Department of Computer Science

COS132 2015
Practical 6

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Ms Mary Adedayo

0 Introduction

0.1 Uploads and Deadline

The uploads will become available on Monday 18 May.

The submission deadline for this practical is **22 May 2015 at 19:30**. Make sure that you have submitted your code to Fitchfork by then. **No late submissions will be accepted**.

0.2 Challenges

Note that the challenges are **NOT** part of the assignment. Each of them is given as a challenge to those who find our usual assignments too trivial. It does not contribute to your marks. It is not expected from the Teaching Assistants to assist you to complete these assignments. It is intended for you to figure it out on your own and stimulate you to use the opportunities you have to develop your programming skills to the fullest. There will also not be any uploads on these.

0.3 Plagiarism Policy

All submissions for this practical should be documented as specified in our plagiarism policy. Refer to the notes about our plagiarism policy which can be found on the COS132 website on ClickUP.

0.4 Download and extract

Create a directory called Practical6. Download the archive called `Practical6.tar.gz` and save it in this directory. Extract the archive.

After extracting this archive the directory you have created for this practical should contain the files and sub-directories shown in the following hierarchy. You are advised to create sub-directories for other tasks.

```
Practical6
└── Practical6.tar.gz
    ├── StudentScores
    │   ├── scores.dat
    │   ├── processScores.h
    │   └── scores.cpp
    ├── StringCounts
    │   ├── stringCount.h
    │   └── strings.cpp
    ├── CustomerAccounts
    │   ├── Account.cpp
    │   ├── Account.h
    │   ├── makefile
    │   └── samplerun
    │       ├── accounts.dat
    │       ├── modified.dat
    │       └── terminal-snapshot.txt
    ├── Challenge_substring
    │   └── main.cpp
    └── Challenge_broken_classes
        ├── BankAccount.cpp
        ├── BankAccount.h
        ├── ChequeAccount.cpp
        ├── CheckAccount.h
        ├── CreditAccount.cpp
        ├── CreditAccount.h
        ├── SavingsAccount.cpp
        ├── SavingsAccount.h
        └── main.cpp
```

1 Student Scores

Write a program that reads the surnames and scores of students from a file and display the records in ascending order of the surnames. Your program should also display the highest and lowest scores with the surname of the student that obtained the scores, the mean, mode, and median of the scores. The file contains the surname and mark obtained by each student on separate lines.

Based on the number records in the file, your program should **dynamically allocate** two arrays that are large enough to hold all the records. One pointer array must be of `string` type for the surnames and the other array must hold `int` type for the marks.

For your convenience, we have provided a file called `scores.dat` which contains the scores obtained by students in a test, which you may use to test your program. You are encouraged to create more test data to test your program thoroughly.

In order to complete the task, you implement the following functions described below.

Functions

unsigned getRecordsNumber(string fileName)
: This function determines the number of student records from a file `fileName`. The returned value should be used to allocate the size of the `string` array in which the names will be stored and `int` array in which the marks will be stored.

void readRecords(**int** * scores, string * names, string fileName)
: This function reads the content of the file with the name `fileName` and stores the mark and name of each student record in the file into the corresponding array.

void sort(**int** * scores, string * names, **size_t** size)
: This function sorts the two arrays. The array containing surnames is sorted in an ascending order as specified from the previous subsection. While sorting the names the corresponding scores array must also be modified so that the name and mark in each index position of the arrays match accordingly. Both `scores` and `names` are arrays of size `size`.

int modeFrequency(**const int** * scores, **size_t** size)
: The mode of the scores array is the score that occurs most often. It is possible to have bimodal values - that is, cases where there is more than one mode because two or more scores occur the same number of time. There might also be cases when there is no mode in the scores (that is, all scores appear once). This function must return the frequency of the modal value(s).

  The function `printModeVaules(const int *, size_t, int)`, will be used to print the mode values that have this number of frequency.

void printModeValues(**const int** * scores, **size_t** size, **int** modeAppearance)
: Prints all numbers that appears `modeAppearance` times in a non-empty array `scores`. If there is no mode (that is, all scores appear once), then print the value -1.

double mean(**const int** * scores, **size_t** size)
: The mean is the average score of all the students.

double median(**const int** * scores, **size_t** size)
: The median is the value in the middle value when the scores are sorted in ascending or descending order. If there is an even number of records, the median is the average of the two middle scores.

int highest(**const int** * scores, **size_t** size)
: Returns the highest score from the array containing of scores of students.

int lowest(**const int** * scores, **size_t** size)
: Returns the lowest score from the array containing of scores of students.

void printScores(**const int** * scores, **const** string * names, **size_t** size)
: This function prints the records of student names and marks from the arrays in the order in which they are stored in the arrays. Here is a sample print format.

```
Name            Score
--------------------
Betty              45
Carl*               0
Mhlangu            85
```

void printFoundNames(**const int*** scores, **const** string* names, **size_t** size, **int** v)
: For a given a mark or score `v`, this function prints the surname(s) of all students that obtained the given mark.

Note: Test scores must be integer values between 0 and 100. Values outside these range must be reported either as 0 (if negative) or 100 (if more than 100) and the corresponding surname in this case must be suffixed with * . For the standard version of the assignment you may assume that surnames do not contain spaces or punctuation.

The following is a sample run of the program using the given test data:

```
Enter the file name:  scores.dat
The records in ascending order of surnames are:

 Name            Score
 --------------------
 Betty              45
 Carl*               0
 Mhlangu            85
 Naughty            30
 Pieterse          100
 Solomon            65
 Walter             70
 Zubi               11

Highest score:  100 Pieterse
Lowest score:  0 Carl*
Mean score:  50.75
Mode score:  -1 Modal value occurrences is 1 time
Median score 55.00
```

Given Files

You are given the header file `processScores.h` and a framework for the driver program `scores.cpp`. You are expected to write the functions in `processScores.cpp` and create your own makefile to test your program.

- Use the following command to create a tarball with these files for upload purposes:

  tar -cvz *.cpp *.h makefile -f scores.tar.gz

- Upload your tarball in the slot(s) containing **Student Scores** in the name.

1.1 Challenge Version

Modify the standard version so that surnames read from file can contain spaces and/or punctuation. Also allow scores with decimal places to be read from file but stored as the nearest integer value.

2 String counts

Write a menu driven program that accepts a c-string from the user and outputs one of the following based on the option selected by the user: the number of vowels in the string, the number of consonants, the number of uppercase letters, the number of lower case letters, total number of letters in the string, total number of digits in the string, the average number of characters per word in the string, or the sentence with the first letter of each word capitalised.

You have been given a file `stringCount.h` that contains the prototype definition for each of the functions required above, the driver containing the main function - `strings.cpp` to test your functions. You are required to write the definition of the functions in an implementation file called `stringCount.cpp` and create your own makefile. Your functions will be tested with a different driver. A sample run of the program is shown below:

```
Enter a string:  She is 8 years old

What would you like to do?
A. Count the number of vowels in the string
B. Count the number of consonants in string
C. Count the number of uppercase alphabets instring
D. Count the number of lower case alphabets in string
E. Count the total number of alphabets in the string
F. Count the total number of digits in the string
G. Find the average number of characters in each word of the string
H. Display the sentence with the first letter of each word capitalised
I. Enter a new string
Q. Quit

Enter your choice:  A
The number of vowels in the string is 5

What would you like to do?
A. Count the number of vowels in the string
B. Count the number of consonants in string
C. Count the number of uppercase alphabets instring
D. Count the number of lower case alphabets in string
E. Count the total number of alphabets in the string
F. Count the total number of digits in the string
G. Find the average number of characters in each word of the string
H. Display the sentence with the first letter of each word capitalized
I. Enter a new string
Q. Quit

Enter your choice:
```

Create a tarball containing all your implementation file and header file and submit it on the CS website in the slot named **String Counts**.

3 Pig Latin

Pig Latin is a constructed language game where words in English language are altered according to a simple set of rules. The rules for changing standard English into Pig Latin are as follows:

- For words that begin with consonant, the initial consonant or consonant cluster is moved to the end of the word, and "ay" is added, as in the following examples:
  - “duck” → “uckday”
  - “glove” → “oveglay”
- For words that begin with a vowel, "way" is added at the end of the word, as in the following examples:
  - “egg” → “eggway”
  - “apple” → “appleway”

Specifications

- Write a function called `isVowel`. It should have one character argument and return **true** if the argument is a vowel and **false** otherwise.
- Write a function with the following prototype

  ```
  void toPigLatin(const char[], char[]);
  ```

  The first parameter is the input. After calling the function, the second parameter should contain the Pig Latin of the given word.
- Write a program that uses the above functions. It should prompt the user for a word and convert the given word into Pig Latin. It should accept a word in mixed case and output its Pig Latin in lower case only.
- The program should produce an error message if the user input is not a character string which contains only alphabetical characters.
- Create the following files that should be included in your upload:
  - A header file called **piglatin.h** which contains the function prototypes.
  - An implementation file called **piglatin.cpp** that contains the implementations of these functions.
  - A driver file that implements your program.
  - A **makefile** to compile and run the system.
- Upload your tarball on the CS website in the slot(s) containing **Pig Latin** in the name.

The following example is a test run of the program:

```
Enter an English word:  inbox
The Pig Latin for inbox is inboxway
```

3.1 Challenge Version

You can enhance the program to apply English rules regarding words that are pronounced as if they start with vowels but spelled starting with a silent character. For example the H in words beginning with H is usually silent in words that are of French origin such as **hour** and **honest**.

- If the first consonant character is silent in a word which is pronounced as if it starts with a vowel, the word should be treated as if it is starting with a vowel. The following are examples:
  - "hour" → "hourway"
  - "honenst" → "honestway"
- If the first consonant character is not silent in a word, the word should still be treated as if it is starting with a consonant. The following shows how they should be treated:
  - "hope" → "opehay"
  - "happy" → "appyhay"

In words where the letter **y** is pronounced like a vowel, the letter **y** can be considered as a vowel.

- If the word has a **y** that is pronounced like a vowel, **y** should be treated as a vowel. The following are examples:
  - "cyclone" → "yclonecay"
  - "yttrium" → "yttriumway"
- If **y** is not pronounced like a vowel, **y** should still be treated as a consonant. Often **y** is pronouced like J. The following shows such a word and how it should be treated:
  - "young" → "oungyay"

4 Fractions

Write a program that allows the user to enter one or more fractions and then displays the sum of the fractions. Fractions must be stored in a structure containing two elements: the numerator and the denominator. The program users to enter fractions one after the other until the user enters zero as the value of a denominator of a fraction. The program should display the addition of all the fractions entered to this point in a fractional form. The entered fractions as well as the final result must be displayed in the lowest terms of the fraction and/or as a integer value if the resulting denominator is 1 after simplifying. You need to implement the functions to add the fractions and simplify them using the prototypes given below:

- **fractionType** addFractions(**fractionType** fractions[], **size_t** length)
- **fractionType** simplyFraction(**fractionType** f)

where `fractionType` is a `struct` that stores the numerator and denominator of a fraction as `int` values called `numerator` and `denominator`.

A sample run of the program is shown below:

```
Fraction Adder
Enter the numerator for fraction 1:  4
Enter the denominator for fraction 1:  2
Enter the numerator for fraction 2:  4
Enter the denominator for fraction 2:  5
Enter the numerator for fraction 3:  -7
Enter the denominator for fraction 3:  4
Enter the numerator for fraction 4:  1
Enter the denominator for fraction 4:  0

2 + 4/5 + -7/4 = 21/20
```

Note: The members of the `struct`, `numerator` and `denominator` must be integer values. You may assume that the number of fractions to be added will not be more than 20 and that input for each numerator and denominator is entered correctly as an integer value.

You are expected to create your own files, named: **makefile**, **main.cpp**, **fractions.cpp** and **fractions.h** . Create a tarball containing all your files and submit it on the submission slot(s) containing **Fractions** in the name.

4.1 Challenge Version

You can enhance the program such that if the user enters a real number at a prompt instead of an integer, the program should allow the user to re-enter the value a second time. If a real number is still entered at the second prompt, the program should then truncate the real number to an integer value for that input.

5 Customer Accounts

Write a program that reads and stores the records of customer accounts into a binary data file. A structure (given in `Account.h`) is used to hold a record in memory. Each customer record contains the Name, Address, City, Postal code, Telephone number, and Account balance fields of the customer.

The program should have a menu that allows the user to perform the following operations:

1. Load records from the data file into memory (user types in the file name)

2. Search for a particular customer's record and display it.
3. Edit the displayed customer record's information.
4. Delete the displayed customer's record.
5. Create and enter details for a new record.
6. Write the records in memory back to the file once the user is done.

You are given an outline of the program in the following files. Complete the program by adding code to these files.

- Implementations of the functions that operate on records and data files in `Account.cpp` (prototypes are in `Account.h`).
- A driver program `banking.cpp` with a main function that interacts with the user.

Records should be held in memory using the STL `vector` container type. It is easy to add items to a vector using `vector::push_back` or to remove them as demonstrated in the `deleteAccount` function already given in `Account.cpp`

The following specifications must be taken into account in each function.

- To search for a record, the user must enter all or a consecutive part of the customer's name. The program should first search if a record with the name exists. If the given name does not exist in the file, the program should report that the record was not found.
- Before deleting a record that exists, a prompt to confirm deletion must be given to the user. A deleted record must no longer be present in the file once the file is saved.
- To change a record, the user should be prompted to change each field with the same prompts that are used when creating a new record. Only this time the existing value for the field is shown with an option to keep it. Only the fields that need to be changed must be re-entered by the user. The changed record should replace the old version of the record in the file.
- After performing one operation, the program must allow the user to select another option from the menu.
- Your code may assume that there are no duplicate records in the file.

Binary file format

All fields except balance are fixed-size character arrays. Therefore an entire Account struct (as defined in `Account.h`) can be written or read at once. For example after opening a binary file for writing:

```
file::write(reinterpret_cast<char*>(&my_account), sizeof(Account));
```

Accounts occur directly after one another in the binary file format; no other values are written in between.

Submission

After testing that your program works, create a tarball that contains all your implementation and header files and a makefile. Submit it to Fitchfork via the CS COS132 website. There are multiple upload slots for this task (all contain **"Customer Accounts"** in the name). You should upload to all the slots, not just one.

Example input/output

The function `displayAccount` should output account data on four lines: a line for name, followed by a line for the address (consisting of the street, a comma and space, then the city, followed by the post code in parentheses), a line for phone number, and finally a line for account balance. For example:

```
Name: Lizel Thompson
Address: 14 Bark street, Essex (0203)
Telephone: 0293085790
Balance: 400.00
```

The function `inputAccount` should prompt for and read data using exactly six lines. Note that this function may be used to edit an existing account (passed by reference); in such a case the user may leave an input blank to indicate that it should retain its existing value (except balance, which should always be entered by the user). For example, if the user is changing only the phone number of the following customer:

```
Enter name (Bernard Dubois):
Enter street (7 Great Dane str):
Enter city (Wildridge):
Enter post code (0729):
Enter phone number (0368413657): 0825551234
Enter account balance (-40.26): -40.26
```

An extensive example run is given in the `samplerun/` subdirectory. This directory contains three files:

- `accounts.dat` is an example account data file loaded into the program.
- `terminal-snapshot.txt` contains example input and output as it would appear on the terminal for a correct version of the program.
- `modified.dat` is the account data file written by the program after modifying the data loaded from `accounts.dat` according to the input given in `terminal-snapshot.txt`

5.1 Challenge Versions

- Modify the standard version such that duplicate records are possible. The program should warn the user when creating a duplicate record. The user should also be allowed to select which duplicate record to delete in the case of a delete operation.

- Make searching case-insensitive.
- Allow the user to search using an extended regular expression (regex) instead of a substring.
- Allow the user to edit the text that is shown when a record is by using forward and back arrows, backspace, delete and insert.

6 Challenge Tasks

6.1 Substring count

Write a function named `substringCount` that takes two C++ `std::string` parameters, and returns an `unsigned int`. The first parameter is the "haystack" and the second is the "needle". Your function should return the number of times that the needle occurs inside haystack. Both parameters should be passed as const reference.

For example, in the haystack `"This is the task"` the needle `"th"` occurs twice. Note that the search must be case- insensitive: `"Th"` is considered the same as `"th"`.

Occurrences of needle may not overlap with itself inside haystack. For example in the haystack `"kekekeke"` the needle `"keke"` occurs exactly twice (if overlapping occurrences were considered, the needle occurs three times)

An example main file is already given to you; you are free to modify this file. Write the implementation of `substringCount` inside the file `substringCount.cpp` and write a prototype for it in the header file `substringCount.h`. You should also write your own `makefile` and compile with the given `main.cpp` file.

Hint: It is possible to correctly implement this function with character-by-character loops. However it would be easier to use a certain member function of `std::string` to do most of the work (see pages 588-589 of Gaddis).

6.2 Class inheritance and method overriding

You are provided a number of files that contain classes that represent bank accounts. There are several syntactical and semantic errors scattered throughout the files. Fix the classes and write a complete test suite for them in `main.cpp`.

`BankAccount` is the abstract base class from which the other three derive: `ChequeAccount`, `SavingsAccount` and `CreditAccount`.

A typedef is used in `BankAccount.h`: `money_t` is simply a different name for **`double`**. Using `money_t` in code makes it clear to other programmers that you are working with a currency and not eg. a fraction or weight value.

`BankAccount` defines the following members:

- `money_t balance;` a protected member that stores the current balance of the account.

- A constructor which initializes the account's balance to a certain amount.
- Mutators `withdraw` and `deposit` that modify the amount if possible and return true or false depending on whether it was possible to do so.
- A const accessor `getBalance` that returns the current balance.
- `interest` which returns the interest for the account (a fraction of the current balance). This is not implemented in `BankAccount`, instead it will be implemented separately in each subclass since different accounts may have different ways of calculating interest.
- `monthly_levy` which returns the monthly administration cost of the account. As with `interest` this function will be implemented not in BankAccount itself but in each of its subclasses.

`SavingsAccount` is the simplest class. The interest for a savings account is 3% of the balance. The levy for personal accounts is R3.00.

A `ChequeAccount` object can be either a personal account or a business account. The monthly levy for personal cheque accounts is R40. The monthly levy for business accounts is R150. The interest rate is 9% in either case.

Depositing should always work for any account (ie. increase balance and return true). Withdrawal should fail for a savings and cheque accounts if the amount to be withdrawn is greater than the balance.

`CreditAccount` is slightly more complicated. A credit account has a credit limit (stored in a class member variable and initialized by an argument to the constructor). When withdrawing from a credit account, withdrawal does not fail if the balance would fall below 0. Instead, withdrawal succeeds if balance does not become less than the credit limit.

The levy for a credit account is R5. The interest rate for `CreditAccount` depends on whether the balance is negative or not. If the balance is negative, a higher interest rate of 12% is used. Otherwise a much lower interest rate of 1% is used.

Complete and debug the classes, then test your implementations in `main.cpp`. You may write your own `makefile`.

7 Marks Allocation

Task	**Percentage**	**Maximum mark**
Student Scores	24%	**getRecordsNumber** [4 marks], **Main** [15 marks], **highest** [4 marks], **lowest** [4 marks], **mean** [3 marks], **median** [5 marks], **sort** [5 marks], **printModeValues** [4 marks], **readRecords** [6 marks], **modeFrequency** [4 marks]
String Counts	22%	20 marks
Pig Latin	24%	**isVowel** [10 marks], **toPigLatin** [15 marks], **main** [15 marks]
Fractions	0%	No upload link
Customer Accounts	30%	**displayAccounts** [12 marks], **findAccount** [10 marks], **inputAccount** [14 marks], **loadAccounts** [13 marks], **saveAccount** [13 marks], **main file** [13 marks]
Total	**100**	