PROGRAMMING METHODOLOGY

Lab 2: Conditional & Loop Structures / Program Debugging

1 Introduction

In this lab tutorial, we will continue with conditional structures, and study loop structure. Besides that, we introduce a technique to find programming mistakes, which can cause logical error when running your program, called program debugging.

2 Conditional Structures

In **Lab 1**, we presented a basic form of conditional structure, called **if-else**. Now, we study a new approach, when we have many cases in making decision with one condition (may contain more than one Boolean expression). Instead of defining many if-else statements, we can use **switch** structure, as follows:

```
switch(expression) {
      case const-expr-1: statements
      case const-expr-2: statements
      case const-expr-3: statements
      default: statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled **default** is executed if none of the other cases are satisfied. A **default** is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

```
1     // switch.c
2     #include <stdio.h>
3
4     int main()
5     {
6           int choice = 0;
7
8           printf("Enter your choice: ");
9           scanf("%d", &choice);
10
11          switch(choice)
12          {
13                case 0: case 1:
14                      printf("Coke\n");
15                      break;
16
17                case 2:
18                      printf("Coffee\n");
19                      break;
20
21                case 3:
22                      printf("Tea\n");
23                      break;
24
25                default:
26                      printf("Water\n");
27                      break;
28          }
29    }
```

Figure 1 Sample program uses switch

The sample program in **Figure 1** illustrates the switch structure. It contains 5 cases, and case **0** and **1** have the same result, the **default** case handles the choices which are out of pre-defined cases. The most important thing that you should notice is each case (incl. default) has to have **break**; unless you define **break**, the program will continue process other cases until reaching the last one. For example, if the program in **Figure 1** doesn't have break statement, when **choice** is **0**, it will print out all cases.

3 Loop Structures

Often in computer programming, it is necessary to perform a certain action a certain number of times or until a certain condition is met. The constructs that enable computers to perform certain repetitive tasks are called loops.

3.1 while loop

A **while** loop is the most basic type of loop structure. It will run until the condition is non-zero (true). Before executing the statements inside while body, it will check whether the

condition is true, then it executes the statements. You should note that in case the condition of loop never becomes false, then we will fall into infinite loop. Let's the sample program in **Figure 2**.

```
1     // while.c
2     #include <stdio.h>
3
4     int main()
5     {
6           int i = 0;
7
8           while(i < 10)
9           {
10                printf("No. %d\n", i);
11
12                i = i + 1;  // Update the condition
13          }
14    }
```

Figure 2 Sample program uses while loop

The flow of all loops can also be controlled by **break** and **continue** statements. A **break** statement will immediately exit the enclosing loop. A **continue** statement will skip the remainder of the block and start at the controlling conditional statement again.

3.2 for loop

The structure of for loop is as in **Figure 3**, the initialization statement is executed only one, before the first evaluation of test condition. Typically, it is used to assign an initial value to some variable, although this is not strictly necessary. The test expression is evaluated each time before the code in the for loop executes. After each iteration of the loop, the increment statement is executed. This often is used to increment the loop index for the loop, the variable initialized in the initialization expression and tested in the test expression.

```
1     // for.c
2     #include <stdio.h>
3
4     int main()
5     {
6           int i = 0;
7
8           for(i = 0; i < 10; i++)
9           {
10                printf("No. %d\n", i);
11          }
12    }
```

Figure 3 Sample program uses for loop

3.3 do-while loop

A **do-while** loop is a post-check while loop, which means that it checks the condition after each run. As a result, even if the condition is zero (false), it will run at least once.

```
1     // dowhile.c
2     #include <stdio.h>
3
4     int main()
5     {
6           int i = 1;
7
8           do {
9                 printf("No. %d\n", i);
10          } while(i < 1);
11    }
```

A do-while loop sometimes uses in validating a user's input. For example, if you want to validate whether an input is correct or not, you can take advantage of do-while loop, as following program.

```
1     #include <stdio.h>
2
3     int main()
4     {
5           int input, i = 0;
6
7           do {
8                 printf("Input = ");
9                 scanf("%d", &input);
10
11                // Control number of loops
12                i++;
13          } while(input < 0);
14
15          printf("Input is a positive integer number!");
16
17    }
```

4 Exercises

1. Write a C program to print sum of all even numbers between 1 to *n* by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
2. Write a C program to print sum of all odd numbers between 1 to *n* by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
3. Write a C program to print table of any number.

4. Write a C program to enter any number and calculate sum of all natural numbers between 1 to *n* by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
5. Write a C program to find first and last digits of any number.
6. Write a C program to calculate sum of digits of any number.
7. Write a C program to calculate product of digits of any number.
8. Write a C program to count number of digits in any number.
9. Write a C program to swap first and last digits of any number.
10. Write a C program to enter any number and print its reverse.
11. Write a C program to enter any number and check whether the number is palindrome or not.
12. Write a C program to check whether a number is Prime number or not. Validating the input, in case the input isn't correct, prompt user to enter it again.
13. Write a C program to check whether a number is Armstrong number or not.
14. Write a C program to check whether a number is Perfect number or not.
15. Write a C program to print all Prime numbers between 1 to *n* by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
16. Write a C program to print all Armstrong numbers between 1 to *n* by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
17. Write a C program to print all Perfect numbers between 1 to *n* by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
18. Write a C program to convert Decimal to Binary number system.
19. Write a C program to compute the Factorial of *n*. Validating the input, in case the input isn't correct, prompt user to enter it again.

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n. \ (n \geq 0)$$

5 Reference

[1] Brian W. Kernighan & Dennis Ritchie (1988). *C Programming Language, 2nd Edition*. Prentice Hall.

[2] Paul Deitel & Harvey Deitel (2008). *C: How to Program, 7th Edition*. Prentice Hall.

[3] *C Programming Tutorial* (2014). Tutorials Point.

[4] *C Programming* (2013). Wikibooks.