Programming report
Week 2 Assignments C++

Jaime Betancor Valado
Christiaan Steenkist
Remco Bos
Diego Ribas Gomes

September 28, 2016

Assignment 10, Text Changer

We were tasked with making a program that changes the text of an input file in different ways as lower case everything, upper case everything, capitalize first letter of each line, etc. For that we were asked to implement a menu with the different functionalities and some functions to handle the options.

Code Listings

Listing 1: `main.ih`

```
1  #include <iostream>
2  #include <unistd.h>
3
4  enum mode
5  {
6      ERROR, OK,
7      CAPITALIZE, LOWER_CASE,
8      VERSION, USAGE
9  };
10
11 struct Vars
12 {
13     int mode = OK;
14     std::string input;
15 };
16
17 void arguments(Vars &vars, int numArgs, char *param[]);
18 int process(Vars const &vars);
19
20 void capitalizeFile(Vars const &vars);
21 void lowerCaseFile(Vars const &vars);
22 void readFile(Vars &vars);
23 void usage();
```

Listing 2: `main.cc`

```
#include "ex10.ih"

int main(int argc, char *argv[])
{
  Vars vars;
  arguments(vars, argc, argv);
  if (process(vars))
    return 1;
}
```

Listing 3: `arguments.cc`

```
#include "ex10.ih"
#include <getopt.h>

using namespace std;

void arguments(Vars &vars, int numArgs, char *param[])
{
  const struct option longOpts[] =
  {
    { "capitalize", 0, 0, 'c' },
    { "help", 0, 0, 'h' },
    { "lc", 0, 0, 'l' },
    { "lower-case", 0, 0, 'l' },
    { "uc", 0, 0, 'c' },
    { "version", 0, 0, 'v' },
    { 0, 0, 0, 0 }
  };

  int flag;
  while ( (flag = getopt_long(numArgs, param, "chlv", longOpts,
     0)) != -1)
  {
    switch (flag)
    {
      case('h'):
        vars.mode = USAGE;
      break;

      case('v'):
        if (vars.mode != USAGE)
          vars.mode = VERSION;
      break;

      case('l'):
        if (vars.mode == CAPITALIZE)
        {
          cout << "ERROR: commands not allowed as combination"
    " (\'-c\' \'-l\')\n";
          vars.mode = ERROR;
        }
        else if (vars.mode != USAGE && vars.mode != VERSION)
```

```
40            vars.mode = LOWER_CASE;
41          break;
42
43          case('c'):
44            if (vars.mode == LOWER_CASE)
45            {
46              cout << "ERROR: commands not allowed as combination"
      " (\'-c\' \'-l\')\n";
47              vars.mode = ERROR;
48            }
49            else if (vars.mode != USAGE && vars.mode != VERSION)
50              vars.mode = CAPITALIZE;
51          break;
52        }
53    }
54
55    readFile(vars);
56  }
```

Listing 4: `capitalize.cc`

```
1  #include "ex10.ih"
2  #include <fstream>
3  #include <string>
4
5  using namespace std;
6
7  void capitalizeFile(Vars const &vars)
8  {
9    for (char character : vars.input)
10   {
11     character = toupper(character);
12     cout << character;
13   }
14 }
```

Listing 5: `lowercasefile.cc`

```
1  #include "ex10.ih"
2  #include <fstream>
3  #include <string>
4
5  using namespace std;
6
7  void lowerCaseFile(Vars const &vars)
8  {
9    for (char character : vars.input)
10   {
11     character = tolower(character);
12     cout << character;
13   }
14 }
```

Listing 6: `process.cc`

```
#include "ex10.ih"
#include <getopt.h>

using namespace std;

int process(Vars const &vars)
{
  switch (vars.mode)
  {
    case(ERROR):
      return 1;

    case(USAGE):
      usage();
    break;

    case(VERSION):
      cout << "VERSION: 1.0\n";
    break;

    case(LOWER_CASE):
      lowerCaseFile(vars);
    break;

    case(CAPITALIZE):
      capitalizeFile(vars);
    break;
  }

  return 0;
}
```

Listing 7: `readfile.cc`

```
#include <iostream>
#include "ex10.ih"

using namespace std;

void readFile(Vars &vars)
{
  if (!isatty(fileno(stdin)))
  {
    string line;
    string input;
    while ( getline(cin, line) )
    {
      input = input + line + '\n';
    }
    vars.input = input;
  }
  else
  {
```

```
20      if (vars.mode != USAGE && vars.mode != VERSION)
21      {
22        cout << "no file redirection\n";
23        vars.mode = ERROR;
24      }
25    }
26  }
```

Listing 8: `usage.cc`

```
1   #include <iostream>
2
3   using namespace std;
4
5   void usage()
6   {
7     cout << "<Text size changer> <Version 1.0>\n\n";
8     cout << "Usage: <Text type changer> [--capitalize (--uc, -u)"
        ",\n";
9     cout << "\t--help (-h), --lower-case (--lc, -l), --version"
        "(-v)] < file\n";
10    cout << "Where:\n";
11    cout << "\t--capitalize (--uc, -u); capitalize the letters"
        " in 'file'\n";
12    cout << "\t--help (-h); shows the usage of the program\n";
13    cout << "\t--lower-case (--lc, -l); decapitalize the letters"
        " in 'file'\n";
14    cout << "\t--version (-v); shows the program's version\n\n";
15    cout << "<Text size changer processes 'file' and writes the"
        " results to the standard\n ";
16    cout << "output stream>\n";
17  }
```

Assignment 11, Questions - It is appropiate to?

Here are the answers and the examples for the questions of assigment 11.

define an int-type parameter

It is appropriate to use an int-type paramater when an int-type value passed as argument must keep its original value after use. In the code example this is illustrated because it shows a function that adds 2 to the variable, and keeps it original value to use for squaring the int-value.

Listing 9: `example`

```
1   void add2(int y)
2   {
3       std::cout << "y = " << y + 2<< '\n';
4   }
5
6   void square(int z)
```

```
{
    std::cout << "z = " << z * z << '\n';

int main()
{
    int x = 5;
    std::cout << "x = " << x << '\n'; // x = 5

    add2(x);                    // x = 7
    square(x)                           // x = 25

    std::cout << "x = " << x << '\n'; // x = 5

}
```

define a std::string value parameter

It is appropriate to use a std::string value parameter when a std::string variable needs to be passed as argument and have multiple copies, but keeping its original value where it is defined. In the code example this is illustrated, because a std::string is first modified to show the std::string without blanks, later the original std::string value is used to capitalize all characters.

Listing 10: `example`

```
int main()
{
    std::string newString;
    removeBlanks(newString); // removes blanks and
        // prints the modified string
    capitalizeString(newString); // capitalizes the original
                    // string
              // so it still has blanks if they were present
}
```

define a const reference to an int-type parameter

It is appropriate to use a const reference to an int-type parameter if the int-type referenced to must not be modified, but that int-type value must be passed as argument. It is somewhat similar to pass by value, however pass by const reference improves performance if big objects are passed, because no copy is made. Hence there is no characteristic situation, because passing by value has the same result.

define a const referente to a std::string value parameter

It is appropriate to use a const reference to a std::string value parameter if the std::string referenced too must not be modified but used somewhere else. assing big objects by const reference improves performance, because no copy is made in memory. The std::string is read-only in this case. In the code example this is illustrated, because a big file is passed to the function after which a specific word is to be found in the file and that shows its occurence.

Listing 11: example

```
int main()
{
  std::string fileName = readFile();
  occurrence(fileName);

}

void occurence(std::string const& fileName)
{
  // code to find occurrence of a chosen word by user
}
```

define a non-const reference to an int-type parameter

It is appropriate to use a non-const reference int-type parameter if the int-type value must not be copied and must be modified. In the code example this is illustrated, because the int-type value is used and modified by a function that does one operation on the int-type.

Listing 12: example

```
int main()
{
  int x = 5;

  square();

  std::cout << x;
}

int square(int &x)
{
  return x * x;
}
```

define a non-const reference to a std::string value parameter

It is appropriate to use a non-const reference std::string value paramater if the std::string is modified but no copy is needed. In the code example this is illustrated, because the blanks of the std::string are removed by a function.

Listing 13: example

```
int main()
{
  std::string withBlanks = "There are blanks";
  removeBlanks(withBlanks);
  std::cout << withBlanks;
}

void removeBlanks(std::string &withBlanks)
{
```

```
10    // code to remove all blanks from string
11  }
```

define a const rvalue-reference to an int-type parameter

It is appropriate to use a const rvalue-reference int-type parameter if we need an unmodifiable reference to an int-type, however this is easier done with a const lvalue reference.

define a const rvalue-reference to a std::string parameter

It is appropriate to use a const rvalue-reference std::string value if we need an unmodifiable reference to a std::strin value, however this is easier done with a const lvalue reference.

define a rvalue-reference to an int-type parameter

It is appropriate to use a non-const rvalue-reference int-type parameter if we need a reference to a returned anonymous variable. This can also be done with an lvalue-reference,however this needs copying. In the code example this is illustrated, because it uses the anonymous variable as argument to display if the anonymous variable is an rvalue.

Listing 14: `example`

```
1   int intVal()
2   {
3     return 5;
4   }
5
6   void receive(int &value)
7   {
8     std::cout << "int value parameter\n";
9   }
10
11  void receive(int &&value)
12  {
13    std::cout << "int R-value parameter\n";
14  }
15
16  int main()
17  {
18    receive(18); // 18 is anonymous variable, hence rvalue
19    int value = 5;
20    receive(value); // value is lvalue
21    receive(intVal()); // the number returned by intVal() is
22      //anonymous, hence rvalue
23  }
```

define a rvalue-reference to a std::string parameter

It is appropriate to use a non-const rvalue-reference std::string value to use a temporary string as argument. In the code example this is illustrated, because it shows the temporary string as output.

Listing 15: example

```
std::string returnString()
{
  return "temporary string";
}

void receive(std::string &value)
{
  std::cout << "string value parameter\n";
}

void receive(std::string &&value)
{
  std::cout << "string R-value parameter\n";
}


int main()
{
  std::string newString = "newString";
  receive(newString);            // newString is lvalue
  receive(returnString());       // returnStrings return
    //std::string is temporary
}
```

define an int-type value

It is appropriate to return an int-type value when a variable declared inside the function must be returned. In the code example this is illustrated, because an operation on a local variable with the function argument is performed.

Listing 16: example

```
int squareLocal(int x)
{
  int value = x * 2;
  return value;
}
```

define a std::string value

It is appropriate to return a std::string value when a std::string declared inside the function must be returned.In the code example this is illustrated, because a local std::string is merged with the function argument and returned.

Listing 17: example

```
std::string mergeStrings(std::string argString)
{
  std::string localString = "Hello";
  std::string newString = localString + argString;
  return newString;
}
```

define a const reference to an int

It is appropriate to return a const reference to an int-type parameter when the returned int-type variable must be unmodifiable and not be copied. An example can be to pass a constant numeric value to different functions, however this may not be useful. Because a constant numeric value is often not used in many situations. If you want a constant number to be defined, like the age of a person which must not be modified and shown later, then a std::string is more useful.

define a const reference to a std::string

It is appropriate to return a const reference to a std::string value if a std::string value must remain unmodified and being the only instance that there is. It also improves performance. In the code example this is illustrated as a function that obtains a name of a person and then prints it (in a real program this may be used to define the name in a database).

Listing 18: `example`

```
name = "Willem";

const std::string& getName()
{
  return name;
}

int main()
{
  std::cout << getName();
}
```

define a non-const reference to an int

It is appropriate to return a reference to an int-type parameter when the int-type variable itself must be returned instead of its value and making a copy of it, however only global variables can be used. In the code example this is illustrated because the global variable referenced to is used in the main function on the left side of an assignment statement. The global variable receives the value assigned to it.

Listing 19: `example`

```
int n;
int& test()

{
```

```
5       return n;
6   }
7
8   int main()
9   {
10      test() = 5; // test() is replaced with the
11      //returned n, which then receives the value 5
12      cout << n;
13      return 0;
14  }
```

define a non-const reference to a std::string

It is appropriate to return a reference to a std::string value when the std::string variable itself must be returned instead of its value and making a copy of it, however only global variables can be used. In the code example this is illustrated because the global variable referenced to is used in the main function on the left side of an assignment statement. The global variable receives the value assigned to it.

Listing 20: `example`

```
1   std::string newString;
2   std::string& test()
3
4   {
5       return newString;
6   }
7
8   int main()
9   {
10      test() = "Test this string\n"; // test() is replaced
11                  //with the returned
12                                      // std::string variable,
13                  //which then receives
14                                      // the value "Test
15                  // this string\n"
16      cout << newString;
17      return 0;
18  }
```

define a const rvalue-reference to an int

It is never appropriate to return a const rvalue reference to an int-type and it is also never appropriate to return a non-const rvalue reference to an int-type because the int-type rvalue has a lifetime lower than the function and does not exist longer. It is better to return by value because you want to use the value of the int-type variable. And if the int-type does exist longer than the function lifetime, you only win some efficiency in not having a copy. The int-type object is small and the compiler can take care of of copies.

define a const rvalue-reference to a std::string

It is sometimes appropriate to return a non-const rvalue reference or a const rvalue reference to a std::string in the case of move semantics. However a move constructor is already present for the return type of a std::string, so returning the std::string by value is ok and the compiler takes care of that.

Assignment 12, Big Numbers

We were tasked with making a function that prints unsigned long longs with separators. We are proud to announce the recursive `printBig` function. The non-recursive variant of this amazing function will be revealed at a later date (at the resubmit mayhaps).

Code Listings

Listing 21: `bignum.h`

```
1  #ifndef BIGNUM_H
2  #define BIGNUM_H
3
4  #include <iostream>
5
6  enum sizes{
7      BASE_THOUSAND = 1000,
8      BASE_TEN = 10,
9      SEPARATION = 3
10 };
11
12 void printBig(std::ostream &outStream, long long value);
13 void printBig(std::ostream &outStream, long long value, size_t
       steps);
14 void splitDigits(std::ostream &outStream, long long value,
       size_t steps);
15
16 #endif
```

Listing 22: `printbig.cc`

```
1  #include "bignum.h"
2
3  using namespace std;
4
5  void printBig(ostream &outStream, long long value)
6  {
7      printBig(outStream, value, 0);
8  }
```

Listing 23: `printsmall.cc`

```
1  #include "bignum.h"
2
3  using namespace std;
```

```
4
5  // This recursive function prints n digits then a seperator (')
       .
6  // Here n is SEPARATION.
7  void splitDigits(ostream &outStream, long long value, size_t
       steps)
8  {
9      if (steps == SEPARATION)
10     {
11         outStream << '\'';
12         return;
13     }
14     if (value == 0)
15         return;
16
17     splitDigits(outStream, value / BASE_TEN, steps + 1);
18
19     outStream << (char)('0' + value % BASE_TEN);
20 }
```

Assignment 13, Fibonacci

We were tasked with making an efficent as well as a horribly inefficient way of calculating Fibonacci numbers. The functions `fib` and `rawfib` are the two methods we implemented to give you these fascinating numbers.

Time Tests

To show how absolutely primitive `rawfib` is we did a timing test where both functions were tasked with calculating the Fibonacci numbers at places 0 through 41. Here are the results of these tests with at the top the performance of our beloved `fib` function and at the bottom the noticably slower `rawfib`.

- `time ./main 41` gave a real time of 0m0.043s
- `time ./main 41 x` gave a real time of 1m37.808s

Code Listings

Listing 24: `main.h`

```
1  #ifndef MAIN_H
2
3  unsigned long long fib(unsigned long long value, unsigned long
       long fibval[]);
4  unsigned long long rawfib(unsigned long long value);
5
6  #endif
```

Listing 25: `main.cc`

```
#include <iostream>
#include "main.h"

using namespace std;

int main(int argc, char **argv)
{
    unsigned long long idx = stoull(argv[1]);
    unsigned long long fibNumber = 0;
    unsigned long long fibval[idx + 1] = {0};

    for (size_t step = 0; step < idx; ++step)
    {
         if (argc > 2)
         fibNumber = rawfib(idx);
     else
         fibNumber = fib(idx,fibval);
    }

    cout << "The " << idx << "th fibonacci number is: " <<
    fibNumber <<"\n";

}
```

Listing 26: `fib.cc`

```
#include "main.h"

unsigned long long fib(unsigned long long value, unsigned long
    long fibval[])
{
    if (value < 2)
        return fibval[value] = 1;

    if (fibval[value] != 0)
        return fibval[value];

    return fibval[value] = fib(value - 1, fibval) +
           fibval[value - 2];
}
```

Listing 27: `rawfib.cc`

```
#include "main.h"

unsigned long long rawfib(unsigned long long value)
{
    if (value <= 2)
        return 1;
    else
        return rawfib(value - 1) + rawfib(value - 2);
}
```

Assignment 14, Evaluator

We were tasked with designing a program that evaluates the design quality of a program. The design quality can roughly be translated to the number of comments divided by the number of lines and is said to be an indicator for the quality of source files.

Score Calculations

The actual design quality is calculated by taking the ratio of two scores. These scores are the line scores and comment scores which are based on the number of lines and comments. Every line counts only once but after 30 lines they count double and at an indentation above 3 they count doubly as well. Every end of line comment counts but c-style comments only fully count when they are at the start of a file, otherwise they only contribute one point per block.

Code Listings

Listing 28: `main.h`

```
#ifndef MAIN_H
#define MAIN_H

#include <iostream>

// Design Quality is multiplied by 2
// therefore it starts at 2 and is
// later multiplied.
struct Vars{
    std::string input;

    size_t indent = 0;
    size_t maxIndent = 0;

    size_t lines = 0;
    size_t lineScore = 0;

    size_t eolComments = 0;
    size_t cStyleComments = 0;
    size_t commentScore = 0;
    size_t commentLength = 0;

    size_t designQuality = 2;
};

size_t const indentLimit = 3;
size_t const lineLimit = 30;
size_t const percent = 100;

size_t getInput(Vars &vars);

void computeScores(Vars &vars);
void computeQuality(Vars &vars);
void printScores(Vars &vars);
```

```
35 void endLine(Vars &vars);
36
37 size_t startComment(Vars &vars, size_t stringPos);
38 size_t endComment(Vars &vars, size_t stringPos);
39
40 #endif
```

Listing 29: `main.cc`

```
1  #include "main.h"
2
3  int main (int argc, char **argv)
4  {
5      Vars vars;
6      while (getInput(vars))
7      {
8          computeScores(vars);
9      }
10     computeQuality(vars);
11     printScores(vars);
12 }
```

Listing 30: `computequality.cc`

```
1  #include "main.h"
2
3  void computeQuality(Vars &vars)
4  {
5      vars.designQuality *= percent * vars.commentScore;
6      vars.designQuality /= vars.lineScore;
7      vars.designQuality = vars.designQuality > percent ?
8          percent : vars.designQuality;
9  }
```

Listing 31: `computescores.cc`

```
1  #include "main.h"
2
3  using namespace std;
4
5  // Only characters indicative of indentations, lines and
       comments are
6  // taken into account. Other characters are simply ignored.
7  void computeScores(Vars &vars)
8  {
9      for (size_t stringPos = 0; stringPos != vars.input.size();
       ++stringPos)
10     {
11         switch (vars.input[stringPos])
12         {
13             case('\n'):
14                 endLine(vars);
15             break;
16
```

```
            case('/'):
                stringPos = startComment(vars, stringPos);
            break;

            case('*'):
                stringPos = endComment(vars, stringPos);
            break;

            case('}'):
                --vars.indent;
            break;
            case('{'):
                vars.maxIndent = ++vars.indent > vars.maxIndent
     ? vars.indent : vars.maxIndent;
            break;
        }
    }
}
```

Listing 32: `endcomment.cc`

```
#include "main.h"

using namespace std;

// The combination "*/" indicates the end of a c style comment.
// Loose '*' are ignored.
size_t endComment(Vars &vars, size_t stringPos)
{
    if (stringPos < vars.input.size() && vars.input[++stringPos
    ] == '/')
    {
        ++vars.cStyleComments;
        vars.commentScore += vars.commentLength == vars.lines ?
            vars.commentLength - 1 : 1;
    }
    else
        --stringPos;
    return stringPos;
}
```

Listing 33: `endline.cc`

```
#include "main.h"

void endLine(Vars &vars)
{
    size_t mul1 = (++vars.lines > lineLimit ? 2 : 1);
    size_t mul2 = (vars.indent > indentLimit ? 2 : 1);
    vars.lineScore += mul1 * mul2;
    ++vars.commentLength;
}
```

Listing 34: `getinput.cc`

```
#include "main.h"

using namespace std;

size_t getInput(Vars &vars)
{
    if (getline(cin, vars.input))
    {
        vars.input += '\n';
        return 1;
    }
    return 0;
}
```

Listing 35: `printscores.cc`

```
#include "main.h"

using namespace std;

void printScores(Vars &vars)
{
    cout << "This file has " << vars.lines << " lines.\n" <<
        "This file has " << vars.eolComments << " end of line
    comments.\n" <<
        "This file has " << vars.cStyleComments << " c-style
    comments.\n" <<
        "This file is at most " << vars.maxIndent << "
    indentations deep.\n" <<
        "The design quality of this file is " << vars.
    designQuality << ".\n";
}
```

Listing 36: `startcomment.cc`

```
#include "main.h"

using namespace std;

// The combination "//" indicates an end of line comment.
// The combination "/*" indicates a c style comment.
// Loose '/' are ignored.
size_t startComment(Vars &vars, size_t stringPos)
{
    if (++stringPos < vars.input.size() && vars.input[stringPos
    ] == '/')
    {
        ++vars.eolComments;
        ++vars.commentScore;
    }
    else if (stringPos < vars.input.size() && vars.input[
    stringPos] == '*')
        vars.commentLength = 0;
```

```
17      else
18          --stringPos;
19
20      return stringPos;
21  }
```

Assignment 15, Bit Shifts

We were tasked with making a program that can use three different methods to find the most significant big. These methods use bit shifts, taking a logarithm and truncating and binary search. It should also return the least significant bit.

Time Tests

We did tests on big and large numbers for every method. The methods in order are `shiftSearch`, `truncateSearch` and `binarySearch`.

- Shift search: `time ./main 10 1 30000000` gave a real of 0m0.238s
- Truncate search: `time ./main 10 2 30000000` gave a real of 0m2.919s
- Binary search: `time ./main 10 3 30000000` gave a real of 0m0.859s
- Shift search: `time ./main 18446744073709551615 1 30000000` gave a real of 0m4.978s
- Truncate search: `time ./main 18446744073709551615 2 30000000` gave a real of 0m2.904s
- Binary search: `time ./main 18446744073709551615 3 30000000` gave a real of 0m0.874s

Results

It seems that the first method is better for the low values and the binary is almost constant in spite of the value and is the best with large numbers. The second one is also almost constant in time in spite of the value but in low values is the slow process and in high values the binary outperforms it in matters of time.

Code Listings

Listing 37: `main.h`

```
1   #ifndef MAIN_H
2   #define MAIN_H
3
4   #include <iostream>
5   #include <cmath>
6
7   struct Vars
8   {
```

```
    unsigned long long value;
    int met;
    size_t iter;
};

size_t const byteSize = 8;

void arguments(Vars &vars, int argc, char **argv);

size_t shiftSearch(unsigned long long number);
size_t truncateSearch(unsigned long long number);
size_t binarySearch(unsigned long long number);
size_t lsbSearch(unsigned long long number);
#endif
```

Listing 38: `main.cc`

```
#include <iostream>
#include "main.h"

using namespace std;

int main(int argc, char **argv)
{
    Vars vars;
    arguments(vars, argc, argv);

    size_t msbOff = 0;
    for (size_t idx = 0; idx != vars.iter; ++idx)
    {
        // 3 functions that search most significant bit
        switch (vars.met)
        {
        case 1:
            msbOff = shiftSearch(vars.value);
        break;
        case 2:
            msbOff = truncateSearch(vars.value);
        break;
        case 3:
            msbOff = binarySearch(vars.value);
        break;
        }
    }

    size_t lsbOff = lsbSearch(vars.value);

    cout << "The MSB offset is: " << msbOff << "\n";
    cout << "The LSB offset is: " << lsbOff << "\n";
}
```

Listing 39: `binary.cc`

```
#include "main.h"
```

```
2
3  size_t binarySearch(unsigned long long number)
4  {
5      int msb = 0;
6      size_t low = 0;
7      size_t high = byteSize * sizeof (number);
8      size_t mid = 0;
9      unsigned long long shift = 1;
10
11     while (true)
12     {
13         mid = (low + high) / 2;
14         shift = (unsigned long long)1 << mid;
15
16         if (mid == low)
17         {
18             msb = mid;
19             break;
20         }
21         if (shift > number)
22             high = mid;
23         else
24             low = mid;
25     }
26     return msb - 1;
27 }
```

Listing 40: `lsb.cc`

```
1  #include "main.h"
2
3  size_t lsbSearch(unsigned long long number)
4  {
5      size_t rest = 0;
6      size_t lsb = 1;
7
8      while (true)
9      {
10         rest = number % 2;
11         number /= 2;
12
13         if (rest == 0)
14             ++lsb;
15         else
16             break;
17     }
18
19     return lsb - 1;
20 }
```

Listing 41: `shift.cc`

```
1  #include "main.h"
2
```

```
3  size_t shiftSearch(unsigned long long number)
4  {
5      size_t msb = 0;
6
7      while (true)
8      {
9          number = number >> 1;
10         ++msb;
11         if (number == 1)
12             break;
13     }
14
15     return msb - 1;
16 }
```

Listing 42: `truncate.cc`

```
1  #include "main.h"
2
3  double const ln2 = 0.693147181;
4
5  size_t truncateSearch(unsigned long long number)
6  {
7      size_t msb = 0;
8      msb = log(number) / ln2;
9      return msb - 1;
10 }
```

Assignment 16, Cipher

We were tasled with making a Vigenre cipher program. The idea of the program is to implement a Vigenre cipher that can translate newlines, tabs and all printable characters.

Design

To encrypt and decrypt the program goes through 3 simple steps for each character. First the program translates the character to a compressed ascii table using the `compressChar` function. This makes the character inhabit a range of numbers from zero to the total number of character minus one. The second step is shifting the compressed character by the right amount. For this the apropriate character in the cipher is used after it has also been compressed. After the shift the now translated character is shifted back into the normal ascii table again and printed.

Code Listings

Listing 43: `main.ih`

```
1  #ifndef CIPHER_H
2  #define CIPHER_H
3
```

```
4  #include <iostream>
5
6  using namespace std;
7
8  enum Action
9  {
10     USAGE,
11     ENCRYPT,
12     DECRYPT
13 };
14
15 struct Vars
16 {
17     string input;
18     string cipher;
19     int action;
20 };
21
22 void arguments(Vars &vars, int argc, char **argv);
23 void process(Vars const &vars);
24
25 void readFile(Vars &vars);
26 void readCipher(Vars &vars, char **argv);
27
28 char compressChar(char character);
29 char decompressChar(char character);
30
31 #endif
```

Listing 44: `main.cc`

```
1  #include "main.ih"
2
3  int main(int argc, char **argv)
4  {
5      Vars vars;
6      arguments(vars, argc, argv);
7      process(vars);
8  }
```

Listing 45: `arguments.cc`

```
1  #include "main.ih"
2
3  void arguments(Vars &vars, int argc, char **argv)
4  {
5      vars.action = USAGE;
6      if (argc == 2)
7          vars.action = ENCRYPT;
8      else if (argc >= 3)
9          vars.action = DECRYPT;
10
11     readFile(vars);
12     readCipher(vars, argv);
```

```
}
```

Listing 46: `compresschar.cc`

```
// Characters \t and \n take the first two indices,
// the other printable characters take the rest, starting at ' '.
char compressChar(char character)
{
    switch (character)
    {
        case ('\t'):
            return 0;
        case ('\n'):
            return 1;
        default:
            return 2 + character - ' ';
    }
}
```

Listing 47: `decompresschar.cc`

```
// Numbers 0 and 1 are mapped to \t and \n,
// the other numbers are mapped to the printable characters
// starting with ' '.
char decompressChar(char character)
{
    switch (character)
    {
        case (0):
            return '\t';
        case (1):
            return '\n';
        default:
            return character + ' ' - 2;
    }
}
```

Listing 48: `process.cc`

```
#include "main.ih"

// The encryption/decryption occurs as follows.
// 1: The characters are compressed into the range 0 to
//     alphabetSize - 1.
// 2: The numbers are shifted by encryption/decryption.
// 3: the number range is transformed back into the
//     printable characters.
void process(Vars const &vars)
{
    int alphabetSize = 2 + '~' - ' ';
    int cipherSize = vars.cipher.length();

    size_t step = 0;
```

```
15      for (char character : vars.input)
16      {
17          char cipherChar = compressChar(vars.cipher[step]);
18          char newChar = compressChar(character);
19
20          if (vars.action == ENCRYPT)
21              newChar = (newChar + cipherChar) % alphabetSize;
22          else
23              newChar = (alphabetSize + newChar - cipherChar) %
    alphabetSize;
24
25          newChar = decompressChar(newChar);
26          cout << newChar;
27          step = (step + 1) % cipherSize;
28      }
29  }
```

Assignment 17, Create a library

We were tasked with creating a library, show how to do it and how we compiled it. All functions are in a file called library.a and the command that we used to create it is:

```
ar rvs librvalue.a arguments.o compresschar.o
decompresschar.o process.o
```

And finally, we compiled the program typing make in the terminal with this makefile file:

Listing 49: `makefile`

```
1  DEPS = main.ih
2
3  %.o: %.cc $(DEPS)
4      g++ -std=c++11 -Wall -c -o $@ $<
5
6  main: main.cc librvalue.a
7      g++ -std=c++11 -Wall -o main main.cc librvalue.a
```