Advanced Graphics

# Lab 3 – Practice with raw pointers

## Submission: One word document containing all your code. Due before the next class.

## Goal

Practice using the following:

* Simple file documentation (at minimum your name, date, filename and purpose). Not necessary to document members or functions.
* Using header and source files must have the extensions hpp and cpp respectively.
* Using a static member in a class
* Using raw pointers and dynamic memory allocation and deallocation (using the new and delete operators).
* Using arrays, passing to function and return from functions.
* Implementing a sort algorithm. (You will use one from the internet and benchmark it)

# Requirements

Implement an Account class using a C++ program. Match the output against the sample output given the provided test harness.

## The Account Class

The members must be declared in a header file and implemented in a source file. You may also have another source file containing the test harness.

|  |
| --- |
| **Account**  Class |
| **Fields** |
| $- LAST\_NUMBER : **int** = 1000  - account\_number\_ : **string**  - balance\_ : **double** |
| **Methods** |
| + «constructor» Account( balance : **double**)  + «destructor» ~Account()  + get\_balance() : **double**  + print\_info() : **void** |

## Description of class members

### Fields

There are three int fields that are private. One of the fields is also static.

balance\_ – this is a private double field that represents the balance of this object.

account\_number\_ – this is a private string field that represents the unique identifier of this object.

LAST\_NUMBER – this is a private static int field that the last number that was used to generate an account number. This is initially set to 1000 and is incremented in the constructor.

### Constructors

**Account(double balance)** – This public constructor takes one double argument and and does the following:

* Assigns the argument to the appropriate field.
* It also assigns the static field LAST\_NUMBER to the field account\_number\_ (you will have to convert the int to a string).
* Increments the static field LAST\_NUMBER, so that future objects will access to a unique value.
* Finally, it outputs a suitable text to the console that the constructor was invoked.   
  This statement maybe commented out to reduce screen clutter during development.

### Destructors

**~Date()** – This public destructor does not implement any logic at this time. It simply output a suitable text to the console indicating that the destructor was invoked.   
This statement maybe also be commented out to reduce screen clutter during development.

### Methods

There are two public methods:

void Account::print\_info() – This method outputs the two private fields (account\_number\_ and balance\_) to the console. See the program output for clues on implementing this method.

double Account::get\_balance() – This method returns the two private field balance\_.

## Other supporting logic required

The following methods must be declared and implemented to successfully run the test harness.

void test\_account() – This method is call in the main(). Copy and paste the following statements to exercise your application:

srand(123456789); //use your student number

const int MAX\_BALANCE = 100000;

const int SIZE = 20; //change this number to investigate performance

//Week03::Account a1(rand() % MAX\_BALANCE);

//Week03::Account a2(rand() % MAX\_BALANCE);

//Week03::Account a3(rand() % MAX\_BALANCE);

//Week03::Account a4(rand() % MAX\_BALANCE);

//a1.print\_info();

//a2.print\_info();

//a3.print\_info();

//a4.print\_info();

std::cout << "\nCreating the list of accounts:\n";

Week03::Account\*\* accounts = create\_accounts(SIZE, MAX\_BALANCE);

std::cout << "\nPrinting the list of accounts:\n";

//print\_accounts(accounts, SIZE);

std::cout << "\nSorting the list of accounts:\n";

sort\_accounts(accounts, SIZE);

std::cout << "\nPrinting the list of accounts:\n";

print\_accounts(accounts, SIZE);

std::cout << "\nDeleting the accounts:\n";

for (int i = 0;i < SIZE;i++)

{

delete(accounts[i]);

}

Week03::Account\*\* create\_accounts(int size, int max\_balance) – This method does the following:

* Creates an array to store the appropriate number of Account pointers.
* Initialize each item to an account object. Use the value rand() % max\_balance as argument to the constructor.
* Return the array.

void print\_accounts(Week03::Account\* accounts[], int size) – This method will invoke the print\_info() method on all of the items in the first argument.

void sort\_accounts(Week03::Account\*\* accounts, int size) – This method will sort the first arguments according to balances. It does not outputs anything to the screen. This method also emits some performance information. The number of swaps and the time taken to complete the sort.

The timing will be covered in class. The code will be in the appendix of this document.

The members must be declared in a header file and implemented in a source file. You may also have another source file containing the test harness and the helper functions.

### Test harness

Call the test\_accounts() in your main method.

### Sample output

Try to understand why and when constructors and destructor are called.

Creating the list of accounts:

Printing the list of accounts:

Sorting the list of accounts:

86swaps

739us (0.739ms)

Printing the list of accounts:

List of accounts:

1003: balance: 30354

1012: balance: 27918

1001: balance: 26974

1015: balance: 23599

1014: balance: 22768

1007: balance: 21981

1019: balance: 21512

1004: balance: 18709

1010: balance: 18110

1013: balance: 17989

1006: balance: 16906

1005: balance: 15861

1017: balance: 15601

1002: balance: 13551

1000: balance: 13259

1009: balance: 12911

1008: balance: 8603

1016: balance: 7712

1018: balance: 7038

1011: balance: 3228

Deleting the accounts:

### Appendix

Code for the timer.hpp file.

/\*\*

\* Author : Yan Chernokow

\* Filename: timer.hpp

\* Purpose : provide a simple way to time code

\* Date : February 2, 2022

\*/

#pragma once

#include <iostream>

#include <chrono>

class Timer

{

public:

Timer()

{

m\_StartTimepoint = std::chrono::high\_resolution\_clock::now();

}

~Timer()

{

Stop();

}

void Stop()

{

auto endTimepoint = std::chrono::high\_resolution\_clock::now();

auto start = std::chrono::time\_point\_cast<std::chrono::microseconds>(m\_StartTimepoint).time\_since\_epoch().count();

auto end = std::chrono::time\_point\_cast<std::chrono::microseconds>(endTimepoint).time\_since\_epoch().count();

auto duration = end - start;

double ms = duration \* 0.001;

std::cout << duration << "us (" << ms << "ms)\n";

}

private:

std::chrono::time\_point<std::chrono::high\_resolution\_clock> m\_StartTimepoint;

};