

CSIS0396A Programming Methodology and Object Oriented Programming (Class A) Assignment 2

Tutor: gechen@csis.hku.hk, Deadline Mar 21, 2002, 11:59pm.

Create a CVS repository and do the following two mini-projects in it, using the project name as the CVS directory name. Each mini-project should have a Makefile, which must be created at the beginning of the mini-project. We will read the history recorded in CVS, so make sure that you start using CVS at the beginning of the project. Keep meaningful log message throughout. Handin your CVS repository via the hand-in system of our department.

Be reminded to create an empty directory, check-out all the files from the repository and check whether you can make the programs there before you hand-in.

1. (*Project sort: Classes as a “better function pointer”. 30 marks*)

In the lecture, we have presented the quick sort algorithm. In the web page, you can find a simple program `quicksort.cc` which sorts integers using a (hand-coded) quick-sort implementation. Such sorting implementation can be generalized to different ordering functions. The C solution is to pass a pointer to the ordering function to `qsort`, to be used in place of “<” to compare integers. But there is a problem: it is difficult for such functions to use other data (except ugly global-variables). One solution that we will use in this assignment is to encapsulate the “ordering function” as an object. Modify the program as described below:

- Your program should sort **coordinates**, i.e., **pairs of floating point numbers**, rather than integers.
- Define an abstract class to **encapsulate** ordering functions.
- Define derived classes of the ordering function class in **dynamic libraries**. Two such classes (and thus libraries) are required. One, called “`sort_yx`”, should order the coordinates from small to large y and then from small to large x. The other, called “`sort_dist`”, order the coordinates from small to large distance from a fixed point. The fixed point should be read from input when the ordering function object is created.
- Each library should export an object, called *func_record* of the following type:

```
struct TFuncRec {
    char* name;                // The name of the ordering function
    TOrderFunc* (*getOrderFunc)(); // A function that returns an object
    TFuncRec* next;           // For building a linked list
};
```

- The main program should repeatedly asks for a list of coordinates. Then it asks for an ordering function. The program then search in a linked list to check whether the requested function is already there. If not, it attempts to load it from shared library, and store it in the list. Then it creates a new ordering function object, initialize it (possibly causing some questions to be asked to initialize the order function object), sort the coordinates, and print the sorted list. Example run:

```
>./quicksort
Length of sequence (0 terminates program)? 2
List of coordinates x and y coordinates, separated by spaces: 2 3 4 5
Order function: sort_yx
2 3
4 5
Length of sequence (0 terminates program)? 2
List of coordinates x and y coordinates, separated by spaces: 2 3 4 5
Order function: sort_dist
Origin (x, y)? 4 4
4 5
2 3
Length of sequence (0 terminates program)? 2
List of coordinates x and y coordinates, separated by spaces: 2 3 4 5
Order function: nofunc
No such function.
Length of sequence (0 terminates program)? 0
```

- Write a Makefile to create the required programs and libraries.
- You may assume that the input are all of the correct form (e.g., when asking for an integer, the user will not type three integers, or a non-numeric string).
- Remember to avoid using any global variable in this assignment.

NB: C++ actually have a similar facility in the `<algorithm>` header. The ordering function object is called a “Compare function”, and can be given to the “sort” template function. Of course, the assignment expect you to write your implementation, rather than calling that of the standard library.

2. (Project guess: Advanced OO programming. 70 marks)

Number guessing is one type of grade-school exercise. A sequence of integers are given to the student, and the student is to give the next number—and explain why he think it is the next number. You are required to write a program which does that: find the “simplest” expression that matches with the given sequence. You can assume that all numbers can be stored in a C++ **int**. Example run of the program:

```
Length of list (0 terminates the program)? 5
Input the list, separated by spaces or newline.
2 3 4 5 6
Next number is 7
Expression: (n+1)
Length of list (0 terminates the program)? 4
Input the list, separated by spaces or newline.
2 6 12 20
Next number is 30
Expression: ((n*n)+n)
Length of list (0 terminates the program)? 0
```

To make things easier, expressions are limited to:

- n , which is the position within the sequence.
- 1 , which is always 1.

- $(\langle Expr1 \rangle \langle binop \rangle \langle Expr2 \rangle)$, where $Expr1$ and $Expr2$ are two expressions, and $binop$ is one of $+$, $-$ and $*$, and performs the corresponding arithmetic computation.
- $(\langle Expr1 \rangle !)$, which performs factorial computations. You can assume that factorials of no more than 10 will ever be required. Factorial of negative numbers return 0.
- $-\langle Expr \rangle$, which is the negation of the expression $Expr$.

What is meant by “simple”? The **complexity** of an expression is defined by the total number of operations and the number of the constant “1” in the expression. An expression is simpler if it has less complexity. You can use the following algorithm to look for the answer:

```
void Search(TSequence &seq) {
    create a queue of expressions;
    insert the expression "n" into the expression
    while (queue is not empty) {
        deque an expression exp
        if (exp is consistent with seq) {
            print the next number and the expression
            deque the whole queue
        } else {
            next = result of modifying exp a bit; // (see below)
            while (next != no more expression) {
                insert next into the queue
                next = result of modifying exp a bit;
            }
        }
    }
}
```

Represent the expression as an object. E.g., the second expression in the example run above is represented by an addition object, which contains one multiplication object and one “N” object. The multiplication object in turn contains two “N” objects. The following member functions will be useful:

- Constructor, copy constructor and destructor. Use deep copying semantics. It is up to you to choose whether to use copy-on-write.
- Print. Each expression should be able to print itself. Depending on the type of objects, this might involve calling print to some inner expression.
- Evaluate. Each expression should be able to evaluate itself, given an integer (the position of the sequence).
- Clone. There must be a way to replicate an expression object without knowing the type of the object. This is done by cloning. The code should essentially just call the copy constructor, but since this is a member function (not a constructor), dynamic binding is allowed.
- GetChangedExpr. This implements the “modify exp a bit” in the algorithm above. It returns a new expression in which **one of the N objects** is changed to another object. When the function is executed multiple times, successive different N objects will be modified to each possible type of expressions. E.g., for the second expression above, the first call returns the expression $(1 * n) + n$, the second call returns $((n + n) * n) + n$,

the third $((n - n) * n) + n$, and so on, until all different operations of the first n is tried, and then it should start modifying the second object and gives $(n * (1)) + n$, etc. When every expression is tried, the function returns a null pointer.

Note: the implementation should be easy. An N object will store an integer which is the next operation to be returned by `GetChangedExpr`. A constant object cannot be changed. A unary object can be changed by recursively changing its operand. And a binary object will store a number which signify whether the next operation to change is in the left or the right operand. The total number of lines involved should be around 10–15, spread among different classes. If you find it more complicated, probably you’ve got the wrong idea.

Requirements:

- One source file must be used solely to define an abstract expression class with a concrete N class.
- Each other derived class of expression should be implemented in a separated file, and should only depend on the features of the above file.
- A list of possible replacements of the N object during a “`GetChanged`” function should be given to the N class in the main function. This list should contains a 1-object, a “+” object, a “-” object, a “*” object, a “!” object and a “-” object. The operands of all these objects should be N objects.
- Write a Makefile to compile the required object files and programs.
- You may assume that the input are all of the correct form (e.g., when asking for an integer, the user will not type three integers, or a non-numeric string).

Hint: Your program can easily eat up a lot of memory. If you are not careful you can easily freeze your computer for a long time before the computer decides that the program uses up too much space and kill it for you. So it is advisable to limit the memory usage of the program, e.g., by running

```
ulimit -v 40960
```

to limit the memory size of your program to 40M. Also, remember to **free all storage** you have allocated once you don’t need it: the program might be executed for a lot of expressions, and garbages created by the program will make it impossible to guess later expressions.

Instruction to hand-in the repository: change directory to the repository (not working directory!), and type “`tar czvf ~/a2.tgz sort guess`”. Then hand-in the file `a2.tgz` in your home directory.