CSC 112 Lab 6
Inventory Database v2.0
Fall 2007

Due
3:00pm Friday, 3/26

1 Overview

This is the same database problem described in the lab 5, with the following changes.

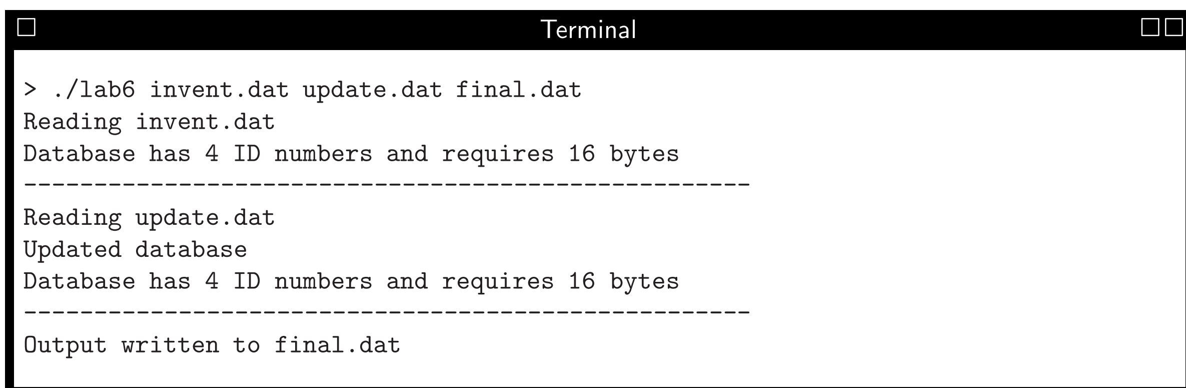
- The program will read the *inventory*, *update*, and *final* data file names from the command line.
- The program will use dynamically allocated arrays instead of static definitions. No wasted space is allowed at any point of your program, dynamically size the arrays when elements are added or deleted.
- The screen output is different than for lab 5.
- Modify the `makefile` to include comments, variables, and a `make clean` option.
- Profile your program, identify the execution *hot spots*, and improve performance.

Note, the product inventory, update, and final data files are as defined in lab 5.

2 Program Description

You must write a C++ program that manages the database files as described in lab 5; however, all arrays must be dynamically sized (no wasted space) and the program must be optimized (described in section 4). Furthermore, file names will be supplied using command line arguments. For example, assume the executable is called `lab6` and the user wishes to execute the program with the files `invent1.dat`, `update1.dat`, and `final1.dat` (the inventory, update, and final data file names respectively). The user would enter the following.

Therefore, the executable file name is followed by the inventory file name, which is followed by the update file name, which is followed by the final file name. If the user omits any of the file names at the command prompt, print an error message and exit the program. Otherwise, the program will read the inventory file and store the **unique** ID values in ascending order in an dynamic array. Note, the array of ID values must be dynamic and sized to store only the unique ID values (**no duplicates and no wasted space at any time**). Once the inventory file has been read the program must indicate the number of ID numbers stored and the amount of memory used (in bytes). After reading the inventory file, update the information in the array using the update file. Once the database (array) is updated, your program should display the number of items stored and the amount of memory used (in bytes). Store the updated information in the final file. Therefore, the final file will contain a list of unique ID numbers sorted in ascending order.



3 Makefiles

As you know, `make` is a utility that performs a series of commands described in a special formatted file. The file `makefile` (or `Makefile`) is the default `make` uses; however, the `make` option `-f` allows you to specify any filename. A makefile contains entries of the form

```
targetName:dependencyList
        commandLine
```

The *targetName* (what the command should generate) is followed by a colon, which is followed by a *dependencyList* (what must be newer than the target to execute the command). The next line consists of a *commandLine*, that should produce the *targetName* (remember, command lines must begin with a tab).

3.1 `makefile` Comments

Placing the character `#` at the beginning of a line in the `makefile` creates a comment. The `make` utility will skip these lines and proceed to the first dependency rule. **For the remaining lab assignments this semester**, you must comment your `makefiles`. Provide your **name**, the **course**, the **assignment number**, and **date**. For example consider the following `makefile`

```
# Programmer: Nomed Nocaed  Course: CSC112 A
# Assignment: 6             Date:   10/22/2007

driver:  driver.o stats.o
      g++ -o driver driver.o stats.o
driver.o:  driver.cpp stats.h
      g++ -c driver.cpp
stats.o:  stats.cpp stats.h
      g++ -c stats.cpp
```

3.2 `make clean`

Often you need to erase all the object files associated with an assignment and compile with a *clean slate*. This can be done using a special target called `clean` that is placed at the end of the `makefile`.

```
# Programmer: Nomed Nocaed  Course: CSC112 A
# Assignment: 6             Date:   10/22/2007

driver:  driver.o stats.o
      g++ -o driver driver.o stats.o
driver.o:  driver.cpp stats.h
      g++ -c driver.cpp
stats.o:  stats.cpp stats.h
      g++ -c stats.cpp
clean:
      \rm -f *.o driver
```

To execute this option enter the command `make clean` at the prompt. In the preceding example, **all** the `.o` files and the executable `driver` will be erased. Do **not** erase source files! Note there is no dependency list, so the commands that follow will always be executed. **For the remaining lab assignments this semester**, your `makefile` must include a `clean` option.

3.3 `makefile` Variables

Often a `makefile` consists of the same options and commands. We can represent these items using `make` variables. Variable (macro) definitions begin at the left margin of the `makefile` and have the form

variableName = *variableDefinition*

For example, our `makefiles` will consistently use the `g++` command for compiling. We could use the following `make` variable to represent this item

```
CC = g++
```

Now the variable `CC` represents the command `g++`. Anytime we need to use the `g++` command in the `makefile`, we would use `$(CC)` instead. *This does not appear to shorten anything, so why should I use it?* Consider the case where you need to compile your program with another compiler. In this situation, you only have to change the `CC` definition. Another use for `make` variables is compiler flags. For example, if we always compile using `-g`, `-pg`, and `-ansi` we could use a single variable,

```
CFLAGS = -g -pg -ansi
```

The `-ansi` compiler flag ensure your code meets the C++ ANSI standards. So compare our new and improved `makefile` to the old style.

```
# Programmer: Nomed Nocaed  Course: CSC112 A
# Assignment: 6             Date:   10/22/2007

CC = g++
CFLAGS = -g -pg -ansi
driver:  driver.o stats.o
      $(CC) -o driver driver.o stats.o
driver.o:  driver.cpp stats.h
      $(CC) $(CFLAGS) -c driver.cpp
stats.o:  stats.cpp stats.h
      $(CC) $(CFLAGS) -c stats.cpp
clean:
      \rm -f *.o driver
```

New and improved!

```
# Programmer: Anilorac Leehrat Course: CSC112 A
# Assignment: 6                Date: 10/22/2007

driver:  driver.o stats.o
      g++ -o driver driver.o stats.o
driver.o:  driver.cpp stats.h
      g++ -g -pg -ansi -c driver.cpp
stats.o:  stats.cpp comp1.h
      g++ -g -pg -ansi -c stats.cpp
clean:
      \rm -f *.o driver
```

Old and inflexible. *It just smells bad...*

Your `makefile` for this assignment must include the following

- Comments indicating your name, course, assignment, and date
- `clean` target that removes the object files and the executable
- Variables for the compile command (`g++`) and the compiler flags (`-p -pg -ansi`).

4 Program Timing and Profiles

In a 1971 paper, Donald Knuth stated that '*less than 4 percent of a program generally accounts for more than half of its running time.*' As a result, if we could identify this 4 percent and optimize it, we should see an increase in performance. You will *time* and *profile* your lab 6 program to improve its speed.

4.1 Timing

Unix provides the `time` command to measure how long a program takes to execute. You just enter the command `time` followed by the executable name, then press return. For example, suppose you want to time your lab 6 program (assume the executable is called `lab6.exe`). You would enter the following command.

Terminal

```
> time ./lab6.exe invent3.dat update3.dat final3.dat
Reading invent3.dat
Database has 3449 ID numbers and requires 13796 bytes
------------------------------------------------------
Reading update3.dat
Updated database
Database has 1437 ID numbers and requires 5748 bytes
------------------------------------------------------
Output written to final3.dat
real    0m0.281s
user    0m0.202s
sys     0m0.077s
```

After the program finishes, a series of four numbers are displayed. The third number indicates the amount time required to complete the execution. In the example above, the program required 0.281 seconds.

4.2 Lab 6 Execution Time

Time your lab 6 program (dynamic array version) using `inventGrande.dat` and `updateGrande.dat` data files. Enter the amount of time your program required below.

	Time
Lab 6 execution time (using `inventGrande.dat` and `updateGrande.dat`)	

4.3 Profiling a Program

A program *profile* is a measurement indicating where a program spends most of its execution time. For example, some profiles list each function, the number of times it was called, and the fraction of execution time it consumes. Profiling is an effective tool for finding program *hot spots*, which are functions or sections of code that consumes most of the execution time.

The Unix utility (command) `gprof` will generate a program profile. However, before you run this utility you must do the following.

1. Compile your code with the `g++` compiler (Gnu C++ compiler)
2. Compile your code with the `-pg` flag (so your `makefile` compiler variable will be equal to `-g -pg`, and `-ansi`)

Update your `makefile` to reflect these changes (should have done this in the previous section).
Once the `makefile` is updated, compile the program and execute it using the `inventGrande.dat` and `updateGrande.dat` data files.

Terminal

```
> ./lab6 inventGrande.dat updateGrande.dat final.dat
Reading inventGrande.dat
Database has 1728 ID numbers and requires 13824 bytes
------------------------------------------------------
Reading updateGrande.dat
Updated database
Database has 860 ID numbers and requires 6880 bytes
------------------------------------------------------
Output written to final.dat
```

After the program completes, enter the command `gprof -b` followed by the executable name

Terminal

```
> gprof -b lab6.exe
Flat profile:

Each sample counts as 0.01 seconds.
%    cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
62.00    11.97     11.97   252617    47.37    47.37  search(int, int*, int)
20.13    15.85      3.89    39569    98.19    98.19  storeValue(int, int*&, int&)
17.48    19.23      3.37    38739    87.11    87.11  deleteValue(int, int*&, int&)
 0.52    19.33      0.10                             main
 0.00    19.33      0.00        1     0.00     0.00  global constructors keyed
 0.00    19.33      0.00        1     0.00     0.00  __static_initialization_and_de
```

The `prof` utility will then generate and display a table of the functions (ordered in decreasing execution time). This example lab 6 program contained `search`, `storeValue`, `deleteValue`, and `main` functions. The example `gprof` table above indicates the `search` function required the most time, since it is listed at the top of the table. Therefore, the `search` function is the program *hot spot* and should be rewritten to increase program performance.

4.4 Profiling Your Lab 6 Program

Compile your lab 6 code (via a your updated `makefile`) using the `g++` compiler and `-p -pg -ansi` compiler flags. Once compilation is complete, profile your executable using the `inventGrande.dat` and `updateGrande.dat` data files. Determine the *hot spot* in **your** code (which functions take the longest time). Write the name of the *hot spot* function and percent time in the space below.

	Function Name	**% Time**
Function that requires most execution time (using `inventGrande.dat` and `updateGrande.dat`)		

4.5 Optimizing Your Code

Using the profile information from the last section, optimize your program so it executes faster (this code and your updated `makefile` is what you will turn-in for lab 6). The *hot spot* is a good place to start. Consider faster algorithms for achieving the same goal. For example, your program's *hot spot* might be `search`. Are better searching algorithms available? **In the space below, list the functions you will rewrite and how you will increase performance.** If you think your program is optimal, indicate so. However, if I find something that can be optimized, then **you lose points**.

After you make your optimization update, profile and time your program (using `inventGrande.dat` and `updateGrande.dat` data files). You should see some improvement. Enter the time results below

	Time
Original lab 6	
Lab 6 (improved version)	

Enter the before and after (lab 6) optimization profile results for the function(s) you optimized below.

	%Time	
Optimized Function Name	**Lab 6 before**	**Lab 6 after**

5 Programming Points and What to Turn-in

You **must** adhere to all of the following points to receive credit for this assignment.

1. Turn-in this completed assignment (answer **all** questions), program, and `makefile` print-outs.
2. The lab will consist of 4 files
   - `main.cpp` Contains the main function.
   - `database.h` Contains the database function prototypes.
   - `database.cpp` Contains the database function definitions (**optimized** version).
   - `makefile` A makefile that compiles the program and generates the executable called `lab6` The `makefile` must be commented, have a `clean` target, and use variables as described in section 3.
3. **All arrays must be dynamically allocated. All arrays must be sized to store only the valid information. No wasted space! No memory leaks!**
4. Must adhere to documentation style and standards.