1

Order Book

Introduction

You have been tasked with creating an exchange where traders can buy and sell Sea Shells. The desired artifact is a program that reads incoming orders and emits trades.

Input (stdin)

The input is a stream of orders. Each order is on a separate line and identified by its line number, starting at one. An order message contains the following space-delimited fields:

- type (string): defines the type of the order, varies by order type
- side (string): either "buy" or "sell", unless ignored for specific order types
- value1 (integer): varies by order type
- value2 (decimal): varies by order type

Input is guaranteed to be valid. Note that:

- String values are unquoted.
- Decimals are formatted as fixed-point with two decimal places (e.g. 99.50).
- All numerical values fit in standard 64-bit primitive types (e.g. long and double).

There are four types of orders, as follows.

1. Market order

An order that is executed immediately at the best available prices. Market orders execute to the extent that opposing orders are available, and any unfilled amount is dropped. Fields:

- type: "market"
- value1: a positive number of Sea Shells to trade
- value2: ignored

Examples:

```
market buy 1000 0.00
market sell 50 0.00
```

2. Limit order

An order that can only execute at a specified price limit. If a limit order can not be filled in its entirety, any unfilled amount is booked for subsequent execution. Fields:

- type: "limit"
- value1: a positive number of Sea Shells to trade
- value2: a positive price limit at which they can be traded. Buy orders may execute at or below this limit. Sell orders may execute at or above it.

```
limit sell 5 57.12
```

3. Stop order

An order that only triggers when a subsequent trade occurs at a price that satisfies a given threshold. A sell-side stop order (aka stop-loss) triggers when the price is at or below the threshold. A buy-side stop order triggers when the price is at or above the threshold. If multiple stop orders are triggered at any given time, the oldest one executes first. Once triggered, a stop order executes as a market order. Triggered stop orders must execute before any subsequent incoming orders. Fields:

- type: “stop”
- value1: a positive number of Sea Shells to trade
- value2: a positive threshold price at which the order should trigger

Examples:

```
stop buy 20 55.00
stop sell 35 45.00
```

4. Cancel order

Cancels a previous order. May cancel a partially filled order, and cancels any unfilled portion in its entirety. Canceling a nonexistent, fully executed or previously canceled order is a no-op. Fields:

- type: “cancel”
- side: ignored
- value1: the number of the order to cancel
- value2: ignored

Example:

```
cancel none 3 0.00
```

Executing orders

An order is executed by matching it with one or more opposing orders, subject to the following rules:

- If multiple orders are available that can match a given order, they should first match by price limit. When matching a sell order, match against the highest buy order first; when matching a buy order, match against the lowest sell order first. If multiple matching orders have the same price limit, then match against the oldest among them first.
- Whenever an order is matched with an opposing order, a trade occurs. The amount of Sea Shells traded is the maximum accommodated by both. The price at which they trade

Execution stops when the order is filled in its entirety or no further matching orders are available. An order must stop executing before any other order (triggered or incoming) can execute.

Note that processing and execution of orders is done in an online manner: by the time order k+1 is processed, order k should have been processed and (possibly) executed.

Output (stdout)

Every time two orders are matched, emit a single output message with the following format:

```
match <taker> <maker> <volume> <price>
```

Fields:

- taker (integer): the number of order being executed
- maker (integer): the number of the opposing order that was matched
- volume (integer): the number of Sea Shells traded
- price (decimal): the price at which they were traded formatted as fixed-point with two decimal places

Example

Input:

```
limit buy 10 99.00
limit buy 15 100.00
limit buy 3 100.50
limit sell 5 100.00
limit buy 5 99.50
stop sell 3 99.49
cancel na 2 0.00
market sell 6 0.00
```

Output:

```
match 4 3 3 100.50
match 4 2 2 100.00
match 8 5 5 99.50
match 8 1 1 99.00
match 6 1 3 99.00
```

Important guidelines and notes

software staff. In addition to correctness, they will be evaluating for efficiency, adequacy, economy and clarity. In writing your code please follow best practices, such as proper use of abstraction and encapsulation, DRY, and annotating the code where appropriate. In particular, please use comments to document any assumptions, shortcuts or limitations.
- You may develop a solution in any development environment, however, your solution must eventually be submitted through this submission system. You may use any programming language supported by the submission system and your solution must comply with the system's limitations, such as keeping everything in a single file, naming of classes (if applicable), and so on.
- You may use standard libraries of your language of choice. The use of code from any other sources (open-source community, third-party libraries, etc) is not allowed.

YOUR ANSWER

We recommend you take a quick tour of our editor before you proceed. The timer will pause up to 90 seconds for the tour. Start tour

For help on how to read input and write output in C++, click here.

Unable to save draft.   Original code   C++

```
#include <map>
#include <set>
#include <list>
#include <cmath>
#include <ctime>
#include <deque>
#include <queue>
#include <stack>
#include <string>
#include <bitset>
#include <cstdio>
#include <limits>
#include <vector>
#include <climits>
#include <cstring>
```

```
#include <sstream>
#include <iostream>
#include <algorithm>
#include <unordered_map>

using namespace std;

struct Order {
    long id;
    string type;
    string side;
    long value1;
    double value2;
    bool active;
    Order(const long id, const string & type, const string & sic
const long value1, const double value2) {
        this->id = id;
        this->type = type;
        this->side = side;
        this->value1 = value1;
        this->value2 = value2;
        this->active = true;
    }
};

class Trade {
public:
        Trade() {
            count = 0;
            table.clear();
            table.push_back(NULL);
        }
        ~Trade() {
            count = 0;
            for (auto & i : table) {
                if (i) {
                    delete i;
                    i = NULL;
                }
            }
        }
        Order * push(const string & line) {
            ++count;
            size_t n = line.size(), i = 0;
            while (i < n and line[i] != ' ') {
```

```
            size_t j = ++i;
            while (i < n and line[i] != ' ') {
                ++i;
            }
            string side = line.substr(j, i - j);
            j = ++i;
            while (i < n and line[i] != ' ') {
                ++i;
            }
            long value1 = stol(line.substr(j, i - j));
            j = ++i;
            while (i < n and line[i] != ' ') {
                ++i;
            }
            double value2 = stod(line.substr(j, i - j));
            Order * order = new Order(count, type, side, value1, value2);
            table.push_back(order);
            return order;
        }
        string execute(const string & line) {
            Order * order = push(line);
            if (order->type == "market" or order->type == "limit
{
                return toMarketAndtoLimit(order);
            }
            if (order->type == "stop") {
                return toStop(order);
            }
            if (order->type == "cancel") {
                return toCancel(order);
            }
            return "";
        }
private:
        long count;
        vector<Order*> table;
        bool condition1(const Order * order, const double value2
const double price) {
            if (order->type == "market") {
                return order->side == "buy" ? value2 < price : value2 > price;
            }
            if (order->side == "buy") {
                return value2 <= order->value2 and value2 < pric
        }
            return false;
        }
        bool condition2(const string & side, const double value2
const double price) {
            return side == "buy" ? value2 <= price : value2 >=
price;
        }
        string double2string(const double val, const int n = 2)
            ostringstream result;
            result.precision(n);
            result << fixed << val;
            return result.str();
        }
        string toMarketAndtoLimit(Order * order) {
            if (!order) {
                return "";
            }
            string result;
            double bound = order->side == "buy" ?
numeric_limits<double>::max() : numeric_limits<double>::min();
            string orderSide = order->side, orderOpposite =
orderSide == "buy" ? "sell" : "buy";
            list<Order*> l;
            while (order->value1 > 0) {
                long id = -1;
                double price = bound;
                for (const auto & i : table) {
                    if (!i or !i->active) {
                        continue;
                    }
                    if (i->type == "limit" and i->side ==
orderOpposite and i->value1 > 0 and condition1(order, i->value2,
price)) {
                        id = i->id;
                        price = i->value2;
                    }
                }
                if (id == -1) {
                    break;
                }
                long volume = min(order->value1, table[id]-
>value1);
                result += "match " + to_string(order->id) + " "
to_string(id) + " " + to_string(volume) + " " +
                for (const auto & i : table) {
                    if (!i or !i->active) {
                        continue;
                    }
                    if (i->type == "stop" and i->side == orderSi
and i->value1 > 0 and condition2(orderSide, i->value2, price)) {
                        l.push_back(i);
                        continue;
                    }
                    if (i->type == "stop" and i->side ==
orderOpposite and i->value1 > 0 and condition2(orderOpposite, i-
>value2, price)) {
                        l.push_back(i);
                        continue;
                    }
                }
            }
            if (order->active and order->type == "market") {
                order->active = false;
            }
            for (auto & i : l) {
                if (!i or !i->active) {
                    continue;
                }
                i->type = "market";
                result += toMarketAndtoLimit(i);
                i->active = false;
            }
            return result;
        }
        string toStop(Order * order) {
            return "";
        }
        string toCancel(Order * order) {
            if (!order or order->id > count) {
                return "";
            }
            table[order->id]->active = false;
            if (order->value1 >= 1 and order->value1 <= count) {
                table[order->value1]->active = false;
            }
            return "";
        }
};
```

```
    Trade trade;
    string line;
    while (getline(cin, line)) {
        cout << trade.execute(line);
    }
    return 0;
}
```

Line: 1 Col: 1

☐ **Test against custom input**

Run Code

Submit code & Continue

(You can submit any number of times)

Download sample test cases *The input/output files have Unix line endings. Do not use Notepad to edit them on windows.*

Status: Error when trying to compile and test: Unauthorized request. Looks like you're not logged in (or logged in from elsewhere).