HW#2 Key Notes

(and a little bit of HW#1)

資料結構與程式設計

Data Structure and Programming

09.28.2016

(HW#1) This will crash…

```
class Row {
public:
   Row(size_t n = 0) { if (n) _data = new int[n]; }
   ~Row() { delete []_data; }
};
void Table::addRow() {
      Row r(3);
      for (size_t i = 0; i < 3; ++i) cin >> r[i];
      _rows.push_back(r);
}
```

- `_rows.push_back(r)` performs a copy of 'r' and push the copy to the vector. It copies (duplicates) the pointer variable "_data" (i.e. memory address of the data), but not the data memory (i.e. the int array) it points to. Therefore, when addRow() ends, 'r' is destructed and the memory is freed. Then the Row in vector becomes garbage.

What does HW#2 do?

1. Handle normal and special keys
   - Combo keys (e.g. UP_ARROW = 27 91 65)
2. Control cursor on the screen
   - Back space, delete, insert…
3. Record command line histories

1. Handle Special Keys

- Keyboard → ASCII code → cmdReader()
  - There should be an interface function to handle the special keys and tell cmdReader() what to do.
- However, different machines/terminals have different keyboard mappings
  - To maximize the code reuse and serve the most users, we should implement an adaptor to handle keyboard mapping so that we can share the same cmdReader()

1. Handle Special Keys --- enum/adaptor

- Keyboard → adaptor → cmdReader()
  - ASCII (0 ~ 255) vs. enum (-2^{31} ~ 2^{31})
- (Concept) "enum" to define named constants
  - For normal keys, just pass its ASCII code (e.g. 'A' = 65) to cmdReader()
  - For ARROW keys, add "ARROW_KEY_FLAG" (1 << 8 = 256) to key code to avoid collision with ASCII code
  - Similarly, for other MOD keys, add "MOD_KEY_FLAG" (1 << 9 = 512) to avoid collision with others

1. Handle Special Keys --- What you should do

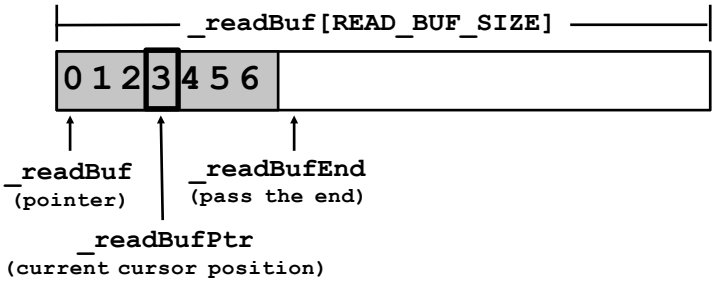
1. Customize "enum ParseChar" to fit your keyboard mapping, if necessary
   - Use the test program "testAsc" to test the key codes of your keyboard
2. Modify "ParseChar getChar(istream&)" which reads the key codes from keyboard or file, and converts them to "enum ParseChar" (as the "adaptor")

2. Control cursor on the screen

- Some key concepts
  1. What have been printed on the screen cannot be undone.
  2. However, you can use “back space” (i.e. char(8)) to move your cursor backwards
  3. New prints will overwrite the old prints
  4. Printing terminates when a char(0) is encountered.

2. Control cursor on the screen

- Data members



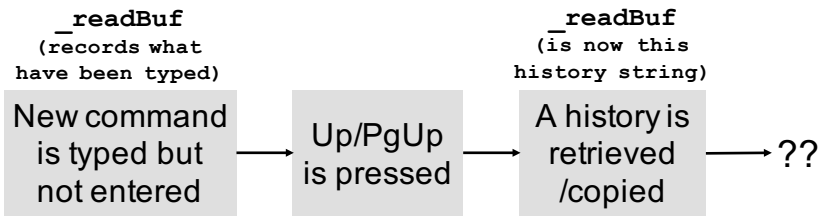
- Key principle
  - Make sure, at any time, what you store on “_readBuf”, “_readBufPtr”, and “_readBufEnd” are always consistent with what you see on the screen.

3. Record command histories

- Use “vector<string>” to record histories
- Data member “_historyIdx”, two meanings:
  1. When a new command is entered, it equals to “_history.size()” → the position to insert new history
  2. When up/down/pgUp/pgDown is applied, it points to the history to retrieve

3. Record command histories

- The remaining issue is:



- If “enter” is pressed, this retrieved/copied history (i.e. in “_readBuf”) will be added to _history
- However, if Down/PgDown is applied, previously typed command should be recovered.
- (Solution) When goes to history, record the previously typed string to _history, and use “bool _tempCmdStored” to identify this.