_val){
    /* usage :
            int binary_val, decimal_val;
            Binary_to_Decimal(&binary_val, &decimal_val);

            binary_val = binary input from user
            decimal_val = decimal output from the function
    */

    int num, flag = 0, rem, base = 1;
    *decimal_val = 0;

    num = *binary_val;
    while (num > 0){
        rem = num % 10;

        if ((rem != 0) && (rem != 1)){
            printf("Enter the correct binary number...\n");
            flag = 1;
            break;
        }

        *decimal_val = *decimal_val + rem * base;
        num = num / 10 ;
        base = base * 2;
    }

    if (flag == 0)
        printf("Decimal number = %d \n", *decimal_val);
}

int main(){

    int binary_val, decimal_val;

    printf("Enter a binary number i.e. 1 and 0 only :  ");
    scanf("%d", &binary_val); /* maximum five digits */

    Binary_to_Decimal(&binary_val, &decimal_val);

    return 0;
}

/* Outputs
Enter a binary number i.e. 1 and 0 only :  1000
Decimal number = 8
*/
```

10.3.2 Decimal to binary conversion

Listing 10.14 converts the decimal number to binary number.

Listing 10.14: Decimal to binary conversion

```c
// Decimal_to_binary.c

#include <stdio.h>

void Decimal_to_binary(int *, int *);

// // function to convert decimal number to binary
void Decimal_to_binary(int *decimal_val, int *binary_val){
//      usage :
//              int binary_val, decimal_val;
//              Decimal_to_binary(&decimal_val, &binary_val);

//              binary_val = binary input from user
//              decimal_val = decimal output from the function

    int num, rem, base = 1;

    num = *decimal_val;
    *binary_val = 0;

    while (num > 0){
        rem = num % 2;
        *binary_val = *binary_val + rem * base;
        num = num / 2;
        base = base * 10;
    }

    // printf("Input number is = %d\n", *decimal_val);
    printf("Binary number = %d\n", *binary_val);
}

int main(){

    int decimal_val, binary_val;

    printf("Enter a decimal integer : ");
    scanf("%d", &decimal_val);

    Decimal_to_binary(&decimal_val, &binary_val);

    return 0;
}

/* Outputs
Enter a decimal integer : 15
Binary number = 1111
*/
```

10.4 Matrices

In this section, various matrix operations are implemented using C.

10.4.1 Size of the matrix

Listing 10.15 checks the total number of elements in the array. Also, it shows the method to find the number of rows and columns in multidimensional arrays. **Note that arrays are bound for all dimensions except first,**

therefore it is compulsory to define the second dimension as shown in Line 8 of the listing.

Listing 10.15: Size of the matrix

```
 1  // elements_of_array.c
 2
 3  #include <stdio.h>
 4
 5  int main(){
 6
 7      int arr1[] = {2, 4, 6, 8};
 8      int arr2[][2] = {   // arrays are bound for all dimension except first
 9                  {2, 3},
10                  {4, 5},
11                  {5, 3}
12      };
13
14
15      int size_of_arr1, elements_arr1;
16      int size_of_arr2, elements_arr2, rows_arr2, col_arr2;
17
18
19      // sizeof = total size of variables in array
20      size_of_arr1 = sizeof(arr1);
21      printf("Size of arr1 : %d\n", size_of_arr1);
22
23      // length = (total size) / (size of each element)
24      elements_arr1 = sizeof(arr1)/sizeof(arr1[0]);
25      printf("Total elements in arr1 : %d\n\n", elements_arr1);
26
27
28      // sizeof = total size of variables in array
29      size_of_arr2 =  sizeof(arr2);
30      printf("Size of arr2: %d\n", size_of_arr2);
31
32      // length of arr2
33      elements_arr2 = sizeof(arr2)/sizeof(arr2[0][1]);
34      printf("Total elements in arr2 : %d\n", elements_arr2);
35
36      // number of rows
37      rows_arr2 = sizeof(arr2)/sizeof(arr2[0]);
38      printf("Number of rows in arr2 : %d\n", rows_arr2);
39
40      col_arr2 = elements_arr2 / rows_arr2;
41      printf("Number of columns in arr2 : %d\n", col_arr2);
42
43      return 0;
44  }
45
46  /* Outputs
47  Size of arr1 : 16
48  Total elements in arr1 : 4
49
50  Size of arr2: 24
51  Total elements in arr2 : 6
52  Number of rows in arr2 : 3
53  Number of columns in arr2 : 2
54  */
```

10.4.2 Dot product

Listing 10.17 calculates the dot product of two vectors i.e. $x^T y$, where $x, y \in R^n$ and R^n is the short notation for $R^{n \times 1}$. Dot product can be implemented using Listing 10.16. Following are the important points to notice in Listing 10.17,

- For dot product, the size of the vectors should be same.
- In Line 21, **'const int N'** is used (instead of 'int N'), because the variable 'N' is used for defining the size of array at Lines 22-23, which can not be of variable type.
- Since, 'N' is constant, therefore it can not be 'passed by reference' during function call; i.e. at Line 27, 'N' is 'passed by value', whereas variable 'result' is 'passed by reference'.

Listing 10.16: **Alogrithm** Dot Product : $x, y \in R^n$

```
for {i = 1:n} do
    result = result + x(i)y(i)
end for
```

Listing 10.17: Dot product

```
 1  // dot_product.c
 2  // matrices must be of same size for dot-product
 3
 4  #include <stdio.h>
 5
 6  void dot_product(int [][1], int [][1], int , int *);
 7
 8  // calculate dot product
 9  void dot_product(int x[][1], int y[][1], int N, int *result){
10
11      int i;
12      *result = 0;
13
14      for (i=0; i<N; i++){
15         *result += (x[i][0] * y[i][0]);
16      }
17  }
18
19  int main(){
20
21      const int N = 4;    // constant is required as we are using it for vector size
22      int x[N][1] = {{2}, {4}, {6}, {8}}; // column vector x of size N x 1
23      int y[N][1] = {{1}, {3}, {5}, {7}}; // column vector y of size N x 1
24      int result;
25
26      // since N is constant therefore it can not be passed by reference.
27      dot_product(x, y, N, &result);
28
29      printf("Dot product = %d", result);
30
31      return 0;
32  }
33
34  /* Outputs
35  dot product = 100
36  */
```

10.4.3 Matrix vector multiplication

If matrix $A \in R^{m \times n}$ and vector $x \in R^n$, then Matrix-vector multiplication is given by ,

$$y_i = \sum_{j=1}^{n} A_{ij} x_j + y_i, \ i = 1, \cdots, m \tag{10.1}$$

Listing 10.18: **Algorithm** Matrix-Vector Product : $A \in R^{m \times n}$ and $x \in R^n$

```
for i = 1:m do
    for j = 1:n do
        result(i) = result(i) + A(i,j)x(j)
    end for
end for
```

Listing 10.18 can be used to implement the equation (10.1), as shown in Listing 10.19. **Note that, in the listing, the sizes (Lines 8-9) are defined as global variable**, as we need to pass them in the function prototype and definitions as well; as the arrays are bound for all dimension except first.

Note: Further, we need not to pass the global variable in the function, as shown in Listing 10.20 whose functionality is same as Listing 10.19; but in Listing 10.19, global variable is passed to function, just to show the limitations of 'variables' in defining the 'array size'.

Listing 10.19: Matrix vector multiplication

```
1  // matrix_vector_mul.c
2  // matrices and vector product
3
4  #include <stdio.h>
5
6  // Note that arrays are bound for all dimensions except first. Therefore,
7  // size is defined as global variable, as we need to pass them in function.
8  const int M = 2;   // constant is required as we are using it for array size
9  const int N = 3;   // constant is required as we are using it for array size
10
11 void matrix_vect_product(int [][N], int [][1], int , int, int [][1]);
12
13 // calculate matrix-vector product
14 void matrix_vect_product(int A[][N], int x[][1], int M, int N, int result[][1]){
15
16     int i, j;
17
18     for (i=0; i<M; i++){
19         for (j=0; j<N; j++)
20             result[i][0] += A[i][j] * x[j][0];
21     }
22 }
23
24 int main(){
25     int i;
26     int x[N][1] = {{2}, {4}, {6}}; // vector x
27     int A[M][N] = {
28             {1, 3, 5},
29             {2, 3, 4}
30     }; // Matrix A of size M x N
31     int result[M][1] = {0};  // initialize with zero
32
```

(continues on next page)

(continued from previous page)

```
    // since M is constant therefore it can not be passed by reference.
    matrix_vect_product(A, x, M, N, result);

    printf("Result = ");
    for (i=0; i<M; i++)
        printf("%d, ", result[i][0]);

    return 0;
}

/* Outputs
Result = 44, 40,
*/
```

Listing 10.20: Matrix vector multiplication (global variables are not passed in function)

```
// matrix_vector_mul_correct.c
// matrices and vector product

#include <stdio.h>

// Note that arrays are bound for all dimensions except first. Therefore,
// size is defined as global variable, as we need to use them in function.
const int M = 2;   // constant is required as we are using it for array size
const int N = 3;   // constant is required as we are using it for array size

void matrix_vect_product(int [][N], int [][1], int [][1]);

// calculate matrix-vector product
void matrix_vect_product(int A[][N], int x[][1], int result[][1]){

    int i, j;

    for (i=0; i<M; i++){
        for (j=0; j<N; j++)
            result[i][0] += A[i][j] * x[j][0];
    }
}

int main(){
    int i;
    int x[N][1] = {{2}, {4}, {6}}; // vector x
    int A[M][N] = {
            {1, 3, 5},
            {2, 3, 4}
    }; // Matrix A of size M x N
    int result[M][1] = {0};  // initialize with zero

    // glablal variables are not passed...
    matrix_vect_product(A, x, result);

    printf("Result = ");
    for (i=0; i<M; i++)
        printf("%d, ", result[i][0]);

    return 0;
}

/* Outputs
```

(continues on next page)

(continued from previous page)

```
Result = 44, 40,
*/
```

10.4.4 Outer product

Listing 10.21 implements the outer product of two vectors i.e. xy^T, where $x \in R^m$ and $y \in R^n$ and the result $A \in R^{m \times n}$; which is implemented in Listing 10.22.

Listing 10.21: **Algorithm** Outer Product : $x \in R^m$ and $y \in R^n$

```
for i = 1:m do
    for j=1:n do
        result(i,j) = result(i,j) + x(i)y(j)
    end for
end for
```

Listing 10.22: Outer Product : $x \in R^m$ and $y \in R^n$

```
// outer_product.c

#include <stdio.h>

// Note that arrays are bound for all dimensions except first. Therefore,
// size is defined as global variable, as we need to use them in function.
const int M = 2;   // constant is required as we are using it for array size
const int N = 3;   // constant is required as we are using it for array size

void outer_product(int [][1], int [][1], int [][N]);

// calculate matrix-vector product
void outer_product(int x[][1], int y[][1], int result[][N]){

    int i, j;

    for (i=0; i<M; i++){
        for (j=0; j<N; j++)
            result[i][j] += x[i][0]*y[j][0];
    }
}

int main(){
    int i, j;
    int x[M][1] = {{2}, {4}}; // vector x
    int y[N][1] = {{2}, {4}, {6}}; // vector x
    int result[M][N] = {0}; // outer product of size M x N

    outer_product(x, y, result);

    printf("Result : ");
    for (i=0; i<M; i++){
        printf("\nRow %d = ", i);
        for (j=0; j<N; j++)
            printf("%d, ", result[i][j]);
    }
    return 0;
}

/* Outputs
Result :
```

(continues on next page)

(continued from previous page)

```
Row 0 = 4, 8, 12,
Row 1 = 8, 16, 24,
*/
```

10.4.5 Matrix-Matrix multiplication

Two matrices $A \in R^{m \times p}$ and $B \in R^{p \times n}$, can be multiplied using Listing 10.23, as shown in Listing 10.24.

Listing 10.23: **Algorithm** Matrix-Matrix multiplication : $A \in R^{m \times p}$ and $y \in R^{p \times n}$

```
for i = 1:m do
    for j=1:n do
        for k=1:p do
            result(i,j) = result(i,j) + A(i,k)B(k,j)
        end for
     end for
end for
```

Listing 10.24: Matrix-Matrix multiplication : $A \in R^{m \times p}$ and $y \in R^{p \times n}$

```c
// matrix_matrix_mul.c
// matrix-matrix product

#include <stdio.h>

// Note that arrays are bound for all dimensions except first. Therefore,
// size is defined as global variable, as we need to use them in function.
const int M = 2;   // constant is required as we are using it for array size
const int N = 3;   // constant is required as we are using it for array size
const int P = 3;   // constant is required as we are using it for array size

void matrix_matrix_mul(int [][P], int [][N], int [][N]);

// calculate matrix-matrix product
void matrix_matrix_mul(int A[][P], int B[][N], int result[][N]){

    int i, j, k;

    for (i=0; i<M; i++){
        for (j=0; j<N; j++){
            for(k=0; k<P; k++){
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main(){
    int i, j;
    int A[M][P] = {{2, 3, 4}, {4, 5, 6}};
    int B[P][N] = {
            {1, 3, 5},
            {2, 3, 4},
            {1, 2, 3}
    }; // Matrix A of size M x N
    int result[M][N] = {0};  // initialize with zero

```

(continues on next page)

(continued from previous page)

```c
    matrix_matrix_mul(A, B, result);

    printf("Result = ");
    for (i=0; i<M; i++){
        printf("\nRow %d : ", i);
        for (j=0; j<N; j++)
            printf("%d, ", result[i][j]);
    }
    return 0;
}

/* Outputs
Result =
Row 0 : 12, 23, 34,
Row 1 : 20, 39, 58,
*/
```

10.4.6 Upper Triangular Matrices multiplication

Two upper triangular matrices $A, B \in R^{n\times n}$ can be multiplied using Listing 10.25, as shown in Listing 10.26.

Listing 10.25: **Algorithm** Matrix-Matrix multiplication : $A \in R^{m\times p}$ and $y \in R^{p\times n}$

```
for i = 1:n do
    for j=i:n do
        for k=i:j do
            result(i,j) = result(i,j) + A(i,k)B(k,j)
        end for
    end for
end for
```

Listing 10.26: Triangular Matrix multiplication : $A \in R^{n\times n}$ and $y \in R^{n\times n}$

```c
// triangular_matrix_mul.c
// trinangular matrix multiplication

#include <stdio.h>

const int N = 3;

void matrix_matrix_mul(int [][N], int [][N], int [][N]);

// calculate matrix-matrix product
void matrix_matrix_mul(int A[][N], int B[][N], int result[][N]){

    int i, j, k;

    for (i=0; i<N; i++){
        for (j=i; j<N; j++){    // starts from 'i'
            for(k=i; k<=j; k++){  // loop start from 'i' and ends at 'j'
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main(){
```

(continues on next page)

(continued from previous page)

```c
    int i, j;
    int A[N][N] = {{2, 3, 4}, {0, 5, 6}, {0, 0, 2}};
    int B[N][N] = {
            {1, 3, 5},
            {0, 3, 4},
            {0, 0, 3}
    };
    int result[N][N] = {0};  // initialize with zero

    matrix_matrix_mul(A, B, result);

    printf("Result = ");
    for (i=0; i<N; i++){
        printf("\nRow %d : ", i);
        for (j=0; j<N; j++)
            printf("%d, ", result[i][j]);
    }
    return 0;
}

/* Outputs
Result =
Row 0 : 2, 15, 34,
Row 1 : 0, 15, 38,
Row 2 : 0, 0, 6,
*/
```

10.4.7 Transpose of matrix

Transpose of matrix $A \in R^{m \times n}$ can be obtained by Listing 10.27, as shown in Listing 10.28.

Listing 10.27: **Algorithm** Transpose of matrix $A \in R^{m \times n}$

```
for i = 1:m do
    for j=i:n do
        result(j,i) = A(i,j)
    end for
end for
```

Listing 10.28: Transpose of matrix $A \in R^{m \times n}$

```c
// matrix_transpose.c

#include <stdio.h>

const int M = 2;
const int N = 3;

void matrix_transpose(int [][N], int [][M]);

// calculate matrix-transpose
void matrix_transpose(int A[][N],  int result[][M]){

    int i, j;

    for (i=0; i<M; i++){
        for (j=0; j<N; j++){
                result[j][i] = A[i][j];
        }
```

(continues on next page)

(continued from previous page)

```
    }
}

int main(){
    int i, j;
    int A[M][N] = {
                    {2, 3, 4},
                    {1, 9, 5}
                };
    int result[N][M] = {0};  // initialize with zero

    matrix_transpose(A, result);

    printf("Result = ");
    for (i=0; i<N; i++){
        printf("\nRow %d : ", i);
        for (j=0; j<M; j++)
            printf("%d, ", result[i][j]);
    }
    return 0;
}

/* Outputs
Result =
Row 0 : 2, 1,
Row 1 : 3, 9,
Row 2 : 4, 5,
*/
```

10.5 Distributions

In this section, various distribution functions are generated, which are used for generating data in mathematical simulations.

10.5.1 Uniform distribution

In Uniform distribution, the random samples are generated with equal density as shown in Fig. 10.1. The data for the figure is generated using Listing 10.29 and saved in the file 'uniform_data.txt'; then data in the file is plotted using Python as show in Listing 10.30.

Listing 10.29: Uniform distribution

```
// uniform_distribution.c
// data is saved in uniform_data.txt file for further processing using python

#include <stdio.h>
#include <stdlib.h>  // required for rand()

int SAMPLES = 100000;  // total number of samples

double uniform_distribution(void);

// function to generate uniform distribution in range [0, 1)
double uniform_distribution() {
    // +2 is used to exclude the 1 in distribution...
    return (double)rand() / (double)(RAND_MAX + 2);  // RAND_MAX = 32767
}
```

(continues on next page)

(continued from previous page)

```c

int main (void){
    int i;
    double uniform_data;

    FILE *fPtr; // file pointer

    if ((fPtr = fopen ("uniform_data.txt", "w")) == NULL){  // "w" = write mode
        printf("File can not be open\n");
    }
    else {
        for (i=0; i<SAMPLES; i++){
        uniform_data = uniform_distribution();
        fprintf(fPtr, "%f, ", uniform_data);   // save data to file
        }
    }

    fclose (fPtr);
    return 0;
}
```

Listing 10.30: Plot data generated by Listing 10.29

```python
# plot_uniform_dist.py
# plot the data generated by file 'uniform_distribution.c'

import matplotlib.pyplot as plt
import numpy as np

f = open(r'uniform_data.txt', 'r').read()  # read file
data_string = f.split(",")   # split content of file on ','
data_list = []
for d in data_string:
    try:
        data_list.append(float(d))  # convert string-value to float and store in list
    except ValueError as error:
        continue

# numpy array is required for calculating 'mean' and variance of distributions
data_array = np.array(data_list, 'float')  # convert python-list to numpy-array

print("Minimum value (a) = ", min(data_array))  # show minimum value of data_array
print("Maximum value (b) = ", max(data_array))  # show maximum value of data_array

 # Calculate mean and variance : a = min value , b = max value
print("Mean (a+b)/2 = ", data_array.mean())  # mean
print("Variance [(b-a)^2]/12 = ", data_array.var())  # variance

plt.hist(data_array, 20)  # Histogram with 20 bins
plt.show()

# Outputs:
# Minimum value (a) =  0.0
# Maximum value (b) =  0.999939
# Mean value =  0.50144555508     i.e. (1+0)/2
# Mean value =  0.0832357216637   i.e. ( (1-0)^2 /12 )
```

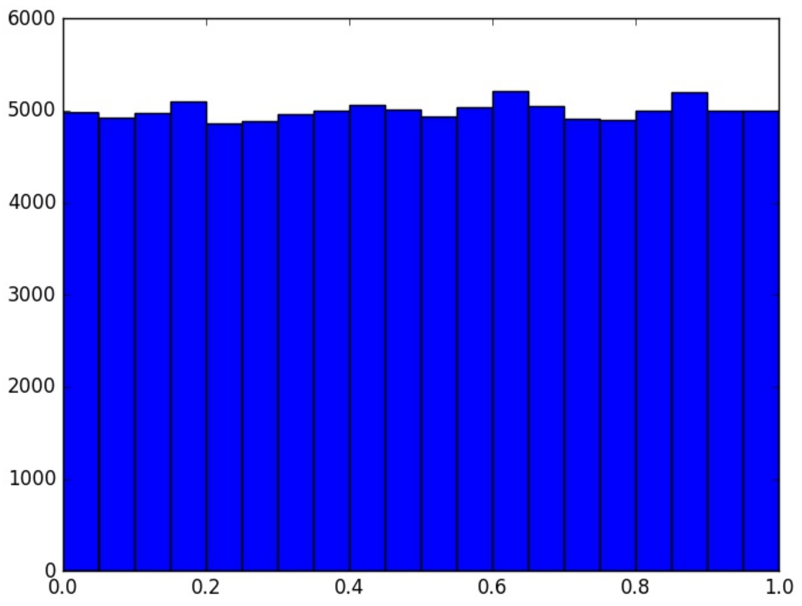


Fig. 10.1: Uniform distributions generated by Listing 10.29

10.5.2 Gaussian distribution

Gaussian distribution can be implemented using ‘Box-Muller method’. This method uses ‘uniform distribution’ for generating the Gaussian distribution, as shown at Lines 35-42 of Listing 10.31. The results of the listing is plotted using Listing 10.32 and result is shown in Fig. 10.2.

Listing 10.31: Gaussian distribution

```
1  // gaussian_distribution.c
2  // data is saved in uniform_data.txt file for further processing using python
3  
4  #include <stdio.h>
5  #include <stdlib.h>  // required for rand()
6  #include <math.h>    // required for sqrt, log and cos etc.
7  
8  # define PI 22/7
9  
10 int SAMPLES = 100000;  // total number of samples
11 
12 double uniform_distribution(void);
13 
14 // function to generate uniform distribution in range [0, 1)
15 double uniform_distribution() {
16     // +2 is used to exclude the 1 in distribution...
17     return (double)rand() / (double)(RAND_MAX + 2);  // RAND_MAX = 32767
18 }
19 
20 int main (void){
21     int i;
22     double uniform_data1, uniform_data2;
23     double gaussian_data;
24 
25     FILE *fPtr; // file pointer
26 
```

(continues on next page)

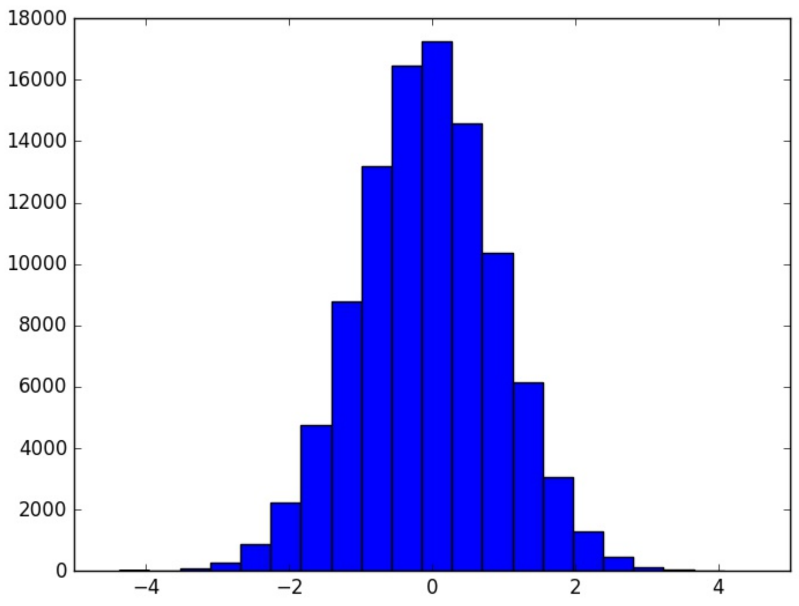


Fig. 10.2: Gaussian distributions generated by Listing 10.31

(continued from previous page)

```
    if ((fPtr = fopen ("gaussian_data.txt", "w")) == NULL){  // "w" = write mode
        printf("File can not be open\n");
    }
    else {
        for (i=0; i<SAMPLES; i++){
        uniform_data1 = uniform_distribution();
        uniform_data2 = uniform_distribution();

        // Box-Muller method to generate Guassian distribution
        // exclude 0, as it log(0) is infinite
        if (uniform_data1 != 0  && uniform_data2 !=0)
            gaussian_data = sqrt(-2.0 * log(uniform_data1)) *
                                    cos(2 * PI * uniform_data2);     // use this or below

            // gaussian_data = sqrt(-2.0 * log(uniform_data1)) *
            //                          sin(2 * PI * uniform_data2);  // use this or above

        fprintf(fPtr, "%f, ", gaussian_data);   // save data to file
        }
    }

    fclose (fPtr);
    return 0;
}
```

Listing 10.32: Plot data generated by Listing 10.31

```
# plot_gaussian_dist.py
# plot the data generated by file 'gaussian_distribution.c'

import matplotlib.pyplot as plt
import numpy as np

```

(continues on next page)

(continued from previous page)

```python
f = open(r'gaussian_data.txt', 'r').read()  # read file
data_string = f.split(",")   # split content of file on ','
data_list = []
for d in data_string:
    try:
        data_list.append(float(d))  # convert string-value to float and store in list
    except ValueError as error:
        continue

# numpy array is required for calculating 'mean' and variance of distributions
data_array = np.array(data_list, 'float')  # convert python-list to numpy-array

print("Minimum value = ", min(data_array))  # show minimum value of data_array
print("Maximum value = ", max(data_array))  # show maximum value of data_array

# Calculate mean and variance
print("Mean = ", data_array.mean())  # mean
print("Variance = ", data_array.var())  # variance

plt.hist(data_array, 20)  # Histogram with 20 bins
plt.show()

# Outputs:
# Minimum value (a) =  -4.528262
# Maximum value (b) =  4.338116
# Mean =  0.00805113272
# Variance =  1.04446231254
```

When a man is, through his desires, confronted with great suffering , he understands their true nature; so when such suffering comes, it should be welcome. Suffering may come in order to eliminate further suffering. Suffering has to come, when it is of use in purging the soul from it's desires.

–Meher Baba

Chapter 11

Introduction to C++

11.1 Introduction

In previous chapters, we learn various features of C programming language. Now, we will discuss the C++ programming language, which can be considered as the superset of C. In the other words, all the codes in the previous chapters can be compiled directly using C++ compiler (i.e. using **g++**, instead of **gcc**), but vice-versa is not possible, as we saw in Listing 3.5, where 'or' keyword was used, and the code was compiled using 'g++'.

The major difference between these two languages is that the C language is 'procedural programming language' whereas the C++ language is the **combination** of 'procedural programming' and 'object oriented programming (OOP)'. In this chapter, we will re-implement the previous codes using standard 'C++ headers' (instead of using 'C headers' in C++), which show the differences in 'procedural programming' using these two languages. Then OOP features of C++ will be discussed in next chapter. Lastly, the C codes are saved with extension '.c', whereas C++ codes are saved with extension '.cpp' or '.cc'. We will use '.cpp' extension in this tutorial.

11.2 Writing first code

Listing 11.1 shows the source code to print the 'Hello World' on the screen. To execute the code in the listing, open the terminal and go to the folder where program is saved and type following commands; which will generate the output on the terminal.

```
$ g++ -o out helloWorld.cpp
$ ./out                         (in Unix)
$ out                           (in Windows)
```

Explanation Listing 11.1

C++ is also **case-sensitive** language. Similar to C language, each program contains one and only one 'main' function, which is the starting point of the code. In this listing, Line 6 has the 'main' function, which contains two other keywords i.e. 'int' and 'void'; here 'void' indicates that, the main 'function' does not have any input parameter, whereas 'int' represents that the return type of this function is 'integer'.

Note that, in C++ the 'iostream' is the header file (instead of 'stdio.h'), which contains all the input and output formats i.e. 'cin' and 'cout' respectively. The function of 'cout' and 'cin' is similar to 'printf' and 'scanf' statements in C. Only these statements are needed to be changed in the previous chapters to convert the C codes into C++, which is discussed in this chapter. Also, the complete list of C++ keywords are shown in Table 1.1.

In Line 9, keyword 'cout' is used, which prints the characters on the screen. Next 'namespace std::' is used along with the 'cout' because 'cout' is defined in this namespace. Further, 'std' is defined in 'iostream header file' therefore it is included at Line 4. Then, 'endl' is used to terminate the line, which is also defined in the 'namespace std'. Lastly '<<" is known as 'stream insertion operator' which inserts the values to output stream i.e. ‘Hello’, ‘end of line (i.e. line-change)’ and ‘World’ will be inserted to output stream. Hence, ‘Hello’ will be displayed at first line and ‘World’ will be displayed at second line in the output, as shown in Lines 14-15.

Similar to C, multiple lines can be commented by using ‘/ *’ and ‘* /’ as shown in Lines 13-16; whereas ‘//’ is used for single line comments. The comments make code more readable; e.g. in this listing, functions of all the lines are described as comments.

Listing 11.1: Print Hello World

```
1  //helloWorld.cpp
2
3  //header file: iostream
4  #include <iostream> // required for cin, cout and endl
5
6  int main(void) //return type: int, input arguments: void i.e. none
7  {
8      // cout belongs to namespace 'std', hence std::cout
9      std::cout << "Hello" << std::endl << "World"; // Hello World
10     return 0; //return 0 i.e. int,
11 }
12
13 /* Outputs
14 Hello
15 World
16 */
```

11.3 ‘Namespaces’ and ‘Line termination’

In Listing 11.1, we used ‘std::’ at various places. To avoid writing the ‘std::’, we can define the namespace at the top of the code with keywords ‘using namespace’, as shown in Listing 11.2. Further, ‘n’ is used for ending the line at Line 10 (instead of endl) .

Listing 11.2: Using namespace

```
1  // helloWorldNamespace.cpp
2
3  #include <iostream> // required for cin and cout
4  //use below line to use cout in the main(), instead of std::cout.
5  using namespace std;
6
7  int main(void) //return type: int, input arguments: void i.e. none
8  {
9      cout << "Hello \nWorld. " << "How Are you?" << endl;
10
11     /* cout << // line is terminated by ';' not by white spaces
12      "Hello \nWorld. "
13      << "How Are you?" << endl; */
14
15     return 0; //return 0 i.e. int,
16 }
17
18 /* Outputs
19 Hello
20 World. How Are you?
21 */
```

Note that, Line 9 can be written in multiple lines as well as shown in Lines 11-13 (uncommet these lines to check the outputs). In the other words, it is the ‘;’ which terminates the line, not the white spaces.

11.4 C to C++ conversion examples

In this section, some of the previous codes are re-written using C++. Here, we can see that, to convert the C code to C++, we need to change the header file i.e. **replace 'stdio.h' with 'iostream'; and then substitute all the 'printf' and 'scanf' statements with 'cout' and 'cin' statements respectively**.

11.4.1 'cin' and 'cout'

In Listing 11.3, the name is provide by the user (Line 11) and then message is displayed through Line 13, which is similar to Listing 2.5.

Listing 11.3: 'cin' and 'cout'

```
1  //cinEx.cpp
2  
3  #include <iostream>
4  using namespace std;
5  
6  int main(){
7  
8     char name[30]; // variable 'name' of type character with max length 30
9  
10    cout << "Enter your name: \t"; // show message on the termainal
11    cin >> name; // get the value of name from user
12 
13    cout << "Hello " <<  name;  // Print Hello + name-provided-by-user
14 
15    return 0;
16 }
17 
18 /* Outputs
19 Enter your name:     Meher
20 Hello Meher */
```

11.4.2 Dot product

Listing 11.4 performed the dot product of two vectors which is discussed in Listing 10.17.

Listing 11.4: Dot product of vectors

```
// dot_product.cpp
// matrices must be of same size for dot-product

#include <iostream>
using namespace std;

void dot_product(int [][1], int [][1], int , int *);

// calculate dot product
void dot_product(int x[][1], int y[][1], int N, int *result){

    int i;
    *result = 0;

    for (i=0; i<N; i++){
       *result += (x[i][0] * y[i][0]);
    }
}
```

(continues on next page)

(continued from previous page)

```

int main(){

    const int N = 4;   // constant is required as we are using it for vector size
    int x[N][1] = {{2}, {4}, {6}, {8}}; // column vector x of size N x 1
    int y[N][1] = {{1}, {3}, {5}, {7}}; // column vector y of size N x 1
    int result;

    // since N is constant therefore it can not be passed by reference.
    dot_product(x, y, N, &result);

    cout << "Dot product = " << result;

    return 0;
}

/* Outputs
dot product = 100
*/
```

11.4.3 Bitwise operators

Listing 11.5 shows the bitwise operations, which is discussed in Listing 3.2.

Listing 11.5: Bitwise operators

```
// shiftOpEx.cpp

#include <iostream>
using namespace std;

int main(){
    int a = 0x2;  // 0010

    int b, c;
    int xor_op, not_op_bad, not_op_good;

    // shift right by one bit and save to b
    b = a >> 1;  // 0001 = 1
    c = a << 2;  // 1000 = 8

    cout << "a = "<< hex << a << ", b = " << hex << b << ", c = " << hex << c << endl;

    xor_op = a ^ b; // 0011 = 3

    cout << "xor_op = " << hex << xor_op << endl;

    not_op_bad = ~a; // fffffffd (expected output = 1101 = 13 = 'd')
    cout << "not_op_bad = " << hex << not_op_bad << endl;

    // suppress all 'f' (i.e. make them 0) by 'and-operation' of 'a' with '0000000f' = 15
    not_op_good = ~a & 0xf; // or not_op_good = ~a & 15;
    cout << "not_op_good = " << hex << not_op_good << endl;
}

/* Outputs
a = 2, b = 1, c = 8
xor_op = 3
not_op_bad = fffffffd
```

(continues on next page)

(continued from previous page)

```
not_op_good = d
*/
```

11.5 Formatted input and outputs

Similar to <Chapter 9, in this section we will see various formatting commands for inputs and outputs in C++.

11.5.1 Integer values in Binary, Hex, Octal & Decimal formats

In this section, we will learn to print the **integer values** in different formats i.e. Binary, Hexadecimal, Octal and Decimal. This can be done using keywords 'bitset', 'hex', 'oct' and 'dec' as shown in Listing 11.6. For further understanding, see the comments and outputs of the code in the listing. **Note that, the variable 'h' is defined as 'hexadecimal' and when it is printed at Line 28, the default output is in 'octal format'; and when again it is printed at Line 32, the default output is in 'decimal format'.** Therefore, it is better to define the print-format for hexadecimal and octal values.

Listing 11.6: Printing numbers in different formats

```
 1  //hecDec.cpp
 2
 3  #include <iostream>
 4  #include <iomanip> // required for std::setbase
 5  #include <bitset> // required for bitset<>(for binary conversion)
 6  using namespace std; // std::hex, std::oct, std::dec
 7
 8  int main(void){
 9
10      int x = 15;
11
12      // hexadecimal number starts with 0x
13      // h is defined as hex 0xb = 13(oct) = 11(dec)
14      int h = 0xb;  // be careful while printing it (see Line 53 and 57)
15
16      // octal number, starts with 0
17      int o = 013; // be careful while printing it as well (see Line 53 and 57)
18
19      cout << " x = " << x << endl << endl;
20      cout << " hex value of x = " << hex << x << endl; // hex
21      cout << " oct value of x = " << oct << x << endl; // octal
22      cout << " decimal value of x = " << dec << x << endl << endl; // decimal
23      cout << " binary value of x = " << bitset<8>(x) << endl << endl; // binary
24
25      cout << " hex value of x (using setbase) = " << setbase(16) << x << endl;
26      cout << " oct value of x (using setbase) = " << setbase(8) << x << endl << endl;
27
28      cout << " h = " << h << endl;  // by default print as oct.
29      cout << " hex value of h = " << hex << h << endl;
30      cout << " oct value of h = " << oct << h << endl;
31      cout << " decimal value of h = " << dec << h << endl;
32      cout << " h = " << h << endl << endl;  // now, by default print as decimal.
33
34      cout << " oct value of o = " << oct << o << endl;
35      cout << " hex value of o = " << hex << o << endl;
36      cout << " decimal value of o = " << dec << o << endl;
37
38      return 0;
39  }
```

(continues on next page)

(continued from previous page)

```

/*
 x = 15

 hex value of x = f
 oct value of x = 17
 decimal value of x = 15

 binary value of x = 00001111

 hex value of x (using setbase) = f
 oct value of x (using setbase) = 17

 h = 13
 hex value of h = b
 oct value of h = 13
 decimal value of h = 11
 h = 11

 oct value of o = 13
 hex value of o = b
 decimal value of o = 11
 */
```

11.5.2 Floating point values in Scientific and Fixed notations

Floating point values can be represented in Scientific and Fixed point notation using keywords '**scientific**' and '**fixed**' respectively, as shown in Listing 11.7. Also, we can set the precision for the outputs using '**setprecision}' as shown in Line 17 of the listing. **Also, note the ambiguities between the actual value of 'a' and all the results**.

Listing 11.7: Floating point values in Scientific and Fixed notations

```
1  //floatFormat.cpp
2
3  #include <iostream>
4  #include <iomanip>  // setw
5  using namespace std;
6
7  int main(void){
8
9      float a=123.32435895;
10
11     cout << a << endl;  // 123.324
12
13     cout << scientific << a << endl;  // 1.233244e+002
14     cout << fixed << a << endl << endl; // 123.324356
15
16     // 1.23324e+002
17     cout << setprecision(5) << scientific << a << " <-- setprecision(5) " << endl;
18     // 123.32436
19     cout << fixed << a << "  <-- automatically using above precision" << endl;
20     cout << setprecision(6) << fixed << a << endl;  // 123.324356
21
22     return 0;
23 }
24
25 /*
26 123.324
```

(continues on next page)

(continued from previous page)

```
1.233244e+002
123.324356

1.23324e+002 <-- setprecision(5)
123.32436  <-- automatically using above precision
123.324356
 */
```

11.5.3 Modify ‘cout’ format

The ‘setw(10)’ is used for setting the minimum width as 10 for the outputs, which is similar to ‘%10d’ in C. Similarly, ‘left’ and ‘right’ keywords are used to left-justify and right-justify the outputs, as shown in Listing 11.8.

Listing 11.8: Modify ‘cout’ format

```
//coutFormat.cpp

#include <iostream>
#include <iomanip>  // setw
using namespace std;

int main(void){

    int x=150, y=12;
    float a=1.230, b=1.200;

    cout << " --- Right justified --- \n";
    cout << setw(10) << right << x << endl;
    cout << right << setw(10) << y << endl;
    cout << setw(10) << a << endl;
    cout << setw(10) << b << endl;

    cout << "\n --- Left justified --- \n";
    cout << left << setw(10) << x << endl;
    cout << left << setw(10) << y << endl;
    cout << setw(10) << left << a << endl;
    cout << left << setw(10) << b << endl;

    // for floating point numbers
    cout << "\n --- 'showpoint for trainling zeros ' --- \n";
    cout << setw(10) << showpoint << a << endl;
    cout << setw(10) << showpoint << b << endl;

    return 0;
}

/*
 --- Right justified ---
       150
        12
      1.23
       1.2

 --- Left justified ---
150
12
1.23
1.2

```

(continues on next page)

(continued from previous page)

```
 --- 'showpoint for trainling zeros' ---
1.23000
1.20000
 */
```

11.5.4 'cin' formats

The **hex** and **oct** keywords can be used with 'cin' statement to read the Hexadecimal and Octal values respectively, as shown in Listing 11.9.

Listing 11.9: 'cin' formats

```
//cinFormats.cpp

#include <iostream>
#include <iomanip>  // setw
using namespace std;

int main(void){

    int i, j;


    cout << "Enter Hexadecimal Value:\t";
    cin >>  hex >> i;
    cout << "i = " << i << endl;

    cout << "Enter Octal Value:\t";
    cin >> oct >> j;
    cout << "j = " << j;

    return 0;

    return 0;
}

/*
Enter Hexadecimal Value: 0xc
i = 12

Enter Octal Value:       014
j = 12
 */
```

11.6 Type 'string'

C++ provide the 'std::string' type to handle strings. Listing 5.8 is re-implemented using 'string-type' in Listing 11.10. Please note the following things about string,

- Size of 'string' is 4 (see output of Line 19); whereas the size of 'char' is 1 byte.
- Value to string can be assigned using 'cin' (Line 16). And similar to char, the input will be read till first 'space' i.e. input 'Meher Krishna' will be read as 'Meher' (see output at Line 60 and 64).
- 'string' can be passed to function as 'const' as well (Line 28); which can not be modified by the function i.e. Line 36 will generate error.
- Note that, string 'b' contains '7' characters, and Lines 40-44 assigns 10 'x' to 'b'. If we look the output of Line 55, then we find that there are only 7 'x' (i.e. equal to size defined at Line 13). In the other word, string size can not be modified after initialization or after getting the input from user.

- If we try to print the string values above it's size (i.e. 7 here), then garbage values will be printed (see output of Line 55).
- Lastly, similar to Listing 5.8, it better to use 'Null character $(i.e. backslash 0)$' in the 'for loop' to access the elements of the string (uncomment Lines 47-52) instead of Lines 39-45.

Listing 11.10: Type 'string'

```
1  // stringType.cpp
2  // parameter passed as 'const' can not be modified
3  
4  #include <iostream>
5  using namespace std; // string is defined in it
6  #define SIZE 10
7  
8  void stringFunc(const string, string);
9  
10 int main(){
11 
12     string a; // size of string is 4 Bytes
13     string b = "String2"; // size of string is 4 Bytes
14 
15     cout << "Enter string a: ";
16     cin >> a;
17 
18     // strings are always passed by reference
19     cout << "Size of a and b are "<< sizeof(a)
20         << " and " << sizeof(b)  << " respectively\n";
21 
22     stringFunc(a, b);
23 
24     return 0;
25 }
26 
27 // strings are always passed by reference
28 void stringFunc(const string a,  string b){
29     int i;
30 
31     for (i=0; i<SIZE; i++){
32         cout << a[i] << ", ";
33 
34         // below line will generate error as string 'a' is
35         // passed with 'const' keyword.
36         // a[i] = 'x';
37     }
38 
39     cout << endl;
40     for (i=0; i<SIZE; i++){
41         cout << b[i] << ", ";
42 
43         // double quote can not be used i.e. "x" is invalid
44         b[i] = 'x';
45     }
46 
47     // cout << endl;
48     // for (i=0; b[i] != '\0'; i++){
49     //     cout << b[i] << ", ";
50 
51     //     b[i] = 'x';
52     // }
53 
54     cout << endl << a;
55     cout << endl << b;
```

(continues on next page)

(continued from previous page)

```

}

/* Output
Enter string a: Meher Krishna            // only Meher will be read
Size of a and b are 4 and 4 respectively
M, e, h, e, r,  ,  ,  ,  ,  ,
S, t, r, i, n, g, 2,  , $, ^,
Meher
xxxxxxx
*/
```

11.7 Scope resolution operator '::'

Scope resolution operation can be used when the namespace-scope or global-variable-scope is hidden in a function due to same name, as shown in Listing 11.11. Here, local variable 'a' is defined at Line 10, which hides the access to global variable (due to same name). Now, '::' should be used to access the global variable as shown in Line 15.

Listing 11.11: Scope resolution operator

```
1  // scopeResolutionEx.cpp
2
3  #include <iostream>
4  using namespace std;
5
6  int a=20;  // global variable
7
8  int main(){
9
10     int a = 4;
11
12     cout << "Local Variable :  " << a << endl;
13
14     // use scope-resolution-operator (::) to access global variable
15     cout << "Glabal Variable :  " << ::a;
16
17     return 0;
18 }
19
20 /* Output
21 Local Variable :  4
22 Glabal Variable :  20
23 */
```

11.8 Function overloading

C++ allows the function overloading, which is not allowed in C. In function overloading, two functions can have same; and the correct function is called based on the **types of the input-parameters**, as shown in Listing 11.12. In the listing, the function 'add2Num' is used 4 times with different input-parameters. Since Line 43 contains inputs of type 'integer' therefore function at Line 11 will be invoked. In the same way other functions will be invoked based on input types as shown in the listing.

Listing 11.12: Function overloading

```cpp
// functionOverload.cpp

#include <iostream>
using namespace std;

int add2Num(int, int);
float add2Num(int, float);
float add2Num(float, float);

// function 1
int add2Num(int a, int b){
    cout << "int + int = " ;
    return a+b;
}

// function 2
float add2Num(int a, float b){
    cout << "int + float = ";
    return a+b;
}

// function 3
float add2Num(float a, int b){
    cout << "float + int = ";
    return a+b;
}

// function 4
float add2Num(float a, float b){
    cout << "float + float = ";
    return a+b;
}


int main(){

    int w = 4;
    int x = 3;
    float y = 3.5;
    float z = 2.5;

    // int + int
    cout << add2Num(w, x) << endl;  // function 1 will be called

    // int + float
    cout << add2Num(x, y) << endl;  // function 2 will be called

    // float + int
    cout << add2Num(y, x) << endl;  // function 3 will be called

    // float + float
    cout << add2Num(y, z) << endl;  // function 4 will be called

    return 0;
}

/* Output
int + int = 7
int + float = 6.5
float + int = 6.5
```

(continues on next page)

(continued from previous page)

```
float + float = 6
*/
```

11.9 Namespaces

Listing 11.13: Namespaces

```cpp
// namespaceEx.cpp

#include <iostream>
#include <cmath>

using std::cout;
using std::endl;

// namespace math1
namespace math1{
    float pi = 3.14;
    float perimeter(float diameter){
        return (pi*diameter);
    }
}


float perimeter(float );

float perimeter(float diameter){
        return (3.15*diameter);
    }

int main(){
    int pi = 3;
    int diameter;
    // using std::cout is declared at the top
    cout << "Enter diameter : "; // therefore only 'cout' is used here
    std::cin >> diameter; // std::cin is used here

    cout << "pi in main() : " << pi << endl; // print local 'pi'
    cout << "pi in namespace-math1 : " << math1::pi << endl;  // print 'pi' in namespace 'math1'

    cout << "permiter = : " << perimeter(100) << endl;
    cout << "permiter from namespace-math1= : " << math1::perimeter(100) << endl;

    return 0;
}
```

Listing 11.14: Namespace inside Namespace

```cpp
// namespaceEx2.cpp
// namespace inside the namespace

#include <iostream>
#include <cmath>

using std::cout;
using std::endl;

// namespace math1
```

(continues on next page)

(continued from previous page)

```
namespace math1{
    float pi = 3.14;

   namespace pm{
        float perimeter(float diameter){
            return (pi*diameter);
        }
    }
}


float perimeter(float );

float perimeter(float diameter){
        return (3.15*diameter);
    }

int main(){
    int pi = 3;
    int diameter;

    // using std::cout is declared at the top
    cout << "Enter diameter : "; // therefore only 'cout' is used here
    std::cin >> diameter; // std::cin is used here

    cout << "pi in main() : " << pi << endl; // print local 'pi'
    cout << "pi in namespace-math1 : " << math1::pi << endl;  // print 'pi' in namespace 'math1'

    cout << "permiter = : " << perimeter(100) << endl;
    cout << "permiter from namespace-math1= : " << math1::pm::perimeter(100) << endl;

    return 0;
}

/* Outputs
Enter diameter : 10
pi in main() : 3
pi in namespace-math1 : 3.14
permiter = : 315
permiter from namespace-math1= : 314
*/
```

11.10 Conclusion

In this chapter, we wrote simple ‘Hello World’ programs, which illustrate the concept of ‘main function’, ‘name space’ and ‘printing output on screen’. Further, we converted some of the previous C codes into C++ codes. Also, various input and output formats are discussed. Lastly, strings, function overloading and scope resolution operators are also discussed.

But I say to you who listen: Love your enemies, do what is good to those who hate you, bless those who curse you, pray for those who mistreat you.

–Jesus Christ

Chapter 12

File operations in C++

12.1 Write data to file

In C++, for writing or reading data from a file, we need to include the 'fstream' header file as shown in Line 4 of Listing 12.1. Next, we need to open the file, which is done at Line 13 using 'ofstream' and 'ios::out' keywords. This line opens the file as 'outFile' and if file does not open successfully, then 'if statement' at line 15 prints the message at Line 16. Note that, the file 'data.txt' file is saved inside the 'data' folder, therefore we need to create the 'data' folder first, as compiler can not create the folder. If file is open successfully, the compiler will reach to 'else' block at Line 19; where it will read the data from the terminal (Line 24), and then save it to file using Line 25. The 'while loop' is used at Line 24, which reads the data from terminal until 'end of file command' is given i.e. 'ctrl-Z' or 'crtl-C', as shown in Fig. 7.1. Also, the data saved by the listing is shown in the Fig. 7.2. Lastly unlike C, in C++, file is automatically closed at the end of the program.

Listing 12.1: Write data to file

```
1  // writeDataEx.cpp
2
3  #include <iostream>
4  #include <fstream> // file stream
5  using namespace std; // std::ofstream
6
7  int main (void){
8      int id;
9      char product[30];
10     float price;
11
12     //  ofstream : to open file
13     ofstream outFile("data2/data.txt", ios::out);
14
15     // create 'data' folder first...
16     if (!outFile){  // print message if file is not open
17         cout << "File can not be open\n";
18     }
19     else {
20         cout << "Enter details i.e. id, product-name and price\n";
21         cout << "-> ";
22         // cin >> id >> product >> price;
23
24         while(cin >> id >> product >> price){  // ctrl+z to exit
25             outFile << id << " " << product << " " << price << endl; // write to file
26             cout << "-> ";
27         }
28     }
29
```

(continues on next page)

(continued from previous page)

```
    return 0;  // ofstream-destructor will close the file
}

/* Outputs
Enter details i.e. id, product-name and price
-> 1 Radio 12.3
-> 2 Oven 14.5
-> 3 Computer 100
-> 4 Laptop 120
-> ^Z
*/
```

12.2 Read data from file

Reading operation is similar to writing operation. The ‘ifstream’ and ‘ios::in’ keywords are used to read the data from the file, as shown in Line 13 Listing 12.2.

Listing 12.2: Read data from file

```
 1  // readDataEx.cpp
 2
 3  #include <iostream>
 4  #include <fstream> // file stream
 5  using namespace std; // std::ifstream
 6
 7  int main (void){
 8      int id;
 9      char product[30];
10      float price;
11
12      //  ifstream : to read file
13      ifstream readFile("data/data.txt", ios::in);
14
15      // create 'data' folder first...
16      if (!readFile){  // print message if file is not open
17          cout << "File can not be open\n";
18      }
19      else {
20
21          while(readFile >> id >> product >> price){  // ctrl+z to exit
22              cout << id << " " << product << " " << price << endl; // print the data
23          }
24      }
25
26      return 0;  // ifstream-destructor will close the file
27  }
28
29  /* Outputs
30  -> 1 Radio 12.3
31  -> 2 Oven 14.5
32  -> 3 Computer 100
33  -> 4 Laptop 120
34  */
```

12.3 Conclusion

In this chapter, we learn to read and write data to file using C++. We saw that the data can be stored permanently in a file; and then this data can be processed and analyzed by some other software like Python and R etc.

Chapter 13

Standard Template Library (STL)

13.1 Understanding templates

The STL is the collection of **C++ templates**. Therefore, let's understand the template with an example first.

In Listing 13.1, the square of 'int' and 'double' is calculated at Lines 7 and 11. Note that, the two functions are exactly same except the 'types' of the input and the output. This repetition of the code can be removed using templates as shown in Listing 13.2.

Listing 13.1: Square of 'int' and 'double'

```
1  // square_ex.cpp
2  
3  #include <iostream>
4  
5  using namespace std;
6  
7  int square(int x){
8      return x*x;
9  }
10 
11 double square(double x){
12     return x*x;
13 }
14 
15 
16 int main(){
17     int a = 2;
18     double b = 2.5;
19 
20     cout << square(a) << endl; // invoke : int square(int)
21     cout << square(b) << endl; // invoke : float square(float)
22 
23     return 0;
24 }
```

Below is the output of the above code,

```
$ g++ -o out square_ex.cpp
$ ./out
4
6.25
```

Listing 13.2 is the function template, which can calculate the square of both 'int' and 'double' types.

Listing 13.2: Function template for calculating square of ‘int’ and ‘double’

```
// template_ex.cpp

#include <iostream>

using namespace std;


template <typename T>
T square(T x){
    return x*x;
}

int main(){
    int a = 2;
    double b = 2.5;

    cout << square<int>(a) << endl;  // or use 'square(a)' without 'int'
    cout << square<float>(b) << endl; // or use 'square(a)' without 'double'

    return 0;
}
```

Note: For ‘function template’, we can use square(a) instead of square<int>(a); but for the ‘class template’ we need to define the type explicitly.

Below is the output of the above code,

```
$ g++ -o out template_ex.cpp
$ ./out
4
6.25
```

13.2 Standard Template Library

The STL (Standard Template Library) is a collection of **C++ templates** which provides various useful implementations of algorithms and data structures such as ‘vector’, ‘queue’, and ‘list’ etc. Also, these templates are ‘standardized’, efficient’, ‘accurate’ and allows ‘re-usability’. The templates can be divided into three categories,

1. **Containers**: The containers are used to manage the collection of objects e.g. vector, lists and map etc.
2. **Iterators**: The iterators are used to loop through the each element of the the containers.
3. **Algorithms**: The algorithms are used on containers to perform various operations such as sorting, searching and transforming etc.

Listing 13.3 is an example of ‘container’, ‘iterator’ and ‘algorithm’ from the STL library. Here an integer-container i.e. ‘**vector v_int**’ is defined at Line 8. Then the ‘**push_back method**’ is used at Line 16, which append the elements in the vector. Next, the variable ‘**vi**’ of type ‘**iterator**’ is used at Line 20, which stores the initial location of the container ‘v_int’. This iterator ‘vi’ is used to print the values stored in the vector using ‘while’ loop at Lines 21-25. Finally, an **algorithm ‘reverse’** is used at Line 28, which reverse the number stored in the **container ‘v_int’** using **iterators ‘begin()’ and ‘end()’**. Following functions are used in this listing,

- **size()** : returns the size of of the vector.
- **begin()** : returns an iterator to the start of the vector.
- **end()** : returns an iterator to the end of the the vector.
- **push_back** : insert the value at the end of the vector.
- **reverse**: algorithm which uses the the iterators.

Note: In Listing 13.3, the 'vectors' and 'iterators' are defined in the header file 'iterator.h'; whereas the algorithm 'reverse' is defined in the header file 'algorithm.h'.

The 'end()' iterator does not represent the position of the last element, but the position after the last element; hence '**!**=' is used at Line 21 of Listing 13.3 (instead of '==' and '<=').

Listing 13.3: STL example

```
1  // stl_ex.cpp
2  
3  #include <iostream>
4  #include <vector>
5  #include <algorithm>
6  
7  int main(){
8      std::vector<int> v_int; // container : vector of type 'int'
9      // std::vector<float> vf; // vector of type 'float'
10     int i = 0;
11 
12     // print the size of vector,
13     std::cout << "vector size: " << v_int.size() << std::endl;
14 
15     for (int i=0; i<5; i++) // int i : local 'i'
16         v_int.push_back(i);
17     std::cout << "vector size: " << v_int.size() << std::endl;
18 
19     // iterator v_int.begin() is stored in 'vi'
20     std::vector<int>::iterator vi = v_int.begin();
21     while (vi != v_int.end()) { // print using while loop
22         std::cout << *vi << " ";
23         vi++;
24     }
25     std::cout << std::endl;
26 
27     // reverse from iterator 'v_int.begin()' to iterator 'v_int.end()'
28     reverse(v_int.begin(), v_int.end());
29     std::cout << "reverse order" << std::endl;
30     for (int i=0; i<v_int.size(); i++) // print using for loop
31         std::cout << v_int[i] << " ";
32     std::cout << std::endl;
33 
34     return 0;
35 }
```

Below are the outputs for the above code,

```
$ g++ stl_ex.cpp -o out
$ ./out

vector size: 0
vector size: 5
0 1 2 3 4

reverse order
4 3 2 1 0
```

Note: Also we can replace the Line 31 with below line. The 'at()' function throws an exception when index is out of range.

```
std::cout << v_int.at(i) << " ";
```

Important: The aim in OOPs (object oriented programming) is to combine the 'data' and 'algorithms' together using 'objects' of the classes. Further the STL has three separate components i.e. 'Containers', 'Iterators' and 'Algorithms', which goes against the OOPs logic of combining the 'data' and the 'algorithms'. But this separation is quite useful, as it reduces the number of algorithm-implementations in STL by increasing the re-usability. In the other words, with the help of separation, an **algorithms** can be used for different **containers** (instead of one algorithm for each component). This is achieved by an intermediate entity i.e. **iterators**.

13.3 Containers

The containers can be divided into three categories i.e.

- **Sequence container**: It is an ordered collection i.e. each element has unique position in the containers which depends on the 'time' and 'place' of insertion (independent of 'values'). STL contains five sequence containers,
  - vector
  - array
  - deque
  - list
  - forward_list
- **Associative container**: It is a sorted collection where the position of the elements depends on the 'values' of the elements. STL has four associative containers,
  - set
  - map
  - multiset
  - multimap
- **Unordered container**: In this container, the position of the elements do not matter i.e. if we put certain elements in an unordered container then their position may vary with time. STL has four types of unordered container,
  - unordered_set
  - unordered_map
  - unordered_multiset
  - unordered_multimap

Important:

- Sequence containers are used to implement the arrays and link lists.
- Associative containers are used to implement the binary trees.
- Unordered containers are used to implement the hash tables.

13.3.1 Sequence containers

In this section, we will see some examples of the sequence containers.

13.3.1.1 Vector

We already saw an example of vector in Listing 13.3, where we have stored the integers in the vector.

A vector is a dynamic array, where we can directly access the elements using 'index'. Note that the insertion or elimination of the element from the 'end' is faster than the other location, as for the other location, the shifting of the elements are required to make a space for new value. below is the another example of vector where the square-roots of the numbers are stored in vector,

Listing 13.4: Vectors

```cpp
// vector_ex.cpp

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

int main(){
    vector<double> vec;  // by default : size = 0

    // append elements
    for (int i=0; i<4; i++)
        vec.push_back(sqrt(i)); // sqrt(i)

    // print elements
    for (int i=0; i<vec.size(); i++)
       cout << vec[i] << endl;

   return 0;
}
```

Below is the output of above code,

```
$ g++ vector_ex.cpp -o out
$ ./out
0
1
1.41421
1.73205
```

13.3.1.2 Array

The size of the vector changes, when we insert or delete an element to it. On the other hand, the ‘arrays’ have the fix sizes and we can only modify the ‘values’ of the elements in the array (not the size of array).

Listing 13.5: Array

```cpp
// array_stl_ex.cpp

#include <iostream>
#include <array>

using namespace std;

int main(){
    array<int, 5> arr = {3, 4}; // int-array of size 5

    // print the values
    for (int i=0; i<arr.size(); i++)
        cout << i << ": " << arr[i] << endl;

    return 0;
}
```

Below is the output of above code,

Note: Arrays are available in ISO C++ 2011 standard, therefore -std=c++11 is used for compilation.

```
$ g++ -std=c++11 array_stl_ex.cpp -o out
$ ./out
0: 3
1: 4
2: 0
3: 0
4: 0
```

13.3.1.3 Deque (double ended queues)

The 'deque' is the dynamic array that can grow in both direction. Therefore the 'insertion' and 'deletion' is faster at both the ends.

Listing 13.6: Deque

```cpp
// deque_ex.cpp

# include <iostream>
#include <deque>

using namespace std;

int main(){
    deque<int> deq = {1, 2}; // initialized deque
    cout << "size: " << deq.size() << endl;

    deq.push_front(10); // insert at the front
    deq.push_back(20); // insert in the end

    for (int i=0; i<deq.size(); i++)
        cout << deq[i] << ", ";

    cout << endl;

    return 0;
}
```

Below is the output of above code,

```
$ g++ -std=c++11 deque_ex.cpp -o out
$ ./out
size: 2
10, 1, 2, 20,
```

13.3.1.4 List

A list is implemented using 'double link list' of elements i.e. each element has it's own memory space. Hence the index-access is not possible in lists. Further, it does not provide a random access i.e. if we want to access the 4th element that we need to go through the 3rd element. Therefore, the access-speed is slower than the 'array' and 'vector'. For insertion and deletion, only two links needed to be changes (instead of shifting the whole array), therefore insertion and deletion is faster in lists.

Note:

- Lists do not provide the index and random access.

- Access to the elements is slower that ‘array’ and ‘vector’.
- But, insertion and deletion is faster than the ‘array’ and ‘vector’.

Listing 13.7: List

```
1  // list_stl_ex.cpp
2
3  #include <iostream>
4  #include <list>
5
6  using namespace std;
7
8  int main(){
9      list<int> lst;
10
11     for (int i=0; i<5; i++)
12         lst.push_back(i*2);
13
14     cout << "Print using for loop" << endl;
15     for (int l : lst) // for l in list:
16         cout << l << " "; // print(l)
17     cout << endl;
18
19     cout << "Print using while loop" << endl;
20     while (! lst.empty()){
21         cout << lst.front() << " "; // print first element
22         lst.pop_front(); // remove the first element
23     }
24     cout << endl;
25
26     // // index and random access is not allowed in list
27     // for (int i=0; i<lst.size(); i++)
28         // cout << lst[i];
29
30     return 0;
31 }
```

Tip:

- The ‘for loop’ at Line 15 is known as ‘**range-based for loop**’, which can be used to loop through any kind of data.
- Further, we can use ‘auto’ in Line 15 (instead of int) , which can determine the type automatically,

```
for (auto l : lst) // for l in list:
            cout << l << " "; // print(l)
```

Below is the output of above code,

```
$ g++ -std=c++11 -o out list_stl_ex.cpp
$ ./out
Print using for loop
0 2 4 6 8
Print using while loop
0 2 4 6 8
```

13.3.1.5 Forward list

Forward list is the ‘single link list’, therefore ‘push_back’ operation will not work as show in Line 12 of Listing Listing 13.8,

Listing 13.8: Forward list

```cpp
// forward_list_stl_ex.cpp

#include <iostream>
#include <forward_list>

using namespace std;

int main(){
    forward_list<int> lst;

    for (int i=0; i<5; i++)
        // lst.push_back(i*2); // push_back will not work
        lst.push_front(i*2);

    cout << "Print using for loop" << endl;
    for (int l : lst) // for l in list:
        cout << l << " "; // print(l)
    cout << endl;

    cout << "Print using while loop" << endl;
    while (! lst.empty()){
        cout << lst.front() << " "; // print first element
        lst.pop_front(); // remove the first element
    }
    cout << endl;

    // // index and random access is not allowed in list
    // for (int i=0; i<lst.size(); i++)
        // cout << lst[i];

    return 0;
}
```

Below is the output of above code,

```
$ g++ -std=c++11 -o out forward_list_stl_ex.cpp
$ ./out
Print using for loop
8 6 4 2 0
Print using while loop
8 6 4 2 0
```

13.3.2 Associative containers

Associative containers **sort** the elements automatically based on one of the two criterion i.e. ‘values’ and ‘key-values’.

Since the items are sorted, therefore ‘push_back’ and ‘push_front’ operations are not allowed in associative containers. The ‘insert()’ operation can be used for inserting item in the containers.

13.3.2.1 Set and multiset

- **set**: a set is collection of unique-values. If the collection have multiple same values, then only one value will be stored and rest will be ignored.
- **multiset**: same as set, but can have duplicate values.

Below is an example of set and multiset, Listing 13.9: Set and multiset

```cpp
// multiset_ex.cpp

#include <iostream>
#include <string>
#include <set>

using namespace std;

int main(){
    set<string> names { "Meher", "Krishna", "Patel", "Meher"};
    multiset<string> names_mul { "Meher", "Krishna", "Patel", "Meher"};

    names.insert("mekrip"); // insert new name

    cout << "set" << endl; // set
    for (string name : names)
        cout << name << " ";
    cout << endl << endl;

    cout << "multiset" << endl; // multiset
    for (string name : names_mul)
        cout << name << " ";
    cout << endl;

    return 0;
}
```

Below is the output of above code. Note that the 'set' contains "Meher" once only, whereas multiset contains it twice.

Note: Names are ordered, first based on 'lowercase/uppercase' and then by alphabets'.

```
$ g++ -std=c++11 -o out multiset_ex.cpp
$ ./out
set
Krishna Meher Patel mekrip

multiset
Krishna Meher Meher Patel
```

13.3.2.2 Map and multimap

- **map**: a map is the collection of key-value pairs. The elements are sorted based on the keys (instead of values). It can be used as **associative arrays** which have non-integer index.
- **multimap**: same as 'map', but elements with duplicate keys are allowed. This can be used as '**dictionary**'.

Below is an example of Map and multimap,

Listing 13.10: Map and multimap

```cpp
// multimap_ex.cpp

#include <iostream>
#include <string>
#include <map>

```

(continues on next page)

(continued from previous page)

```
using namespace std;

int main(){
    map<int, string> ranks;
    multimap<int, string> ranks_mul;

    ranks = {
        {2, "Meher"},
        {3, "Patel"},
        {1, "Krishna"},
        {1, "Mekrip"}, // duplicate key will be igonred
        {4, "Krishna"}
    };

    // map
    cout << "map" << endl;
    for (auto rank : ranks)
        // first : key,   second : value
        cout << rank.first << ": " << rank.second << endl;
    cout << endl;

    // multimap
    ranks_mul = {
        {2, "Meher"},
        {3, "Patel"},
        {1, "Krishna"},
        {1, "Mekrip"}, // duplicate key will be igonred
        {4, "Krishna"}
    };


    cout << "multimap" << endl;
    for (auto rank : ranks_mul)
        // first : key,   second : value
        cout << rank.first << ": " << rank.second << endl;
    cout << endl;

    return 0;
}
```

Below is the output of above code. Note that the 'map' contains the key "1" once only, whereas multimap contains it twice.

```
$ g++ -std=c++11 -o out multimap_ex.cpp
$ ./out
map
1: Krishna
2: Meher
3: Patel
4: Krishna

multimap
1: Krishna
1: Mekrip
2: Meher
3: Patel
4: Krishna
```

13.3.3 Unordered containers

Unordered containers do not have any order. Rest of the working is same as 'associative containers', as show in below two examples,

13.3.3.1 Unordered set and multiset

Below is an example of unordered set and multiset,

Listing 13.11: Unordered set and multiset

```cpp
// unorder_multiset.cpp

#include <iostream>
#include <string>
#include <unordered_set>

using namespace std;

int main(){
    unordered_set<string> names { "Meher", "Krishna", "Patel", "Meher"};
    unordered_multiset<string> names_mul { "Meher", "Krishna", "Patel", "Meher"};

    names.insert("mekrip"); // insert new name

    cout << "set" << endl; // set
    for (string name : names)
        cout << name << " ";
    cout << endl << endl;

    cout << "multiset" << endl; // multiset
    for (string name : names_mul)
        cout << name << " ";
    cout << endl;

    return 0;
}
```

Below is the output of above code,

```
$ g++ -std=c++11 -o out unorder_multiset.cpp
$ ./out
set
mekrip Patel Krishna Meher

multiset
Patel Krishna Meher Meher
```

13.3.3.2 Unordered map and multimap

Below is an example of unordered map and multimap,

Listing 13.12: Unordered map and multimap

```cpp
// unorder_multimap.cpp

#include <iostream>
#include <string>
#include <unordered_map>

```

(continues on next page)

(continued from previous page)

```cpp
using namespace std;

int main(){
    unordered_map<int, string> ranks;
    unordered_multimap<int, string> ranks_mul;

    ranks = {
        {2, "Meher"},
        {3, "Patel"},
        {1, "Krishna"},
        {1, "Mekrip"}, // duplicate key will be igonred
        {4, "Krishna"}
    };

    // map
    cout << "map" << endl;
    for (auto rank : ranks)
        // first : key,   second : value
        cout << rank.first << ": " << rank.second << endl;
    cout << endl;

    // multimap
    ranks_mul = {
        {2, "Meher"},
        {3, "Patel"},
        {1, "Krishna"},
        {1, "Mekrip"}, // duplicate key will be igonred
        {4, "Krishna"}
    };


    cout << "multimap" << endl;
    for (auto rank : ranks_mul)
        // first : key,   second : value
        cout << rank.first << ": " << rank.second << endl;
    cout << endl;

    return 0;
}
```

Below is the output of above code,

```
$ g++ -std=c++11 -o out unorder_multimap.cpp
$ ./out
map
4: Krishna
1: Krishna
3: Patel
2: Meher

multimap
4: Krishna
1: Mekrip
1: Krishna
3: Patel
2: Meher
```

13.4 Iterators

All containers provide same set of iterators. The following two iterators are the most important iterators,

- **begin()** : returns an iterator to the start of the vector.
- **end()** : returns an iterator to the end of the the vector.

Warning: The 'end()' iterator does not represent the position of the last element, but the position after the last element; hence '!=' is used at Line 18 of Listing 13.13 (instead of '==' and '<=').

We already see the example of iterator in Listing 13.3 using 'for' loop using 'size()' option. In the below code, the Listing 13.3 is reimplemented using 'for' loop with 'begin()' and 'end()' iterators,

Important: It is better to iterate using 'begin()' and 'end()' option than the 'size()' option as,

- 'begin()' and 'end()' method can be applied to all type of containers.
- 'begin()' and 'end()' method is faster than the 'size()' method.

Also in 'C++ 11', we can use 'range based for loop' for iterations as shown in Listing 13.7.

Listing 13.13: Iterators

```
1  // iterator_ex.cpp
2
3  #include <iostream>
4  #include <vector>
5
6  using namespace std;
7
8  int main() {
9      vector<int> v_int; // container : vector of type 'int'
10     int i = 0;
11
12     for (int i=0; i<5; i++) // int i : local 'i'
13         v_int.push_back(i);
14
15     // iterator
16     vector<int>::iterator vi;
17
18     for (vi = v_int.begin(); vi != v_int.end(); ++vi)
19        cout << *vi << " ";
20
21     cout << endl;
22
23     return 0;
24 }
```

Note: We can use 'const_iterator' instead of 'iterator' at line 16, which provides **read only** access for the container's element.

```
// iterator
vector<int>::const_iterator vi;

// below will be invalid
// *vi = 3; // as it is read only
```

There are 5 types of iterators,

- Random access iterator
- Bidirectional iterator
- Forward iterator
- Input iterator
- Output iterator

13.4.1 Random access iterator

Random access iterators allows the random access of the elements in the containers. Listing 13.14 shows the example of a random access iterator. These iterators can be increment or decrement using mathematical operations (Lines 17 and 21). Also, these iterators can be compared with each others (Line 26).

Note:

- Random access iterators are provided by '**vector**', '**deque**' and '**array**'.
- Pre-increment (Line 21) is faster than post-increment, as post-increment stores the value in the temporary variable before incrementing it.

Listing 13.14: Random access iterator

```
1  // random_iter.cpp
2  
3  #include <iostream>
4  #include <vector>
5  #include <algorithm> // 'find' is algorithm
6  
7  using namespace std;
8  
9  int main(){
10     vector<int> v_int = {12, 222, 11, 14, 25, 19}; // container
11 
12     vector<int>::iterator vi = v_int.begin(); // iterator  vi -> 12
13     vector<int>::iterator vi2 = find(vi, v_int.end(), 222); // iterator  vi -> 222
14     cout << *vi << endl; //
15 
16     // increment iterator by 3
17     vi = vi + 3;  // vi -> 14
18     cout << *vi << endl;
19 
20     // ++vi is faster than vi++
21     ++vi; // vi -> 25
22     cout << *vi << endl;
23 
24 
25     cout << *vi2 << endl; // vi2 -> 222
26     if (vi > vi2) // true as vi is pointing to 25, which comes after 222
27        vi2 = vi; // vi2 ->25
28    cout << *vi2 << endl;
29 
30     return 0;
31 }
```

13.4.2 Bidirectional iterator

Note:

- Bidirectional iterators are provided by **list**, **set/multiset** and **map/multimap**.

- Bidirectional iterator can be increment and decremented using 'vi++' and 'vi–' etc.
- Increment and decrement is not allowed using mathematical operations i.e. 'vi + 3' is invalid.

13.4.3 Forward iterator

Note:

- Forward iterators are provided by 'forward_list' only.
- These iterators can only be incremented (not decremented).
- Also, these can not be incremented by mathematical operations i.e. 'vi + 3' is invalid.

13.4.4 Input iterator

Input iterators read and process the values while iterating **forward**. We can read the value using input iterators, but can not write the value. Note that, this iterator can not move backward.

```
int i = *itr;
```

13.4.5 Output iterator

Output iterators output the value while iterating **forward**. We can write the value to the output iterator but can not read the value. Note that, this iterator can not move backward.

```
int *itr = 20;
```

13.5 Iterator functions

STL provides some functions as well for iterators as shown in Lines 17 and 21 of Listing 13.15.

Note: The function 'advance (Line 17)' is quite useful for incrementing the 'non random access iterators'.

Listing 13.15: Iterator functions

```
1  // itr_function.cpp
2
3  #include <iostream>
4  #include <vector>
5  #include <algorithm>
6
7  using namespace std;
8
9  int main(){
10     vector<int> v_int = {12, 222, 11, 14, 25, 19}; // container
11
12     vector<int>::iterator vi = v_int.begin(), vi3; // iterator  vi -> 12
13     vector<int>::iterator vi2 = find(vi, v_int.end(), 222); // iterator  vi -> 222
14     cout << *vi << endl; //
15
16     // increment iterator by 3; good for 'non random access iterator'
17     advance(vi, 3); // vi -> 14
```

(continues on next page)

(continued from previous page)

```
    cout << *vi << endl;

    // distance between two iterator
    cout << distance(vi, vi2) << endl; // -2


    return 0;
}
```

13.6 Algorithms

The algorithms are used on containers to perform various operations such as sorting, searching and transforming etc. as shown in this section.

In the below code, the ‘min_element’, ‘max_element’, ‘find’, ‘sort’ and ‘reverse’ algorithms are used on a integer-vector,

Listing 13.16: Algorithms

```
// algorithm_ex.cpp

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main(){
    vector<int> vec = { 10, 12, 3, 4, 15};

    // min_pos is the iterator
    vector<int>::iterator min_pos = min_element( vec.begin(), vec.end() );
    cout << "minimum value: " << *min_pos << endl;

    // 'auto' is easy to use
    auto max_pos = max_element( vec.begin(), vec.end() );
    cout << "maximum value: " << *max_pos << endl;

    sort( vec.begin(), vec.end());
    cout << "minimum value: " << *vec.begin() << endl << endl;

    cout << "print the value of vector" << endl;
    for (int v : vec)
        cout << v << " ";
    cout << endl << endl;

    // find the position of first 10
    vector<int>::iterator pos_10 = find( vec.begin(), vec.end(), 10 );

    // reverse the elements from the element 10
    reverse( pos_10, vec.end());
    // print values
    cout << "print the value of vector (after reverse)" << endl;
    for (int v : vec)
        cout << v << " ";
    cout << endl << endl;

    // note that pos_10 is still the 3rd position
    // maximum value - exclude the 3rd position
```

(continues on next page)

(continued from previous page)

```
    cout << "maximum value (exclude 3rd position): "
        << *max_element( vec.begin(), pos_10) << endl;

    // maximum value - including the 3rd position
    cout << "maximum value (include 3rd position): "
       << *max_element( vec.begin(), ++pos_10) << endl;

    return 0;
}
```

Below is the output of above code,

```
$ g++ -std=c++11 -o out algorithm_ex.cpp
$ ./out
minimum value: 3
maximum value: 15
minimum value: 3

print the value of vector
3 4 10 12 15

print the value of vector (after reverse)
3 4 15 12 10

maximum value (exclude 3rd position): 4
maximum value (include 3rd position): 15
```

13.7 Tables

In previous sections, we saw that the functions may not available to all kinds of containers e.g. the function 'push_front' is available for 'deque' but not for 'vector'.

In this section, few tables are added which show the various functions available for different types of container.

Note: Following notations are used in the tables,

- 'ctr' : container
- 'itr' : iterator
- 'pos' : position
- 'val' : value
- 'idx' : index
- 'ctr.erase(itr1 [,itr2])' : parameters in the bracket (i.e. [, itr2]) are optional

13.7.1 Function common to all container

Table 13.1 shows the functions which are common to all the containers.

Table 13.1: Functions common to all types of container

Function	Description
ctr.begin()	returns an iterator to the first element of container 'ctr'
ctr.end()	returns an iterator to the last element of the container 'ctr'
ctr.rbegin()	reverse begin (points to last element)
ctr.rend()	reverse end (points to first element)
ctr.cbegin()	returns a **constant** iterator to the first element of 'ctr'
ctr.cend()	returns a **constant** iterator to the last element of 'ctr'
ctr.empty()	return 'true' if 'ctr' is empty, otherwise 'false'
ctr.size()	return the number of elements in 'ctr'
ctr.insert(pos, val)	insert value 'val' at position 'pos'
ctr.erase(itr1 [, itr2])	erase element from position 'itr1' to 'itr2-1'.
ctr.max_size()	return maximum numbers of elements which can be inserted in 'ctr'
ctr1.swap(ctr2)	swap the contents of 'ctr1' and 'ctr2'
swap(ctr1, ctr2)	swap the contents of 'ctr1' and 'ctr2'
ctr.clear()	delete all elements and set the size of container to 0 (not for array)
ctr1 = ctr2	copy the element of ctr2 in ctr1
ctr1 == ctr2,	return 'true' if equal, otherwise 'false'
ctr1 != ctr2	return 'true' if not equal, otherwise 'false'
ctr1 < ctr2 and others	not applicable for unordered container

13.7.2 Vector operations

Some of the possible vector operations are listed in Table 13.2,

Table 13.2: Vector operations

Function	Description
Declaration	Declaration and initialization of array 'v'
vector<eType> v	create vector 'v' of size '0' and type 'eType'
vector<eType> v(n)	create vector 'v' of size 'n' (initialize with 0)
vector<eType> v(n, val)	create vector 'v' of size 'n' (initialize with 'val')
vector<eType> v(v2)	create a copy 'v' of vector 'v2'
vector<eType> v = v2	create a copy 'v' of vector 'v2'
vector<eType> v = {1, 2}	create 'v' and initialize with values (1, 2)
Assignment	
v1 = v2	copy v2 to v1
v.assign(n, val)	assign 'n' elements of value 'val'
v1.swap(v2)	swap the values of vector 'v1' and 'v2'
swap(v1, v2)	swap the values of vector 'v1' and 'v2
Access	
v[idx]	returns the element at 'idx' without range-check
v.at(idx)	return element at idx, and throws exception if out of range
v.front()	return first element (no check for existence of first element
v.back()	return last element (no check for existence of last element
Iterator	
v.begin()	iterator to first element
v.end()	iterator to last element
v.cbegin()	constant iterator to first element (i.e. read only)
v.cend()	constant iterator to last element
v.rbegin()	reverse begin (points to last element)
v.rend()	reverse end (points to first element)
v.crbegin()	constant reverse iterator (points to last element)
v.crend()	constant reverse iterator (points to first element)

Continued on next page

Table 13.2 – continued from previous page

Function	Description
Checks	
v.empty()	returns true if size=0
v.size()	return size of the vector
v.max_size()	return maximum size of the vector
v.capacity()	return maximum size without reallocation
v1 == v2, v1 !=v2 etc.	returns true if condition is met
Insert and remove	
v.push_back(val)	append value to end
v.pop_back()	remove value from the end
v.insert(pos, val)	instert 'val' at 'pos'
v.insert(pos, n, val)	insert 'n' copies 'val' at 'pos'
v.erase(pos)	remove element from 'pos'
v.erase(itr1, itr2)	delete element from location 'itr1' to 'itr2'
v.clear()	remove all the elements.

Some of the examples of vector operations are added below,

```
// vector_test.cpp

#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> v; // vector with no element
    vector<int> v1 = {10, 12}; // 10 12
    vector<int> v2(v1); // 10 12
    vector<int> v3 = v1; // 10 12
    vector<int> v4(3); // initialize with zeros 0 0 0
    vector<int> v5(3, 1); // 1 1 1

    for (auto i : v5)
        cout << i << " "; // 0, 0, 0
    cout << endl;

    cout << v1.size() << endl; // 2
    cout << v1.max_size() << endl; // 1073741823
    cout << v1.capacity() << endl; // 2

    return 0;
}
```

13.7.3 Deque

The operations in deque is same as vector operations (Table 13.2) with following changes,

- 'capacity()' option is **not** available in 'deque'.
- 'deque' allows insertion and deletion from the front as well using **push_front** and **pop_front** respectively.

13.7.4 Arrays operations

Table 13.3 shows the various ways to declare and initialize the arrays. Further, Listing 13.17 shows the examples of the functions shown in Table 13.3.

Table 13.3: Array operations

Function	Description
Declaration	Declaration and initialization of array ‘a’
array<eType, N> a	create array ‘a’ of size ‘N’ and type ‘eType’ with garbage values
array<eType, N> a(a2)	create a copy ‘a’ of array ‘a2’
array<eType, N> a = a2	create a copy ‘a’ of array ‘a2’
array<eType, N> a = {1, 2}	create ‘a’ and initialize with values (1, 2, 0, 0 . . .)
Assignment	
a = a2	copy a2 to a
a.fill(val)	assign ‘val’ to each element in array ‘a’
a1.swap(a2)	swap the values of array ‘a1’ and ‘a2’
swap(a1, a2)	swap the values of array ‘a1’ and ‘a2’
Access	
a[idx]	returns the element at ‘idx’ without range-check
a.at(idx)	return element at idx, and throws exception if out of range
a.front()	return first element (no check for existence of first element
a.back()	return last element (no check for existence of last element
Iterator	
a.begin()	iterator to first element
a.end()	iterator to last element
a.cbegin()	constant iterator to first element (i.e. read only)
a.cend()	constant iterator to last element
a.rbegin()	reverse begin (points to last element)
a.rend()	reverse end (points to first element)
a.crbegin()	constant reverse iterator (points to last element)
a.crend()	constant reverse iterator (points to first element)
Checks	
a.empty()	returns true if size=0
a.size()	return size of the array
a.max_size()	return maximum size of the array
a1 == a2, a1 !=a2 etc.	returns true if condition is met

Listing 13.17: Some examples of Array operations

```
// array_dec_init.cpp

#include <iostream>
#include <array>

using namespace std;

int main(){
    array<int, 3> a; // garbage values 134514972 1 65535
    array<int, 3> a1 = {10, 12}; // 10 12 0
    array<int, 3> a2(a1); // 10 12 0
    array<int, 3> a3 = a1; // 10 12 0

    for (auto i : a)
        cout << i << " ";
    cout << endl;

    // size and max_size have same values for array
    cout << a1.size() << endl; // 3
    cout << a1.max_size() << endl; // 3

    return 0;
}
```

13.8 More examples

In previous sections, we saw the various elements of the 'standard template library' i.e. containers, iterators and algorithms. In this section, we will see some more examples along with some new features of each element.

13.8.1 Iterating over the items

As mentioned before, we should use the 'iterators' or 'range based for loops' for iterating over the elements in the containers. The Listing 13.18 shows these two methods of iteration at Lines 19 and 28 respectively.

Note: Then 'range based for loop' is available in 'C++ 11', therefore we need to compile using 'C++ 11'

Listing 13.18: Correct ways of iterations

```
1  // vector_iter.cpp
2  
3  #include <iostream>
4  #include <vector>
5  #include <algorithm>
6  
7  using namespace std;
8  
9  int main(){
10     vector<int> v;
11     v.push_back(2);
12     v.push_back(-1);
13     v.push_back(5);
14 
15     vector<int>:: iterator itr_begin = v.begin(); // stores start position
16     vector<int>:: iterator itr_end = v.end(); // stores end position
17 
18     // print values in the container
19     for (vector<int>:: iterator itr=itr_begin; itr!=itr_end; ++itr){
20         cout << *itr << " ";  // 2 -1 5
21     }
22     cout << endl;
23 
24     // sort the values
25     sort(itr_begin, itr_end); // sort values from itr_begin and itr_end
26 
27     // use 'range based for loop'
28     for (int i : v)
29         cout << i << " "; // -1 2 5
30     cout << endl;
31 
32     return 0;
33 }
```

```
$ g++ -std=c++11 -o out vector_iter.cpp
$ ./out
2 -1 5
-1 2 5
```

13.8.2 Container's header

In Listing 13.19, all the headers of STL are added in Lines 3-14. Also, some features of vectors are shown i.e. 'at (Line 26)' 'clear (Line 29)', 'converting vector to array (Line 34)' and 'swap (Line 38)'.

Listing 13.19: List of headers in STL

```
// stl_headers.cpp

#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set> // set and multiset
#include <unordered_set> // unordered set and multiset
#include <map> // map and multimap
#include <unordered_map> // unordered map and multimap
#include <iterator>
#include <algorithm>
#include <numeric> // numeric algorithms
#include <functional> // functors

using namespace std;

int main(){
    vector<int> v = {2, 4, 1};
    vector<int> w = {2, 4, 1};

    // size of the vector
    cout << "size = " << v.size() << endl; // size = 3, index start with 0

    // access the individual elements
    cout << "v[2] = " << v.at(2) << endl; // v[2] = 1
    cout << "v[2] = " << v[2] << endl; // v[2] = 1

    v.clear(); // clear the vector
    cout << "size = " << v.size() << endl; // size = 0

    // vectors are the 'dynamic contiguous array'
    // save vector in array using pointer
    int* pw = &w[0];  // pointer to vector
    cout << "w[1] = " << pw[1] << endl; // w[1] = 4

    cout << "size (v, w): " << v.size() << ", " << w.size() << endl; // 0, 3
    v.swap(w); // save w in v; and v in w
    cout << "size after swap (v, w): " << v.size() << ", " << w.size() << endl; // 3, 0

    return 0;
}
```

Below is the output of above code,

```
$ g++ -std=c++11 -o out stl_headers.cpp
$ ./out
size = 3

v[2] = 1
v[2] = 1

size = 0

w[1] = 4

size (v, w): 0, 3
size after swap (v, w): 3, 0
```

13.8.3 List

In Listing 13.20, the insertion and deletion operation is performed on the double linked-list. Please see the location of iterator after 'insert (Line 25-26)', 'delete (Lines 31-34)' and 'slice (43-50)' operations. Read comments for more details.

Listing 13.20: Insertion and deletion in double link-list

```
 1  // link_list.cpp
 2  // double linked-list
 3
 4  #include <iostream>
 5  #include <list>
 6  #include <algorithm>
 7
 8  using namespace std;
 9
10  int main(){
11      list<int> l = {3, 5, 2, 1, 10}, l2 = {0, 1};
12
13      l.push_back(-2); // -2 will be added to end
14      l.push_front(14); // 14 will be added to front
15
16      // print values in the list
17      for (int i : l)
18          cout << i << " "; // 14 3 5 2 1 10 -2
19      cout << endl;
20
21      // find location of first 2
22      list<int>::iterator itr = find(l.begin(), l.end(), 2);  // itr -> 2
23
24      // insert 9 before 2
25      l.insert(itr, 9); // 14 3 5 9 2 1 10 -2,
26      cout << *itr << endl;  //                        itr -> 2
27
28      ++itr; // increment the itr (or itr++)
29      cout << *itr << endl;  //                        itr -> 1
30      // delete item at location itr i.e. 1
31      l.erase(itr); // 14 3 5 9 2 10 -2
32
33      // // itr is pointing to deleted item, therefore following line will generate error
34      // cout << *itr << endl;  // itr -> 1, pointing to delete item '1'
35      // itr++; // increment the itr (or itr++);
36      // cout << *itr << endl; // error
37
38      // print values in the list
39      for (int i : l)
40          cout << i << " "; // 14 3 5 9 2 10 -2
41      cout << endl;
42
43      // slice
44      // locate 2 in l
45      list<int>::iterator itr2 = find(l.begin(), l.end(), 2);  // itr -> 2
46      // insert from '2 to end' of l, to the beginning of l2
47      l2.splice(l2.begin(), l, itr2, l.end());
48      for (int i : l2)
49          cout << i << " "; // 2 10 -2 0 1
50      cout << endl;
51
52
53      return 0;
54  }
```

13.8.4 Set

In Listing 13.21, a set (i.e. associative container) is used, which stored the unique values in sorted order. The insert operation return two types of values i.e. 'iterator' and 'boolean' (see Lines 22-28). Also, in associative containers, the elements can be erased by providing the values directly as shown in Line 30 (without using the iterators).

Listing 13.21: Insertion and deletion in set

```
 1  // set_op_ex.cpp
 2  
 3  #include <iostream>
 4  #include <set>
 5  
 6  using namespace std;
 7  
 8  int main(){
 9      set<int> s={4, 2, 2};
10      for (int i : s)
11          // automatically sorted and duplicate removed
12          cout << i << " "; // 2 4 (not 4 2 2)
13      cout << endl;
14  
15      s.insert(1); // 1 2 4  i.e. automatically sorted
16      s.insert(12); // 1 2 4 12
17  
18      set<int>:: iterator itr; // iterator
19      itr = s.find(4); // find 4
20      cout << *itr << endl; // location of itr->4
21  
22      pair<set<int>::iterator, bool> ret;
23      ret = s.insert(2);
24  
25      // ret.first is the iterator
26      cout << "value entered: " << *ret.first << endl; // 2
27      // ret.second => 0: False (value not inserted), 1: True
28      cout << "value inserted: " << ret.second << endl; // 0 as already exist
29  
30      s.erase(2); // provide value of erase directly
31      for (int i : s)
32          cout << i << " "; // 1 4 12
33      cout << endl;
34  
35      return 0;
36  }
```

13.8.5 Multimap

Multimap can be used as dictionary as show in Listing 13.22, where a dictionary of 'name' is created,

Listing 13.22: Dictionary using multimap

```
// map_op_ex.cpp

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main(){
```

(continues on next page)

(continued from previous page)

```
    multimap<string, string> m; // multimap

    // pair: type is defined explicitly
    m.insert (pair<string, string>("name", "Meher")); // pair

    // make_pair: type is inferred automatically
    m.insert (make_pair("name", "Krishna"));  // make_pair

    for ( auto i : m)
        cout << i.first << ": " << i.second << endl;


    return 0;
}
```

Below is the output of above code,

```
$ g++ -std=c++11 -o out map_op_ex.cpp
$ ./out
name: Meher
name: Krishna
```

When a man is, through his desires, confronted with great suffering , he understands their true nature; so when such suffering comes, it should be welcome. Suffering may come in order to eliminate further suffering. Suffering has to come, when it is of use in purging the soul from it's desires.

–Meher Baba

Chapter 14

Object Oriented Programming

14.1 Introduction

Object oriented programming (OOP) increases the re-usability of the code. Also, the codes become more manageable than non-OOP methods. But, it takes proper planning, and therefore longer time, to write the codes using OOP method. In this chapter, we will learn following features of OOP along with examples.

- Class and Object
- Data encapsulation (or Data hiding)
- Inheritance
- Polymorphism
- Data abstraction
- Interface (Abstract class)

14.2 Class and object

A 'class' is user defined template which contains variables, constants and functions etc.; whereas an 'object' is the instance (or variable) of the class. In simple words, a class contains the structure of the code, whereas the object of the class uses that structure for performing various tasks, as shown in this section.

14.3 Create class and object

Class is created using keyword 'class' as shown in Line 6 of Listing 14.1 where the class 'Jungle' is created. As a rule, class name is started with uppercase letter, whereas function name is started with lowercase letter. The class 'Jungle' contains one **public** method i.e. 'welcomeMessage' at Line 10, which prints a message. The 'public' method can be accessed from outside the class. Then an object 'j' of the class Jungle is created at Line 16. Then the public method 'welcomeMessage' is called at Line 17, which prints the message as shown in Line 23.

Listing 14.1: Create class and object

```
1  // classObject1.cpp
2
3  #include <iostream>
4  using namespace std;
5
6  class Jungle{
7  public:  // to allow access to function 'welcomeMessage' outside the class
8
9      // welcome message
```

(continues on next page)

(continued from previous page)

```
    void welcomeMessage(){
        cout << "Welcome to Jungle";
    }
};

int main(){
    Jungle j;   // 'j' is object of class 'Jungle'
    j.welcomeMessage();  // accessing class-function

    return 0;
}

/* Outputs
Welcome to Jungle
*/
```

In Listing 14.2, the method 'welcomeMessage' has an input parameter i.e. 'name' (see Line 10). Next, a name is read from user using 'getline' at Line 18-19, Then the name is sent as input parameter while calling this the (see Line 22), which is printed by the method as shown in Line 28-28.

Listing 14.2: Method with input parameter

```
1  // classObject2.cpp
2
3  #include <iostream>
4  using namespace std;
5
6  class Jungle{
7  public:  // to allow access to function 'welcomeMessage' outside the class
8
9      // welcome message
10     void welcomeMessage(string name){
11         cout << "Welcome to Jungle " << name;
12     }
13 };
14
15 int main(){
16     string name;
17
18     cout << "Enter your name : ";
19     getline(cin, name);  // read name with spaces
20
21     Jungle j;   // 'j' is object of class 'Jungle'
22     j.welcomeMessage(name);  // accessing class-function
23
24     return 0;
25 }
26
27 /* Outputs
28 Enter your name : Meher Krishna
29 Welcome to Jungle Meher Krishna
30 */
```

In Listing 14.2, the 'name' was provided from the 'main' function, but **did not store in the class**, therefore it can not be used by other methods in the class (if exist). In Listing 14.3, the name provided by the 'main' function is stored in a **private** variable 'visitorName' (Line 8). Then, two **public** methods, i.e. 'setVisitorName' and 'getVisitorName', are provided to set (Line 35) and get (Line 24) the 'name' of the visitor.

Listing 14.3: Create class and object

```
1  // classObject2.cpp
2
3  #include <iostream>
4  using namespace std;
5
6  class Jungle{
7  private: // not accessible outside the class
8      string visitorName;
9
10 public:  // to allow access to function 'welcomeMessage' outside the class
11
12     // setVisitorName is accessible outside the class, which will set the visitor name
13     void setVisitorName(string name){
14         visitorName = name;
15     }
16
17     // function to retrieve the visitorName as it is not accessible directly
18     string getVisitorName(){
19         return visitorName;
20     }
21
22     // welcome message
23     void welcomeMessage(){
24         cout << "Welcome to Jungle " << getVisitorName();
25     }
26 };
27
28 int main(){
29     string name;
30
31     cout << "Enter your name : ";
32     getline(cin, name);  // read name with spaces
33
34     Jungle j;   // 'j' is object of class 'Jungle'
35     j.setVisitorName(name); // set the visitor name
36     j.welcomeMessage();  // accessing class-function
37
38     return 0;
39 }
40
41 /* Outputs
42 Enter your name : Meher Krishna
43 Welcome to Jungle Meher Krishna
44 */
```

Note: Unlike Python, in C++ the class variables should defined as 'private' attributes. This is known as **data hiding**.

Data hiding is required in C++ and Java etc. as direct access can be a serious problem here and can not be resolved. In Python, data is not hidden from user and we have various methods to handle all kind of situations as shown in the Python tutorials.

14.4 Constructor

Whenever, the object of a class is create then all the attributes and methods of that class are attached to it; and the constructor (Lines 13-15) is executed automatically. **Note that, the constructor and the class have**

the same name. Here, the constructor contains one variable i.e. ‘name’ (Line 13). Therefore, when the object ‘j’ is created at Line 35, the value ‘Meher Krishna’ will be assigned to parameter ‘name’ and finally saved in ‘visitorName’ by the method ‘setVisitorName’ (Line 18) through constructor (Line 14).

Listing 14.4: Constructor

```
1  // constructorEx.cpp
2  
3  #include <iostream>
4  using namespace std;
5  
6  class Jungle{
7  private: // not accessible outside the class
8      string visitorName;
9  
10 public:  // to allow access to function 'welcomeMessage' outside the class
11 
12     // constructor : automatically called at the time of object-creation
13     Jungle(string name){
14         setVisitorName(name);
15     }
16 
17     // setVisitorName is accessible outside the class, which will set the visitor name
18     void setVisitorName(string name){
19         visitorName = name;
20     }
21 
22     // function to retrieve the visitorName as it is not accessible directly
23     string getVisitorName(){
24         return visitorName;
25     }
26 
27     void welcomeMessage(){
28         cout << "Welcome to Jungle " << getVisitorName();
29     }
30 };
31 
32 int main(){
33     string name;
34 
35     Jungle j("Meher Krishna");    // 'j' is object of class 'Jungle'
36     j.welcomeMessage();  // accessing class-function
37 
38     return 0;
39 }
40 
41 /* Outputs
42 Enter your name : Meher Krishna
43 */
```

14.5 Inheritance

Suppose, we want to write a class ‘RateJungle’ in which visitor can provide ‘rating’ based on their visiting-experience. If we write the class from the starting, then we need define attribute ‘visitorName’ again; which will make the code repetitive and unorganizable, as the visitor entry will be at multiple places and such code is more prone to error. With the help of inheritance, we can avoid such duplication as shown in Listing 14.5; where class Jungle is inherited at Line 33 by the class ‘RateJungle’. Now, when the object ‘r’ of class ‘RateJungle’ is created at Line 62 of Listing 14.5, then this object ‘r’ will have the access to ‘visitorName’ as well (which is in the parent class).

Note: Two constructors are provided in Listing 14.5. The first constructor (Line 39) has only one parameter i.e. 'name', whereas the second constructor (Line 44) has two parameters i.e. 'name' and 'rating'. Therefore, if we provide only one parameter (Line 62) then first constructor will be invoked; whereas if we provide two constructor (Line 63), then second constructor will be invoked.

Listing 14.5: Inheritance

```
1  // InheritanceEx.cpp
2  
3  #include <iostream>
4  using namespace std;
5  
6  class Jungle{
7  private: // not accessible outside the class
8      string visitorName;
9  
10 public:  // to allow access to function 'welcomeMessage' outside the class
11 
12     // constructor : automatically called at the time of object-creation
13     Jungle(string name){
14         setVisitorName(name);
15     }
16 
17     // setVisitorName is accessible outside the class, which will set the visitor name
18     void setVisitorName(string name){
19         visitorName = name;
20     }
21 
22     // function to retrieve the visitorName as it is not accessible directly
23     string getVisitorName(){
24         return visitorName;
25     }
26 
27     void welcomeMessage(){
28         cout << "Welcome to Jungle " << getVisitorName();
29     }
30 };
31 
32 
33 class RateJungle : public Jungle{
34 private:
35     int feedback;
36     string name;
37 public:
38     // constructor 1
39     RateJungle(string name) : Jungle(name){
40         feedback = 0;  // set feedback to zero by default
41     }
42 
43     // constructor 2
44     RateJungle(string name, int value) : Jungle(name){
45         feedback = value;  // set feedback to zero by default
46     }
47 
48     void setFeedback(int value){
49         feedback = value;
50     }
51 
52     void printRating(){
53         cout << "Thanks " << getVisitorName() << endl;
```

(continues on next page)

(continued from previous page)

```
54         cout << "Your feedback is set as : " << feedback << endl;
55     }
56 };
57
58
59 int main(){
60     string name;
61
62     RateJungle r("Meher");    // constructor 1 will be initialized
63     RateJungle s("Krishna", 3);    // constructor 2 will be initialized
64
65     // Feedback is set to 0 by constructor
66     r.printRating();   // print the feedback value
67
68     r.setFeedback(2); // provide feedback value
69     r.printRating();   // print the feedback value
70
71     s.printRating();   // print the feedback value
72
73     return 0;
74 }
75
76 /* Outputs
77 Thanks Meher
78 Your feedback is set as : 0
79
80 Thanks Meher
81 Your feedback is set as : 2
82
83 Thanks Krishna
84 Your feedback is set as : 3
85
86 */
```

In Listing 14.6, an empty constructor is added at Line 54, for setting the value of the object ‘p’ (Line 74) through Line 83. **Note that, since we added empty constructor in child class, therefore we need to add in parent class as well as shown in Line 18.**

Listing 14.6: Inheritance

```
1  // InheritanceEx2.cpp
2
3  #include <iostream>
4  using namespace std;
5
6  class Jungle{
7  private: // not accessible outside the class
8      string visitorName;
9
10 public:  // to allow access to function 'welcomeMessage' outside the class
11
12     // constructor : automatically called at the time of object-creation
13     Jungle(string name){
14         setVisitorName(name);
15     }
16
17     // empty constructor
18     Jungle(){ // it is required as child-class is using empty constructor
19
20     }
21
```

(continues on next page)

(continued from previous page)

```
    // setVisitorName is accessible outside the class, which will set the visitor name
    void setVisitorName(string name){
        visitorName = name;
    }

    // function to retrieve the visitorName as it is not accessible directly
    string getVisitorName(){
        return visitorName;
    }

    void welcomeMessage(){
        cout << "Welcome to Jungle " << getVisitorName();
    }
};


class RateJungle : public Jungle{
private:
    int feedback;
    string name;
public:
    // constructor 1
    RateJungle(string name) : Jungle(name){
        feedback = 0;  // set feedback to zero by default
    }

    // constructor 2
    RateJungle(string name, int value) : Jungle(name){
        feedback = value;  // set feedback to zero by default
    }

    // constructor 3 : empty constructor
    RateJungle(){  // empty constructor in parent class is compulsory

    }

    void setFeedback(int value){
        feedback = value;
    }

    void printRating(){
        cout << "Thanks " << getVisitorName() << endl;
        cout << "Your feedback is set as : " << feedback << endl;
    }
};


int main(){
    string name;

    RateJungle r("Meher");   // constructor 1 will be initialized
    RateJungle s("Krishna", 3);   // constructor 2 will be initialized
    RateJungle p;

    // Feedback is set to 0 by constructor
    r.printRating();  // print the feedback value
    r.setFeedback(2); // provide feedback value
    r.printRating();  // print the feedback value

    s.printRating();  // print the feedback value

```

(continues on next page)

(continued from previous page)

```
    p.setVisitorName("Patel");  // constructor 3 is used
    p.setFeedback(5);
    p.printRating();

    return 0;
}

/* Outputs
Thanks Meher
Your feedback is set as : 0
Thanks Meher
Your feedback is set as : 2

Thanks Krishna
Your feedback is set as : 3

Thanks Patel
Your feedback is set as : 5
*/
```

14.6 Polymorphism

In OOP, we can use same name for methods and attributes in different classes; the methods or attributes are invoked based on the object type; e.g. in Listing 14.7, the method ‘scarySound’ is used for class ‘Animal’ and ‘Bird’ at Lines 8 and 15 respectively. Then object of these classes are created at Line 26-27. Finally, method ‘scarySound’ is invoked at Lines 29-30; here Line 29 is the object of class Animal, therefore method of that class is invoked and corresponding message is printed. Similarly, Line 30 invokes the ‘scaryMethod’ of class Bird and corresponding line is printed.

Listing 14.7: Polymorphism

```
 1  // PolymorphismEx.cpp
 2
 3  #include <iostream>
 4  using namespace std;
 5
 6  class Animal{
 7  public:
 8      void scarySound(){
 9          cout << "Animals are running away due to scary sound." << endl;
10      }
11  };
12
13  class Bird{
14  public:
15      void scarySound(){
16          cout << "Birds are flying away due to scary sound." << endl;
17      }
18  };
19
20  // empty class
21  class Insect{
22  };
23
24  int main(){
25      Animal a;
26      Bird b;
27      Insect i;
```

(continues on next page)

(continued from previous page)

```

    a.scarySound();
    b.scarySound();

    return 0;
}

/* Outputs
Animals are running away due to scary sound.
Birds are flying away due to scary sound.
*/
```

14.7 Abstract class

Abstract classes are the classes which contains one or more abstract method; and abstract methods are the methods which does not contain any implementation, but the child-class need to implement these methods otherwise error will be reported. In this way, we can force the child-class to implement certain methods in it. We can define, abstract classes and abstract method using keyword 'virtual void', as shown in Line 27 of Listing 14.8. Since, 'scarySound' is defined as abstract method at Line 27, therefore it is compulsory to implement it in all the subclasses.

Note: Look at the class 'Insect' in Listing 14.7, where 'scarySound' was not defined but code was running correctly; but now the 'scarySound' is abstract method, therefore it is compulsory to implement it, as done in Line 16 of Listing 14.8.

Listing 14.8: Abstract class

```
1  // abstractClassEx.cpp
2
3  #include <iostream>
4  using namespace std;
5
6  class Jungle{
7  private: // not accessible outside the class
8      string visitorName;
9
10 public:  // to allow access to function 'welcomeMessage' outside the class
11
12     // setVisitorName is accessible outside the class, which will set the visitor name
13     void setVisitorName(string name){
14         visitorName = name;
15     }
16
17     // function to retrieve the visitorName as it is not accessible directly
18     string getVisitorName(){
19         return visitorName;
20     }
21
22     // welcome message
23     void welcomeMessage(){
24         cout << "Welcome to Jungle " << getVisitorName();
25     }
26
27     virtual void scarySound() = 0;
28 };
29
```

(continues on next page)

(continued from previous page)

```cpp
class Animal : public Jungle{
public:
    void scarySound(){
        cout << "Animals are running away due to scary sound." << endl;
    }
};

class Bird : public Jungle{
public:
    void scarySound(){
        cout << "Birds are flying away due to scary sound." << endl;
    }
};

// Now, it is compulsory to define 'scarySound' in Insect as well
class Insect : public Jungle{
public:
    void scarySound(){
        cout << "Insects do not care about scary sound." << endl;
    }
};

int main(){
    Animal a;
    Bird b;
    Insect i;

    a.scarySound();
    b.scarySound();
    i.scarySound();

    return 0;
}

/* Outputs
Animals are running away due to scary sound.
Birds are flying away due to scary sound.
Insects do not care about scary sound.
*/
```

When a man is, through his desires, confronted with great suffering , he understands their true nature; so when such suffering comes, it should be welcome. Suffering may come in order to eliminate further suffering. Suffering has to come, when it is of use in purging the soul from it's desires.

–Meher Baba

Chapter 15

Exception handling

15.1 Introduction

In this chapter, examples of exceptional handling are shown.

15.2 Type checking

Listing 15.1: ype checking

```
// typeChecking.cpp

#include <iostream>
using namespace std;

int main(){
    int a;
    cin >> a;
    // cin.fail() : fail if type is not correct
    if (cin.fail()){
        cout << "Wrong input. Please enter one integer\n";
        std::cin.clear();
        std::cin.ignore(256,'\n');   // ignore the line change
        std::cin >> a;
    }

    cout << "a = " << a;
    return 0;
}
```

Listing 15.2: Type checking

```
// typeChecking2.cpp

#include <iostream>
using namespace std;

int main(){
    float a;  // a must be integer only
    cin >> a;
    cout << a;
    // cin.fail() : fail if type is not correct
    // static_cast<int>(a) != a : check if a is integer or not
```

(continues on next page)

(continued from previous page)

```
    if (cin.fail() || (static_cast<int>(a) != a) ){
        cout << "Wrong input. Please enter one integer\n";
        std::cin.clear();
        std::cin.ignore(256,'\n');   // ignore the line change
        std::cin >> a;
    }
    return 0;
}
```

15.3 Exceptional handling

Listing 15.3: Exceptional handling

```
// exceptionEx.cpp

#include <iostream>
using namespace std;

float division(float, float);

float division(float a, float b){
    if (b == 0){
        // if denominator is zero, then throw the error
        throw "Divide by zero error";  // exception of type 'const char *'
    }

    return a/b;
}

int main(){
    float a, b, c;
    cout << "enter the value of a and b : ";
    cin >> a >> b;

    try{  // try division
        c = division(a, b); // if successful
        cout << "division of numbers = " << c << endl;   // then print result
    }
    catch(const char *msg){ // catch exception of 'const char *'
        // else print the result received from 'throw' statement
        cout << msg << endl;
    }

    cout << "product of numbers = " << a*b;
    return 0;
}
```

Listing 15.4: Exceptional handling

```
// exceptionEx2.cpp

#include <iostream>
using namespace std;

float division(float, float);

float division(float a, float b){
    if (b == 0){
```

(continues on next page)

(continued from previous page)

```
        // if denominator is zero, then throw the error
        throw "Divide by zero error";  // exception of type 'const char *'
    }

    if ( b == a){
        throw 2*a;  // exception of type float
    }

    return a/b;
}

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

    cout << "enter the value of a and b : ";
    cin >> a >> b;

    try{  // try division
        c = division(a, b); // if successful
        cout << "division of numbers = " << c << endl;    // then print result
    }
    catch(const char *msg){ // catch exception of 'const char *'
        // else print the result received from 'throw' statement
        cout << msg << endl;
    }
    catch(float answer){ // catch exception of type float
        cout << "Showing 2 * a = " << answer << endl;
    }

    cout << "product of numbers = " << a*b;
    return 0;
}
```