The following materials have been collected from the numerous sources such as Stanford CS106 and Harvard CS50 including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

Problem Set 2 – Sorting

Table of Contents

Getting Started

First of all, get GitHub/idebtor/**nowic/psets/pset02** and you may see the following files:

- **pset02.pdf** – All about the Problem set 02.

- **random.pdf** – reading material about random number generation.

- **csort**.cpp – a C source code that has a selection sort function.

- **quicksort.cpp, bubble.cpp, insertion.cpp** – Sorting algorithms. Assume that you already have **selection.cpp** through **lab02.**

- **sortDriver1.cpp** – Skeleton code that runs various sorting algorithms interactively. You may watch out the usage of **clock()** that measures the elapse time of the processor.

- **sortx.exe –** A solution provided to compare it with your work.

Warming-up

Compile and run the following C code, **csort.cpp**, and make sure that you understand how the code works.

```c
// csort.cpp - Reference: Fundamentals of Data Structures by Horowitz, Sahni

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX_SIZE 100
#define SWAP(x, y, t)  ((t) = (x), (x) = (y), (y) = (t))
void sort(int[], int);

int main(int argc, char *argv[]) {
  int i, n;
  int list[MAX_SIZE];
  printf("Enter the number of samples: ");
  scanf("%d", &n);
  if (n < 1 || n > MAX_SIZE) {
      printf("Improper value of n");
      exit(EXIT_FAILURE);
  }
  printf("\nUnSorted array:\n");
  for (i = 0; i < n; i++) {        // randomly generate numbers
      list[i] = rand() % 1000;
      printf("%d  ", list[i]);
  }

  sort(list, n);

  printf("\nSorted array:\n");
  for (i = 0; i < n; i++) {
      printf("%d  ", list[i]);
  }
  printf("\n");
}

void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n - 1; i++) {
      min = i;
      for (j = i + 1; j < n; j++)
          if (list[j] < list[min])
              min = j;
      temp = list[i];
      list[i] = list[min];
      list[min] = temp;
  }
}
```
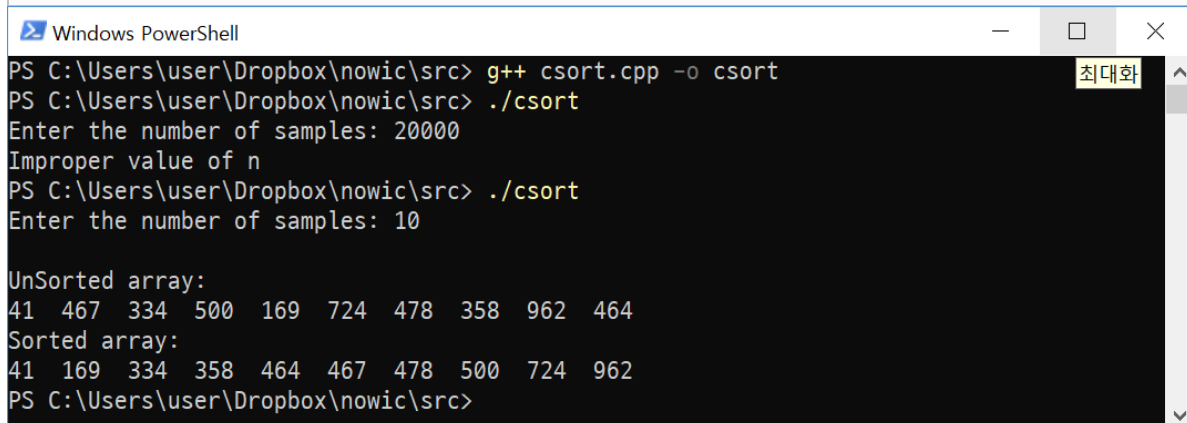
```
Windows PowerShell                                        —    □    ✕

PS C:\Users\user\Dropbox\nowic\src> g++ csort.cpp -o csort       최대화
PS C:\Users\user\Dropbox\nowic\src> ./csort
Enter the number of samples: 20000
Improper value of n
PS C:\Users\user\Dropbox\nowic\src> ./csort
Enter the number of samples: 10

UnSorted array:
41  467  334  500  169  724  478  358  962  464
Sorted array:
41  169  334  358  464  467  478  500  724  962
PS C:\Users\user\Dropbox\nowic\src>
```

Last updated: 3/8/2019

Rewriting C Program (csort.cpp) into C++

In this project, let's rewrite **csort.cpp** in C++ such that it follows the C++ coding convention and works much better. There are a few things that can be improved in this program and set them as your **[Program Specification]** as a guideline.

1. The magic number such as **MAX_SIZE** should not be used if possible.
2. Ask the user to specify the sample size (number of entries) if he/she does not pass it through the command line. You may quit if the user enters **0**. Otherwise keep asking a proper number of samples to sort.
3. Based on the sample size, **list[ ]** must be dynamically allocated for extensibility. You may use **malloc()** and **free()** or **new and delete in C++.**
4. The error message should tell what went wrong and what is a good value.
5. The main function should return a status, **EXIT_SUCCESS** or **EXIT_FAILURE**.
6. The **sort()** should be named something like **selectionSort**.
7. It is better to separate the function and its driver. In this problem set, we have a few sort functions separately from **sortDriver1.cpp** that tests them, respectively.
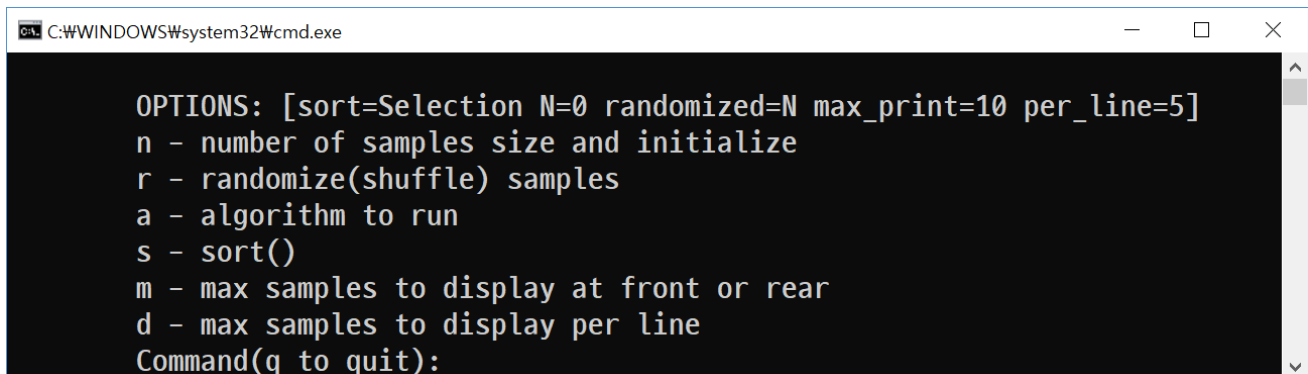
When I say C++ coding convention, don't use the following C style code:

- C style include files such as stdio.h, stdlib.h, math.h, etc.
- C style functions such as printf, fprintf, sprintf, strcpy, strcat, etc
- C style macros such as #define MAX_SIZE   10
- C style malloc functions such as malloc, calloc, realloc, free.

Step 1: Handling an arbitrary number of samples

We want to sort an arbitrary number of samples in sorting and display its result. While you follow the skeleton code, **sortDriver1.cpp,** and implement the code that works like **sortx.exe** provided as a sample.

Begin with **sortx.exe** provided with you.

```
C:\WINDOWS\system32\cmd.exe                                        —    □    ×

    OPTIONS: [sort=Selection N=0 randomized=N max_print=10 per_line=5]
    n - number of samples size and initialize
    r - randomize(shuffle) samples
    a - algorithm to run
    s - sort()
    m - max samples to display at front or rear
    d - max samples to display per line
    Command(q to quit):
```

This code shows some sorting options such that the user can set it up as he/she wishes interactively.

For example, the number of samples to sort can be set freely and select one of sorting algorithms available. Also, the input and output data can be display as user's specification. It is useful when you can check the sorting of the large data set. I recommend that you fully understand those options given, especially, **options m and d.**

Last updated: 3/8/2019

Many options in the following menu are self-explanatory. The first line which begins with "OPTIONS" shows the current status of the code. The skeleton code in **sortDriver1.cpp** also offers you to follow when you code.

Option n

Get an integer input from the user.
If the user enters wrong, keep on asking "Retry".

If user's input is less than 1, display an error message and go to the menu.
If user's input is valid, do the following.

1. set N, the number of samples, with the new value that the user entered.

2. Before allocating the new list, free the old list if not NULL

3. Allocate memory for new data samples

4. Fill the list with numbers from 0 to n - 1.

5. Print the new list and go the menu.

Option a

Provide the sorting algorithms such as Bubble, Insertion, Quicksort, and Selection sort. The use may select one of the **algorithms**.

You may follow suggestions described in the code. It may be a good chance to use an enumerated data type. Here is an example which is a self-explanatory.

```
// definition of Color type
enum Color {RED, GREEN, BLUE, WHITE, PURPLE, … };

// variable declarations
enum Color x, y, z;

x = BLUE;
y = WHITE;
z = PURPLE;
if(x == BLUE){
   …
}

switch(y){
case WHITE:
   …
}
```

Instead of Color, we may name our algorithms as shown below:

```
enum algorithm_list {BUBBLE, INSERTION, QUICKSORT, SELECTION};
enum algorithm_chosen;      // set it menu item a
                           // used in menu item s with a switch statement
```

Option r

If the list is newly created or sorted, the list is filled with sorted numbers. Therefore, it is common that the list elements are shuffled before sorting. Do this coding with the following function proto-type.

```
void randomize(int list[], int n);
```

For every sample, starting from the first element in the list, it is swapped with the element randomly selected by the index generated by a 'real' (not pseudo) random number out of from 0 to N-1.

Hint: Refer to **rand()** and **srand()** functions explained random.pdf.

Option s

This option executes the algorithm which is already set previously. According to the options the user sets, execute sorting itself and display its result.

Option m and Option d

In these options, you write the following function.

```
void printList(int *list, int n, int max_print, int per_line);
```

When we have a long list, we want to print some in the front part of the list and some in the rear part of the list. The **max_print** specifies the number of samples in either front or rear part of the list. The **per_line** determines how many samples prints per line. If max_print is larger than or equal to the sample size N, prints **max_print/2** samples only.

For example:

```
Case 1) N = 12, max_print = 100, per_line = 20
        0    1    2    3    4    5
        6    7    8    9    10   11
Case 2) N = 12, max_print = 100, per_line = 4
        0    1    2    3
        4    5
        6    7    8    9
       10   11
Case 3) N = 12, max_print = 4, per_line = 6
        0    1    2    3
        8    9    10   11
Case 4) N = 100, max_print = 10, per_line = 6
        0    1    2    3    4    5
        7    8    9
       91   92   93   94   95   96
```

        97   98   99

There should be a line feed between the front part and the rear part.

Observe the sample runs using sortx.exe provided with you.

Step 2: Using a function pointer:

Function Pointers provide some extremely interesting, efficient and elegant programming techniques. You can use them to replace switch/if-statements, to realize your **own _late-binding_** or to implement **_callbacks._**

Unfortunately - probably due to their complicated syntax - they are treated quite carelessly in most computer books and documentations. If at all, they are addressed quite briefly and superficially. They are less error prone than normal pointers because you will never allocate or deallocate memory with them. All you've got to do is to understand what they are and to learn their syntax. But keep in mind: Always ask yourself if you really need a function pointer.

It's nice to realize one's own **_late-binding_** but to use the existing structures of C/C++ may make your code more readable and clearer.
(http://www.newty.de/fpt/intro.html#why)

You may watch this lecture (https://dojang.io/mod/page/view.php?id=592) about the function pointer in Korean.

Definition

Function Pointer is a pointer, i.e. a variable, which points to the address of a function. You must keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus, a function in the program code is, like e.g. a character field, nothing else than an address. It is only important how you, or better your compiler/processor, interpret the memory a pointer point to.

Instead of referring to data values, a function pointer points to executable code within memory. When dereferenced, a function pointer can be used to invoke the function it points to and pass its arguments just like a normal function call. [from Wikipedia]

Function pointers can be used to simplify code by providing a simple way to select a function to execute based on **_run-time values_**. It is so called late-binding.

A function pointer always points to a function with a specific signature! Thus all functions, you want to use with the same function pointer, must have the **same parameters and return-type!**

Simple Example

Suppose that we have a very simple function to print out the sum of two numbers and returns the sum. Let us see how we can create a function pointer from there.

```
#include <iostream>
using namespace std;

int fun(int x, int y) {                    // function implementation
  return x * 2 + y;
}

int foo(int x, int y) {
  return x + y * 2;
}

int add(int x, int y) {
return x + y;
}

void main() {                              // using function pointer
  int (*funcPtr) (int, int) = fun;         // & is not needed
  cout << "funcPtr() returns " << funcPtr(2, 3) << endl;
  funcPtr = foo;
  printf("funcPtr() returns " << funcPtr(2, 3) << endl;
}
```

Sample run:

```
funcPtr() returns 7
funcPtr() returns 8
```

Facts About Function pointers

Differences from normal pointers:

A function pointer points to code, not data. Typically, a function pointer stores the start of executable code.

We do not allocate nor de-allocate memory using function pointers.

Same as normal pointers;

We can have an array of function pointers.

```
int main() {
   // fp is an array of function pointers
   int (*fp[])(int, int) = { fun, foo, add };
   for (int i = 0; i < 3; i++)
     cout << "fp(" << i << ") returns "  (*fp[i])(2, 3) << endl;;
}
```

A function pointer can be passed as an argument and can also be returned from a function. This feature of the function pointer is extremely useful. In OOP, class methods are another example implemented using function pointers.

Make use of a function pointer

Using a function pointer, we would like to simplify our sort driver program a bit.

(1) Copy sortDriver1.cpp to sortDriver2.cpp

(2) Declare a function pointer variable for many sort functions to begin with. For example:

```cpp
void (*sortFn)(int *, int);          // sort function pointer
```

(3) Set the function pointer with a default sort function. This code exists before the menu starts.

(4) As a user selects a sort algorithm at the menu item 'a', update the function pointer with the sort function the user selects.

(5) As a user selects the menu item 's' – sort(), invoke a sort function which is stored in the sort function pointer. This means that your code for cases of 'a' and 's' becomes simple – only several lines. For example,

```cpp
    start = clock();
    sortFn(list, N);
    end = clock();
```

Submitting your solution

- Include the following line at the top of your every source file with your name signed.
  On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  Signed: _____   Section: _____   Student Number: _____
- Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not get sympathy for code that "almost" works.
- If you only manage to work out the Project problem partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit, Due and Grade points

Files to submit: csort.cpp, sortDriver1.cpp, sortDriver2.cpp

Due: 11:55 pm March **13**, 2019

Grade: 4 points
- ■ csort.cpp: 0 point, but required
- ■ sortDriver1.cpp: 3 points
- ■ sortDriver2.cpp: 1 point