ECNG 2007: Computer Systems and Software Design (2018/2019)

Lab 2 – Numerical Conversion and Encryption

1. Numerical Conversion

Programming is not limited to usage on the computer. Many embedded systems, that use microprocessors, also require programming, albeit on the low level side. In order to program at the microprocessor level, it is essential to know how to operate between different numerical systems, since this forms the core of how data is represented and transformed within the microprocessor.

This section will deal with the four most popular numerical representations that are widely used for microprocessors which are **binary, octal, decimal** and **hexadecimal**. This section will also deal with how it is possible to move from one numerical representation to another.

1.1 Binary

The binary representation of a number only uses two units: zero (0) and one (1). For instance, if we wish to represent the decimal number "3" in binary, the result is "11".

How did we obtain this? The following steps illustrate this process:

3/2 : 1 R 1

1/2 : 0 R 1

Binary is represented in a base of 2. We use this base and divide the number, keeping track of its remainder. Once the division result is zero, we then concatenate the remainder from the result go up to obtain the binary representation of the number.

If we wished to do the reverse, given a binary number, we just use the base of the binary number and multiply each bit by its corresponding power. For example, binary "11" translated into decimal would be:

$$11_2 = (1 \times 2^1) + (1 \times 2^0) = 3_{10}$$

Microprocessors typically operate with a fixed bit width. For example, if the microprocessor in question operates on an 8-bit level, the binary representation of the decimal number “3” would now be “0000 0011”.

Note that a bit is represented as either a zero or one in the above example. There are eight numbers in the binary representation of the decimal number “3”, where each number is referred to a bit. Hence, the microprocessor in question contains 8-bits.

This means that the maximum number that can be represented by an 8-bit microprocessor is 255. This number is obtained by using the formula:

$$2^{\#\ of\ bits} - 1$$

If we are only operating with positive numbers starting from zero, this means that the maximum range of numbers for this microprocessor is from 0 to 255. A number represented in this form is referred to as an **unsigned** number. Any numbers outside this range cannot be represented since the microprocessor does not have any additional bits for such representations.

Things get a little trickier when a microprocessor is required to represent negative numbers. A negative number in binary is represented as the **two’s compliment** of its positive equivalent.

For instance, we saw that the positive decimal number “3” is represented as “0000 0011” for an 8-bit microprocessor. Its negative equivalent is obtained using the following steps for **two’s compliment**:

1) Obtain the positive equivalent of the number in binary: “0000 0011”
2) Invert all the bits in its representation: “1111 1100”
3) Add binary “1” to the result: “1111 1101”

Now, how do we obtain the decimal representation if we only had the binary representation?

Using our example above, this is done using the following steps:

1) Obtain the decimal equivalent for the highest bit and set its result to negative:
   “1000 0000” is “-128”
2) For all other bits in the representation, obtain the positive decimal equivalent:
   “111 1101” is “125”
3) Sum the two together: “-128” + “125” = “-3

Note that a number in this form is referred to as a **signed** number. In short, if a microprocessor deals with negative numbers, the first bit of its binary representation indicates whether it is positive or negative. Typically, a zero means positive and a one means negative.

What does this mean for the effective numerical range of the microprocessor though?

We saw that if only positive numbers were represented, the range was from 0 to 255. However, since negative numbers are represented and the first bit indicates the sign of the number, the range is shifted.

The maximum positive number is given by the binary representation “0111 1111”, which in decimal is “127”. The maximum negative number in binary is “1000 0001”, which in decimal is “-127”. Hence, the effective range in decimal is from -127 to 127.

You may ask why “1000 0000” is not considered, which in decimal is “-128”. However, in the binary representation, remember that the first bit just indicates the sign of the number i.e. whether it is positive or negative. The zeroes after the first bit indicate that the actual number is zero.

1.2 Octal

An octal representation of a number is given in base 8. This means that the numbers used to represent a bit (or digit) in octal is from zero to seven.

For example, if we wished to represent the decimal number “1792” in octal, we follow the same division process that was seen for binary numbers, just using the new base that we wish to transform into.

1792/8 = 224 R 0

224/8 = 28 R 0

28/8 = 3 R 4

3/8 = 0 R 3

The result for the decimal number “1792” is the octal number “3400”.

Conversely, if we wished to transform an octal number back to decimal, we use the base of the octal number and multiply each digit by its corresponding power.

$$17_8 = (1 \times 8^1) + (7 \times 8^0) = 15_{10}$$

1.3 Hexadecimal

A hexadecimal representation of a number is given in base 16. Unlike binary and octal numbers which was a downscale of a decimal number, a hexadecimal is an upscale of a decimal number. This means that 16 symbols are required to represent each digit.

Typically, the numbers 0 to 9 represent the first ten digits of the hexadecimal base. The remaining six digits are represented by the letters A – F. The table below shows how this hexadecimal representation translates into its decimal equivalent.

Decimal	**Hexadecimal**
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

The processes for upscaling are reversed when compared to downscaling. If we wish to convert the decimal number “7562” into its hexadecimal representation, we continuously divide it by the hexadecimal base.

7562/16 = 472 R 10 (A)

472/16 = 29 R 8

29/16 = 1 R 13 (D)

1/16 = 0 R 1

Note that since we have remainders greater than 9 occurring, these remainders need to be translated to their hexadecimal representation. The result for the decimal number “7562” is the hexadecimal number “1D8A”.

Conversely, if we wished to transform a hexadecimal number back to decimal, we use the base of the hexadecimal number and multiply each digit by its corresponding power.

$$1D8A_{16} = (1 \times 16^3) + (13 \times 16^2) + (8 \times 16^1) + (10 \times 16^0) = 7562_{10}$$

Remember that each digit in the hexadecimal representation must be converted to its equivalent decimal representation before doing the multiplication above.

1.4 Conversion Code for Binary <-> Decimal

We have looked at the algorithms associated with converting each of the numerical representations to and from decimal. Now, we will look at how these algorithms can be implemented in C++.

We will be using the object-oriented approach using classes. Consider the example code snippet below that shows the class named “Converter” that will contain all our implemented methods:

```
class Converter {
public:
	Converter() {};

	int decToBin(int n) {
		int result = 0, i = 1;
		while (n > 0) {
			result += n % 2 * i;
			n = n / 2;
			i *= 10;
		}
		return result;
	}
};
```

Note that an iterative approach is used to convert a decimal number to a binary number. Iterative approaches typically execute faster than recursive approaches.

The first method included in our “Converter” class is our Decimal to Binary conversion. Note how the operation occurs with respect to the algorithm discussed earlier. The base 2, which represents binary, in prevalent in this method for its entire operation.

Consider the code snippet below for our Binary to Decimal conversion method:

```
int binToDec(int num) {
	int dec_value = 0, base = 1;
	while (num) {
		int last_digit = num % 10;
		num = num / 10;
		dec_value += last_digit * base;
		base = base * 2;
	}
	return dec_value;
}
```

Note how this operation occurs in the reverse order, using the base 10 first to obtain each digit and then the base 2 to conduct the multiplication operations for summing to the end result.

The reason why base 10 is used at first is because “num”, as an input, is still seen as a decimal number. We use the modulus and division operations to extract each digit consecutively and then multiply it by its corresponding base 2 power, as seen in its corresponding algorithm.

Now, try writing your own main function to create an object of type “Converter” and then to output the result of each of these functions. Verify that both of these functions work correctly.

1.5 Conversion Code for Octal <-> Decimal

Consider the code snippet below that shows the method for decimal to octal conversion:

```
int decToOct(int num) {
	int result = 0, i = 1;
	while (num > 0) {
		result += num % 8 * i;
		num = num / 8;
		i *= 10;
	}
	return result;
}
```

Note the similarity that this function has with respect to the decimal to binary method. All that has changed is the base that we are operating in, since the algorithmic steps are essentially the same.

The conversion for octal to decimal also is quite similar to that of the binary to decimal method. Make an attempt and write your own method, noting the algorithmic steps taken for the conversion process.

1.6 Conversion Code for Hexadecimal <-> Decimal

Things get a little trickier when dealing with hexadecimal numbers. Since hexadecimal numbers contain the numeric characters 0 to 9, as well as the alphabetic characters A to F, an integer input will not work. A string input is required for processing.

Consider the code snippet below that shows the method for hexadecimal to decimal conversion:

```
int hexToDec(std::string num) {
  int base = 16, dec_value = 0;
  for (int i = 0; i < num.length(); i++) {
    if (num[i] >= '0' && num[i] <= '9') {
       dec_value += (int(num[i]) - '0') * pow(base, num.length() - i - 1);
              }
    else if (num[i] >= 'A' && num[i] <= 'F') {
       dec_value += (int(num[i] - 'A' + 10) * pow(base, num.length() - i - 1));
    }

    else if (num[i] >= 'a' && num[i] <= 'f') {
       dec_value += (int(num[i] - 'a' + 10) * pow(base, num.length() - i - 1));
    }
  }

  return dec_value;
}
```

Note that we use the ASCII table here to do our conversions for the alphabetic characters A to F. A link to the ASCII table can be found here -> https://www.asciitable.com/

Note that the code also considers lowercase alphabetic characters for its conversion. The offset "– 'A' + 10", as seen in the last two branches of the "if" statement, indicate how the conversion takes place for obtaining the equivalent decimal value. Try to figure out why this offset is required, using the ASCII table.

Typecasting is also done to ensure that the correct data type is obtained, during each step of the conversion process. If typecasting is not done, the compiler will throw an error.

Consider the code snippet below that shows the method for decimal to hexadecimal conversion:

```
        std::string decToHex(int num) {
                int temp = 0; std::string hexNum;

                while (num > 0) {
                        temp = num % 16;

                        if (temp < 10) {
                                temp += 48;
                        }
                        else {
                                temp += 55;
                        }

                        hexNum = char(temp) + hexNum;
                        num = num / 16;
                }

                return hexNum;
        }
};
```

Note that we also use the ASCII table here for our conversion process. Typecasting is also done here for "hexNum" in order to obtain the equivalent character representation of the ASCII number.

1.7 Other Conversions

The preceding sections indicate how to do conversions to and from the decimal base to other bases. However, what if we decided to not use the decimal base and do a straight conversion instead between two bases e.g. binary to octal?

You are required to research how these conversions are done.

2. In-Lab: Encryption

Before the in-lab exercise begins, please note that **all the work done in-lab would be required for use later for the last section.** As of such, students should ensure that they have some form of media or online storage to save their code.

It is required that the following encryption techniques are coded using Object Oriented Programming. It is possible to test each encryption technique as regular functions initially but the ultimatum is that they must be coded as objects using the previous instructions given within this lab script as a guideline. It is important that this is done since this sets the basis for all future assignments, projects and exams within the course.

The basis for encryption is the use of ciphers. Ciphers transform data from one form to another as an encrypted way to mask data from prying eyes. Typically, most basic ciphers use some form of encryption key. However, ciphers can be created using hardware and the first form of encryption that will be investigated will be using a logic gate.

2.1. XOR Shift

The basis for this encryption is on the XOR gate relationship amongst the bits used for ASCII. If we consider the letter 'a' with an ASCII equivalent of binary 0110 0001, it is possible to use an encryption key to encode the data.

Given the binary encryption key as 0110 0010, if 'a' is masked using the XOR gate, the following can be obtained:

Inputs		Output
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

A: 0110 0001

B: 0110 0010

O: 0000 0011

If the encrypted key is applied to the output, it is possible to decrypt the information and retrieve the original binary value of ‘a’.

O: 0000 0001

B: 0110 0010

A: 0110 0001

As an exercise, convert the word “hello” using the key 0111 1000 and decrypt it to ensure that it was done correctly. Recall that the ‘^’ operator is used for the XOR gate in C++.

2.2. Caesarian

The Caesarian shift is one of the simplest and widely known encryption techniques. It is a substitution cipher that uses a fixed shift number *N* to determine the mapping of letters within the alphabet.

If the cipher is contained to only the letters in the alphabet, it is called a ***bounded*** Caesarian shift. If the cipher is not contained and allows for all ASCII characters to be encoded, it is called an ***unbounded*** Caesarian shift.

The following code snippet shows an example of how the ***bounded*** Caesarian shift can be implemented:

```
const string alphabet = "abcdefghijklmnopqrstuvwxyz";

string caesar(string str, int amt) {
   string encryptedText= "";

   for (int i=0; i<str.length(); i++)
      encryptedText += alphabet [(alphabet.find(str[i]) + amt) % 26];

   return encryptedText;
```

```
}


string decaesar(string str, int amt) {
  string decryptedText= "";

  for (int i=0; i<str.length(); i++){
     int index = alphabet.find(str[i])-amt;

     while( index < 0) index+=26;

     decryptedText += alphabet [index];
  }

  return decryptedText;
}
```

Using the code snippet:

- Write the main program to implement this program showing that the encryption and decryption functions work.
- Instead of using a constant string to represent the alphabet, modify the code to use the ASCII representation table instead.

2.3. ROT13

The ROT13 cipher is also a substitution cipher where *N* is 13. Its implementation is similar to that of the ***bounded*** Caesarian shift cipher. This means that the basis of the ROT13 cipher only uses letters of the alphabet and does not encrypt any special characters.

The code snippet below shows an example of how the ROT13 cipher can be implemented:

```
const string alphabet = "abcdefghijklmnopqrstuvwxyz";
```

```
string rot13(string str) {

   string encryptedText= "";

   for (int i=0; i<str.length(); i++)
      encryptedText += alphabet [(alphabet.find(str[i]) + 13) % 26];

   return encryptedText;
}
```

Note that since the encryption is symmetrical due to *N* being 13 and the cipher is bounded to the letters of the alphabet, the encryption and decryption function uses the same implementation.

Using the code snippet:

- Write the main program to implement this program showing that the encryption and decryption functions work.
- Instead of using a constant string to represent the alphabet, modify the code to use the ASCII representation table instead.

2.4. Atbash

The Atbash cipher is a substitution cipher but it does not operate on a shift basis such as the Caesarian and ROT13 ciphers. Instead, it operates on the mirror basis where 'a' maps to 'z', 'b' maps to 'y' and so on. This means that the Atbash cipher is ***bounded*** to the alphabet.

The implementation for the Atbash cipher is slightly more difficult that a general shift cipher but it is possible to determine how this cipher works using mathematics (this is one way of how algorithms are formed). For example, since 'a' maps to 'z', let the position of 'a' be 1 and the position of 'z' be 26. If we were to create a formula for this mapping, it would be such that a general shift of 25 characters has taken place.

However, when we move to the mapping of 'b' to 'y', a general shift of 23 characters occur since 'b' is position 2 and 'y' is position 25. Notice any patterns as yet to the mirror basis?

If the start and end positions are known, a formula can be created for the mapping which is:

```
output = start_pos + end_pos – input
```

If we attempt to use this formula on ‘a’, the result would be:

```
‘a’ = 1 + 26 – 1 = 26 = ‘z’
```

Similarly, if we attempt to use the formula of ‘b’, the result would be:

```
‘b’ = 1 + 26 – 2 = 25 = ‘y’
```

Using the algorithm for the Atbash cipher:

- Implement the algorithm as a function and test its encryption and decryption in the main function. *(Hint: Do not use the constant string alphabet, the function can be implemented using a loop and string manipulation)*

2.5. Vigenere

The Vigenere cipher encrypts alphabetic text by using multiple Caesar ciphers based on the letters of a keyword as the shift. This shift is known as the polyalphabetic substitution and the Vigenere cipher is ***bounded*** to the alphabet.

The keyword is not a fixed length of letters and as of such, if the keyword is shorter than the input word to be encrypted, the keyword is repeated until the end of the input word. An example of this can be seen below if the keyword is set to ‘key’:

Input	L	A	B	E	X	E	R	C	I
Keyword	K	E	Y	K	E	Y	K	E	Y

The code snippet below shows an example of how the Vigenere cipher can be implemented:

```
string vigenere(string str, string cipher){
	string encryptedText = "";

	for(int i=0, j=0; i<str.length(); i++){
		encryptedText+= alphabet[(alphabet.find(str[i])
				+ alphabet.find(cipher[j]))%26];
		j++;
```

```
            if(j==cipher.length()) j=0;
        }
        return encryptedText;

}
```

Using the code snippet:

- Write the main program to show that the encryption function works.
- Using the encryption function, determine how to formulate the decryption algorithm and thus implement the decryption function.
- Instead of using a constant string to represent the alphabet, modify the code to use the ASCII representation table instead.

3. Objected Oriented Programming

In preparation for the lab exam, ensure that the work done during the in-lab exercise is saved before proceeding with this section.

Using the code for each of the ciphers, implement the encryption and decryption functions as methods within a class. It is also required that:

- Constructors are used for each different encryption function
- Access modifiers are also used during the implementation
- Methods may be implemented either within the class declaration or outside of the class declaration
- Do not use the constant string as a lookup table. Use the ASCII table instead.