Week 2 Practice Questions - Functions and Pointers

1. computeSalary
2. digitValue
3. divide
4. perfectProd
5. countOddDigits
6. allOddDigits
7. extOddDigits

Questions

1. (**computeSalary**) Write a C program that determines the gross pay for each employee in a company. The company pays "straight-time" for the first 160 hours worked by each employee for four weeks and pays "time-and-a-half" for all hours worked in excess of 160 hours. You are given a list of employee Ids (an integer), the number of hours each employee worked for the four weeks, and the hourly rate of each employee. The program should input this information for each employee, then determine and display the employee's gross pay. The sentinel value of –1 is used for the employee *id* to indicate the end of input. Your program should include three functions, apart from the main() function, to handle the input, the output and the computation of the gross pay. The function prototypes for the functions are given as follows:

```
void readInput(int *id, int *noOfHours, int *payRate);
void printOutputs(int id, double grossPay);
double computeSalary1(int noOfHours, int payRate);
void computeSalary2(int noOfHours, int payRate, double *grossPay);
```

The function **computeSalary1()** uses call by value for returning the result to the calling function. The function **computeSalary2()** uses call by reference to pass the result through the pointer parameter, grossPay, to the caller.

A sample program template is given below to test the functions:

```
#include <stdio.h>
void readInput(int *id, int *noOfHours, int *payRate);
void printOutputs(int id, double grossPay);
double computeSalary1(int noOfHours, int payRate);
void computeSalary2(int noOfHours, int payRate, double *grossPay);

int main()
{
   int id = -1, noOfHours, payRate;
   double grossPay;

   readInput(&id, &noOfHours, &payRate);
   while  (id != -1) {
      printf("computeSalary1(): ");
      grossPay = computeSalary1(noOfHours, payRate);
      printOutputs(id, grossPay);
      printf("computeSalary2(): ");
      computeSalary2(noOfHours, payRate, &grossPay);
      printOutputs(id, grossPay);
      readInput(&id, &noOfHours, &payRate);
   }
   return 0;
}
void readInput(int *id, int *noOfHours, int *payRate)
{
   /* Write your program code here */
}
void printOutputs(int id, double grossPay)
```

```
{
   /* Write your program code here */
}
double computeSalary1(int noOfHours, int payRate)
{
   /* Write your program code here */
}
void computeSalary2(int noOfHours, int payRate, double *grossPay)
{
   /* Write your program code here */
}
```

Some sample input and output sessions are given below:

(1) Test Case 1:

```
Enter ID (-1 to end):
11
Enter number of hours:
155
Enter hourly pay rate:
8
computeSalary1(): ID 11 grossPay 1240.00
computeSalary2(): ID 11 grossPay 1240.00
Enter ID (-1 to end):
12
Enter number of hours:
165
Enter hourly pay rate:
8
computeSalary1(): ID 12 grossPay 1340.00
computeSalary2(): ID 12 grossPay 1340.00
Enter ID (-1 to end):
-1
```

(2) Test Case 2:

```
Enter ID (-1 to end):
11
Enter number of hours:
155
Enter hourly pay rate:
8
computeSalary1(): ID 11 grossPay 1240.00
computeSalary2(): ID 11 grossPay 1240.00
Enter ID (-1 to end):
12
Enter number of hours:
160
Enter hourly pay rate:
8
computeSalary1(): ID 12 grossPay 1280.00
computeSalary2(): ID 12 grossPay 1280.00
Enter ID (-1 to end):
13
Enter number of hours:
200
Enter hourly pay rate:
8
computeSalary1(): ID 13 grossPay 1760.00
computeSalary2(): ID 13 grossPay 1760.00
Enter ID (-1 to end):
-1
```

(3) Test Case 3:

```
Enter ID (-1 to end):
```

```
11
Enter number of hours:
165
Enter hourly pay rate:
8
computeSalary1(): ID 11 grossPay 1340.00
computeSalary2(): ID 11 grossPay 1340.00
Enter ID (-1 to end):
-1
```

(4) Test Case 4:

```
Enter ID (-1 to end):
-1
```

2. **(digitValue)** Write a function that returns the value of the k^{th} digit (k>0) from the right of a non-negative integer *num*. For example, if num is1234567 and k is 3, the function will return 5 and if num is 1234 and k is 8, the function will return 0. Write the function in two versions. The function **digitValue1()** returns the result, while **digitValue2()** passes the result through pointer parameter `result`. The prototypes of the function are given below:

```
int digitValue1(int num, int k);
void digitValue2(int num, int k, int *result);
```

A sample program template is given below to test the functions:

```
#include <stdio.h>
int digitValue1(int num, int k);
void digitValue2(int num, int k, int *result);
int main()
{
   int num, digit, result=-1;

   printf("Enter the number: \n");
   scanf("%d", &num);
   printf("Enter k position: \n");
   scanf("%d", &digit);
   printf("digitValue1(): %d\n",  digitValue1(num, digit));
   digitValue2(num, digit, &result);
   printf("digitValue2(): %d\n", result);
   return 0;
}
int digitValue1(int num, int k)
{
   /* Write your program code here */
}
void digitValue2(int num, int k, int *result)
{
   /* Write your program code here */
}
```

Some sample input and output sessions are given below:

(1) Test Case 1:

```
Enter the number:
234567
Enter k position:
3
digitValue1(): 5
digitValue2(): 5
```

(2) Test Case 2:

```
Enter the number:
```

```
234567
Enter k position:
1
digitValue1(): 7
digitValue2(): 7
```

(3) Test Case 3:

```
Enter the number:
123
Enter k position:
8
digitValue1(): 0
digitValue2(): 0
```

3. (**divide**) Write a function that performs integer division by subtraction. The function computes the quotient and remainder of dividing *m* by *n*. Both *m* and *n* are positive integers. Write the function in two versions, and the function prototypes are given below:

```
int divide1(int m, int n, int *r);
void divide2(int m, int n, int *q, int *r);
```

In the first version **divide1()**, the function returns the quotient of dividing *m* by *n*, and the remainder is passed to the caller through the pointer parameter *r*. In the second version **divide2()**, the pointer variable *q* is used to store the quotient which will be returned to the caller, and the remainder is passed to the caller through the pointer parameter *r*. Please note that in this question, you are not allowed to use the division (/) and modulus (%) operators.

A sample program template is given below to test the functions:

```
#include <stdio.h>
int divide1(int m, int n, int *r);
void divide2(int m, int n, int *q, int *r);
int main()
{
   int m, n, q, r;

   printf("Enter two numbers (m and n): \n");
   scanf("%d %d", &m, &n);
   q = divide1(m, n, &r);
   printf("divide1(): quotient %d remainder %d\n", q, r);
   divide2(m, n, &q, &r);
   printf("divide2(): quotient %d remainder %d\n", q, r);
   return 0;
}
int divide1(int m, int n, int *r)
{
   /* Write your program code here */
}
void divide2(int m, int n, int *q, int *r)
{
   /* Write your program code here */
}
```

Some sample input and output sessions are given below:

(1) Test Case 1:

```
Enter two numbers (m and n):
10 3
divide1(): quotient 3 remainder 1
divide2(): quotient 3 remainder 1
```

(2) Test Case 2:

```
Enter two numbers (m and n):
3 5
divide1(): quotient 0 remainder 3
divide2(): quotient 0 remainder 3
```

(3) Test Case 3:

```
Enter two numbers (m and n):
32 7
divide1(): quotient 4 remainder 4
divide2(): quotient 4 remainder 4
```

(4) Test Case 4:

```
Enter two numbers (m and n):
0 7
divide1(): quotient 0 remainder 0
divide2(): quotient 0 remainder 0
```

4. (**perfectProd**) A perfect number is one that is equal to the sum of all its factors (excluding the number itself). For example, 6 is perfect because its factors are 1, 2, and 3, and 6 = 1+2+3; but 24 is not perfect because its factors are 1, 2, 3, 4, 6, 8, 12, but 1+2+3+4+6+8+12 = 36. Write a function that takes a number, num, and returns the product of all perfect numbers that are less than the number. For example, if the number is 100, the function should return 168 (which is 6*28), as 6 and 28 are the only perfect numbers less than 100. Write the function in two versions. The function **perfectProd1()** returns the result to the caller, while **perfectProd2()** passes the result through the pointer parameter `result`. The prototype of the function is given as follows:

```
int perfectProd1(int num);
void perfectProd2(int num, int *prod);
```

A sample program template is given below to test the functions:

```
#include <stdio.h>
int perfectProd1(int num);
void perfectProd2(int num, int *prod);
int main()
{
   int number, result=0;

   printf("Enter a number: \n");
   scanf("%d", &number);
   printf("perfectProd1(): %d\n", perfectProd1(number));
   perfectProd2(number, &result);
   printf("perfectProd2(): %d\n", result);
   return 0;
}
int perfectProd1(int num)
{
   /* Write your code here */
}
void perfectProd2(int num, int *prod)
{
   /* Write your code here */
}
```

Some sample input and output sessions are given below:

(1) Test Case 1:

```
Enter a number:
100
perfectProd1(): 168
perfectProd2(): 168
```

(2) Test Case 2:

```
Enter a number:
500
perfectProd1(): 83328
perfectProd2(): 83328
```

(3) Test Case 3:

```
Enter a number:
1000
perfectProd1(): 83328
perfectProd2(): 83328
```

(4) Test Case 4:

```
Enter a number:
10000
perfectProd1(): 677289984
perfectProd2(): 677289984
```

5. **(countOddDigits)** Write a C function to count the number of odd digits, i.e. digits with values 1,3,5,7,9 in a non-negative number. For example, if num is 1234567, then 4 will be returned. Write the function in two versions. The function **countOddDigits1()** returns the result to the caller, while **countOddDigits2()** passes the result through the pointer parameter `result`. The function prototypes are given below:

```
int countOddDigits1(int num);
void countOddDigits2(int num, int *result);
```

A sample program template is given below to test the functions:

```
#include <stdio.h>
int countOddDigits1(int num);
void countOddDigits2(int num, int *result);
int main()
{
   int number, result=-1;

   printf("Enter the number: \n");
   scanf("%d", &number);
   printf("countOddDigits1(): %d\n", countOddDigits1(number));
   countOddDigits2(number, &result);
   printf("countOddDigits2(): %d\n", result);
   return 0;
}
int countOddDigits1(int num)
{
   /* Write your program code here */
}
void countOddDigits2(int num, int *result)
{
   /* Write your program code here */
}
```

Some sample input and output sessions are given below:

(1) Test Case 1:

```
Enter the number:
34567
countOddDigits1(): 3
countOddDigits2(): 3
```

(2) Test Case 2:

```
Enter the number:
```

```
357
countOddDigits1(): 3
countOddDigits2(): 3
```

(3) Test Case 3:

```
Enter the number:
2468
countOddDigits1(): 0
countOddDigits2(): 0
```

6. (**allOddDigits**) Write a function that returns either 1 or 0 according to whether or not all the digits of the positive integer argument number are odd. For example, if the input argument is 1357, then the function should return 1; and if the input argument is 1234, then 0 should be returned. Write the function in two versions. The function **allOddDigits1()** returns the result to the caller, while **allOddDigits2()** passes the result through the pointer parameter `result`. The function prototypes are given below:

```
int allOddDigits1(int num);
void allOddDigits2(int num, int *result);
```

A sample program template is given below to test the functions:

```
#include  <stdio.h>
int allOddDigits1(int num);
void allOddDigits2(int num, int *result);
int main()
{
   int number, p=999, result=999;

   printf("Enter a number: \n");
   scanf("%d", &number);
   p = allOddDigits1(number);
   if (p == 1)
      printf("allOddDigits1(): yes\n");
   else
      printf("allOddDigits1(): no\n");
   allOddDigits2(number, &result);
   if (result == 1)
      printf("allOddDigits2(): yes\n");
   else
      printf("allOddDigits2(): no\n");

   return 0;
}
int allOddDigits1(int num)
{
   /* Write your program code here */
}
void allOddDigits2(int num, int *result)
{
   /* Write your program code here */
}
```

Some sample input and output sessions are given below:

(1) Test Case 1:

```
Enter a number:
1357
allOddDigits1(): yes
allOddDigits2(): yes
```

(2) Test Case 2:

```
Enter a number:
```

```
2468
allOddDigits1(): no
allOddDigits2(): no
```

(3) Test Case 3:

```
Enter a number:
13
allOddDigits1(): yes
allOddDigits2(): yes
```

(4) Test Case 4:

```
Enter a number:
9
allOddDigits1(): yes
allOddDigits2(): yes
```

7. (**extOddDigits**) Write a function that extracts the odd digits from a positive number, and combines the odd digits sequentially into a new number. The new number is returned to the calling function. If the input number does not contain any odd digits, then the function returns -1. For example, if the input number is 1234567, then 1357 will be returned; and if the input number is 28, then –1 will returned. Write the function in two versions. The function **extOddDigits1()** returns the result to the caller, while the function **extOddDigits2()** returns the result through the pointer parameter, `result`. The function prototype is given as follows:

```
int extOddDigits1(int num);
void extOddDigits2(int num, int *result);
```

A sample program template is given below to test the functions:

```
#include <stdio.h>
#define INIT_VALUE 999
int extOddDigits1(int num);
void extOddDigits2(int num, int *result);
int main()
{
   int number, result = INIT_VALUE;

   printf("Enter a number: \n");
   scanf("%d", &number);
   printf("extOddDigits1(): %d\n", extOddDigits1(number));
   extOddDigits2(number, &result);
   printf("extOddDigits2(): %d\n", result);
   return 0;
}
int extOddDigits1(int num)
{
   /* Write your program code here */
}
void extOddDigits2(int num, int *result)
{
   /* Write your program code here */
}
```

Some sample input and output sessions are given below:

(1) Test Case 1:

```
Enter a number:
1234
extOddDigits1(): 13
extOddDigits2(): 13
```

(2) Test Case 2:

```
Enter a number:
2468
extOddDigits1(): -1
extOddDigits2(): -1
```

(3) Test Case 3:

```
Enter a number:
1357
extOddDigits1(): 1357
extOddDigits2(): 1357
```

(4) Test Case 4:

```
Enter a number:
7
extOddDigits1(): 7
extOddDigits2(): 7
```