Algorithms in C++: Assignment 7

1. Objective

Your goal is to write a program that uses dynamic programming to solve the 0-1 knapsack problem.

2. Problem

You are a thief! However, you are not just any thief. You want to maximize your profit by choosing the best items that can fit into your knapsack. You are also not greedy! You have studied computer science and realize that you must use dynamic programming to get the absolute best solution. Your silent accomplice `test.sh` will give you a text file containing the inventory of items in the store. It will look as follows but might contain significantly more lines.

```
book,3,35
old laptop,2,40
cheesecake,5,50
boots,4,10
headphones,1,30
```

Each line contains three fields, separated by commas. The first is the item's description, the second is the item's weight in pounds, and the third is the item's value in dollars.

Having been programming in C++ for a whole semester now, you realize it's going to be much better for you to write a program `knapsack.cpp` to parse and manipulate this data than to generate a table by hand. However, you know that users (including yourself, especially if you're in a hurry to accomplish your dirty deed) make mistakes. So, your program must do some error checking. Here are examples of what to expect:

1) No arguments or too many arguments.

```
Usage: ./knapsack <capacity> <filename>
```

2) Valid capacity, but missing file, as in: *10 notfound.txt*

```
Error: Cannot open file 'notfound.txt'.
```

3) Invalid capacity, as in 'xxx' or some negative number.

```
Error: Bad value 'xxx' for capacity.
```

4) You also need to check the contents of the file, as your accomplice might have made a mistake when taking inventory of the store. Some possible errors are as follows:

```
Error: Line number 1 does not contain 3 fields.
Error: Invalid weight 'x' on line number 1.
Error: Invalid value '-3' on line number 2.
```

Once you are confident that the input is clean, you should move forward with the dynamic programming algorithm and produce the desired output. Assuming that the 5 items above are in a file called input.txt and that your knapsack holds 5 pounds, you will run your program and see the following output:

```
$ ./knapsack 5 input.txt
Candidate items to place in knapsack:
   Item 1: book (3 pounds, $35)
   Item 2: old laptop (2 pounds, $40)
   Item 3: cheesecake (5 pounds, $50)
   Item 4: boots (4 pounds, $10)
   Item 5: headphones (1 pound, $30)
Capacity: 5 pounds
Items to place in knapsack:
   Item 1: book (3 pounds, $35)
   Item 2: old laptop (2 pounds, $40)
Total weight: 5 pounds
Total value : $75
```

Notice that several combinations of items weigh 5 pounds, including the cheesecake by itself and the boots-headphones combo. However, the book and old laptop clearly have more value, and you'll want to steal them. Well, maybe not – it's actually an HP laptop and C++ programming book. ☹

3. Advice

You are allowed to use the following struct to make your program more readable:

```
struct Item {
    unsigned int item_number, weight, value;
    string description;

    explicit Item(const unsigned int item_number,
                  const unsigned int weight,
                  const unsigned int value,
                  const string &description) :
                          item_number(item_number),
                          weight(weight),
                          value(value),
                          description(description) { }

    friend std::ostream& operator<<(std::ostream& os, const Item &item) {
        os << "Item " << item.item_number << ": " << item.description
           << " (" << item.weight
           << (item.weight == 1 ? " pound" : " pounds") << ", $"
           << item.value << ")";
        os.flush();
        return os;
    }
};
```