1. Lösungsidee:

Man kann die Angaben zur Reihenfolge leicht als Bedingungen der Art A => B sehen erkennen. Man kann also für jedes Kleidungsstück (allgemeiner kann man es als Item bezeichnen) eine Menge von Items finden, die vorausgesetzt werden. Wenn die Menge leer ist, so kann das Item jederzeit angezogen werden, da es ja keine anderen vorausgesetzt sind.
Um nun eine mögliche Ankleidungsreihenfolge zu bestimmen, kann man folgenden, rekursiven Algorithmus benutzen:

Eingaben: Menge A aller schon angezogenen Items, Menge V aller noch verbliebenen Items
Am Anfang ist A leer und V enthält alle Items.

1) Finde ein Item v der Menge V, für dessen Menge R der vorausgesetzten Items gilt:
 R \ A = {}, dann kann das Item angezogen werden. Wird kein einziges gefunden, springe zu Schritt 3.
2) Sei A'= A U {v} (U steht für 'vereinigt mit') und V' = V \ {v}, dann soll der Algorithmus mit A' und V' (statt A und V) rekursiv ausgeführt werden.
3) Entweder ist die Menge V leer und alle Items wurden angezogen und unser Algorithmus war erfolgreich, oder V ist nicht leer und es gibt Items, die nicht angezogen werden können, und es gibt keine Lösung für das Problem.

Die Reihenfolge ergibt sich aus der Reihenfolge mit der Items in A eingefügt werden. Um alle Lösungen zu finden, kann man Teillisten verwenden, die bei jedem rekursiven Aufruf weitergegeben werden. Wenn V dann leer ist, wird diese Liste dann zur Menge der Lösungen hinzugefügt und der Algorithmus macht mit den vorherigen A und V weiter, sucht also das nächste Item v, für welches die Bedingung aus 1) gilt, etc etc.
Man kann sich das ganze als rekursives Durchqueren eines Baumes vorstellen, wobei der Pfad gespeichert wird und die Lösungsreihenfolge darstellt.

2. Programm-Dokumentation:

Zuerst einmal möchte ich das Eingabeformat beschreiben. Eine einfache Eingabedatei kann z.B so aussehen:

```
[Items]
A
B
C

[Rules]
A B
A C

#EOF
```

Nach [items] folgt eine Auflistung aller gültigen Items und nach [rules] folgen die Regeln, wobei A B bedeutet A vor B bzw A => B.

Der Dateiname der Eingabedatei wird als erste Befehlszeilenparameter übergeben. Als zweiter Befehlszeilenparameter kann auch noch die Option single angegeben werden, was das Progamm veranlasst nur eine einzige Lösung auszugeben.
Dabei wird der gleiche Algorithmus verwendet wie bei der Bestimmung aller Lösungen - es werden also auch alle Lösungen bestimmt. Der Grund dafür liegt

darin, da die Bestimmung aller Lösungen nur einiger kurzer Ergänzungen des Grundalgorithmus bedarf und ich nicht den Quellcode durch zwei gleiche Algorithmen unnötig aufblähen wollte.

Zur Entwicklung und zur Umsetzung der Lösungsidee:
Ich verwende extensiv STL, benutze typedefs und andere Konstrukte, die den Quellcode meiner Meinung nach nicht besonders schön aussehen lassen, aber auch nicht zu lange werden lassen und ihn dafür nur auf das Wichtigste beschränken.
Es gibt 2 wichtige Methoden: Item::CanBeInserted(const ConstPointerSet &usedItems) und DressingSolutions::Solve_r(Item::ConstPointerSet &usedItems, ItemOrder &currentOrder).
Die Datentypen sind der Lösungsidee entsprechend nachmodelliert:
Item enthält die Eigenschaften eines Items/Kleidungsstückes: die Menge aller anderen vorausgesetzten Items und den Namen des Items. ItemOrder speichert eine Reihenfolge, und DressingSolutions speichert alle Lösungsreihenfolgen.

Nun zu den beiden wichtigen Methoden:
Item::CanBeInserted(const ConstPointerSet &usedItems) überprüft nichts weiter als die Bedingung in Schritt 1) des Algorithmuses: Die Methode liefert "wahr" zurück, falls die Menge der vorausgesetzten Items des Item R *ohne* die Menge der bereits angezogenen Items A gleich der leeren Menge ist, also falls alle vorausgesetzten Items angezogen sind.
Um dies zu prüfen, wird folgender Algorithmus verwendet:
A und R sind aufsteigend nach einem eindeutigen Kriterium sortiert (z.B. beide Beziehen sich auf die gleiche Gesamtmenge von Items im Speicher durch Zeiger und die Adresse der Datenstrukturen wäre dann ein solches Kriterium).
1. Wiederhole bis entweder A oder R vom Anfang an vollständig durchlaufen sind:
   1) Seien a und r, die derzeitig ausgewählten Elemente der Mengen A bzw R.
   2) Falls r < a, so gibt es ein Element aus R, dass nicht in A vorkommt und die Prüfung ist beendet, da R \ A nicht leer ist und CanBeInserted gibt gleich "falsch" zurück.
   3) Falls a < r, so wird zum nächsten Element von A iteriert und wieder von 1. angefangen.
   4) Falls a = r, dann wird zum nächsten Element von A und R weiter iteriert und es wird wieder bei 1. angefangen.
2. Falls vollständig durch R iteriert wurde, dann war die Prüfung erfolgreich, da alle Elemente von R durchlaufen wurden und auch jedes Element in A vorkam (,da sonst die Prüfung vorzeitig beendet worden wäre).
3. Falls nicht vollständig durch R iteriert wurde, so muss die Schleife beendet worden sein, weil vollständig durch A iteriert wurde, und R enthält damit Elemente die nicht in A sein können, da sie größer als das größte Element in A sind und folglich in der aufsteigend sortierten Menge A nicht enthalten sind, und die Prüfung ist damit negativ ausgefallen.

DressingSolutions::Solve_r(Item::ConstPointerSet &usedItems, ItemOrder &currentOrder) ist eigentlich eine recht genaue Implementierung des Algorithmus, es gibt jedoch zwei Änderungen, die aber nicht das äußere Verhalten an sich betreffen:
1. Es wird am Anfang der Funktion überprüft, ob schon alle Kleidungsstücke angezogen wurden und
2. die Menge V der verbliebenen Items wird implizit durch die Menge aller Items ohne die Menge der angezogenen Items dargestellt, deshalb wird durch die Menge aller Items iteriert statt durch V und neben der Bedingung des Algorithmuses wird auch noch überprüft, ob ein Item nicht schon Element der angezogenen Items ist. Dadurch wird sichergestellt, dass auch wirklich nur Elemente, die die Menge V ausmachen würden, getestet werden.

3. Programm-Ablaufprotokoll

```
[...]>"Aufgabe 2.exe" test1.txt
90 Lösungen gefunden

Lösung #1: Bluse->Mütze->Strümpfe->Hose->Pullover->Schal->Jacke->Handschuhe-
>Schuhe
Lösung #2: Bluse->Mütze->Strümpfe->Hose->Pullover->Schal->Jacke->Schuhe-
>Handschuhe
Lösung #3: Bluse->Mütze->Strümpfe->Hose->Pullover->Schal->Schuhe->Jacke-
>Handschuhe
Lösung #4: Bluse->Mütze->Strümpfe->Hose->Pullover->Schuhe->Schal->Jacke-
>Handschuhe
Lösung #5: Bluse->Mütze->Strümpfe->Hose->Schuhe->Pullover->Schal->Jacke-
>Handschuhe
Lösung #6: Bluse->Strümpfe->Hose->Mütze->Pullover->Schal->Jacke->Handschuhe-
>Schuhe
Lösung #7: Bluse->Strümpfe->Hose->Mütze->Pullover->Schal->Jacke->Schuhe-
>Handschuhe
Lösung #8: Bluse->Strümpfe->Hose->Mütze->Pullover->Schal->Schuhe->Jacke-
>Handschuhe
Lösung #9: Bluse->Strümpfe->Hose->Mütze->Pullover->Schuhe->Schal->Jacke-
>Handschuhe
Lösung #10: Bluse->Strümpfe->Hose->Mütze->Schuhe->Pullover->Schal->Jacke-
>Handschuhe
Lösung #11: Bluse->Strümpfe->Hose->Pullover->Mütze->Schal->Jacke->Handschuhe-
>Schuhe
Lösung #12: Bluse->Strümpfe->Hose->Pullover->Mütze->Schal->Jacke->Schuhe-
>Handschuhe
.
.
.
Lösung #83: Strümpfe->Bluse->Mütze->Hose->Pullover->Schal->Schuhe->Jacke-
>Handschuhe
Lösung #84: Strümpfe->Bluse->Mütze->Hose->Pullover->Schuhe->Schal->Jacke-
>Handschuhe
Lösung #85: Strümpfe->Bluse->Mütze->Hose->Schuhe->Pullover->Schal->Jacke-
>Handschuhe
Lösung #86: Strümpfe->Mütze->Bluse->Hose->Pullover->Schal->Jacke->Handschuhe-
>Schuhe
Lösung #87: Strümpfe->Mütze->Bluse->Hose->Pullover->Schal->Jacke->Schuhe-
>Handschuhe
Lösung #88: Strümpfe->Mütze->Bluse->Hose->Pullover->Schal->Schuhe->Jacke-
>Handschuhe
Lösung #89: Strümpfe->Mütze->Bluse->Hose->Pullover->Schuhe->Schal->Jacke-
>Handschuhe
Lösung #90: Strümpfe->Mütze->Bluse->Hose->Schuhe->Pullover->Schal->Jacke-
>Handschuhe

[...]>"Aufgabe 2.exe" test3.txt
24 Lösungen gefunden

Lösung #1: Halskette->Socken->Boxershort->Tshirt->Hose->Pullover->Jacke->Schuhe
Lösung #2: Halskette->Socken->Boxershort->Tshirt->Pullover->Hose->Jacke->Schuhe
Lösung #3: Halskette->Socken->Boxershort->Tshirt->Pullover->Jacke->Hose->Schuhe
Lösung #4: Socken->Boxershort->Halskette->Tshirt->Hose->Pullover->Jacke->Schuhe
Lösung #5: Socken->Boxershort->Halskette->Tshirt->Pullover->Hose->Jacke->Schuhe
Lösung #6: Socken->Boxershort->Halskette->Tshirt->Pullover->Jacke->Hose->Schuhe
Lösung #7: Socken->Boxershort->Tshirt->Halskette->Hose->Pullover->Jacke->Schuhe
Lösung #8: Socken->Boxershort->Tshirt->Halskette->Pullover->Hose->Jacke->Schuhe
Lösung #9: Socken->Boxershort->Tshirt->Halskette->Pullover->Jacke->Hose->Schuhe
Lösung #10: Socken->Boxershort->Tshirt->Hose->Halskette->Pullover->Jacke->Schuhe
Lösung #11: Socken->Boxershort->Tshirt->Hose->Pullover->Halskette->Jacke->Schuhe
```

```
Lösung #12: Socken->Boxershort->Tshirt->Hose->Pullover->Jacke->Halskette->Schuhe
Lösung #13: Socken->Boxershort->Tshirt->Hose->Pullover->Jacke->Schuhe->Halskette
Lösung #14: Socken->Boxershort->Tshirt->Pullover->Halskette->Hose->Jacke->Schuhe
Lösung #15: Socken->Boxershort->Tshirt->Pullover->Halskette->Jacke->Hose->Schuhe
Lösung #16: Socken->Boxershort->Tshirt->Pullover->Hose->Halskette->Jacke->Schuhe
Lösung #17: Socken->Boxershort->Tshirt->Pullover->Hose->Jacke->Halskette->Schuhe
Lösung #18: Socken->Boxershort->Tshirt->Pullover->Hose->Jacke->Schuhe->Halskette
Lösung #19: Socken->Boxershort->Tshirt->Pullover->Jacke->Halskette->Hose->Schuhe
Lösung #20: Socken->Boxershort->Tshirt->Pullover->Jacke->Hose->Halskette->Schuhe
Lösung #21: Socken->Boxershort->Tshirt->Pullover->Jacke->Hose->Schuhe->Halskette
Lösung #22: Socken->Halskette->Boxershort->Tshirt->Hose->Pullover->Jacke->Schuhe
Lösung #23: Socken->Halskette->Boxershort->Tshirt->Pullover->Hose->Jacke->Schuhe
Lösung #24: Socken->Halskette->Boxershort->Tshirt->Pullover->Jacke->Hose->Schuhe

[...]>"Aufgabe 2.exe" test3.txt
7 Lösungen gefunden

Lösung #1: Socken->Unterhose->Unterhemd->Strumpfhose->Schneeanzug->Mütze-
>Schneebrille->Wintersocken->Schuhe->Schneeeisen
Lösung #2: Socken->Unterhose->Unterhemd->Strumpfhose->Schneeanzug->Mütze-
>Wintersocken->Schneebrille->Schuhe->Schneeeisen
Lösung #3: Socken->Unterhose->Unterhemd->Strumpfhose->Schneeanzug->Wintersocken-
>Mütze->Schneebrille->Schuhe->Schneeeisen
Lösung #4: Socken->Unterhose->Unterhemd->Strumpfhose->Wintersocken->Schneeanzug-
>Mütze->Schneebrille->Schuhe->Schneeeisen
Lösung #5: Socken->Unterhose->Unterhemd->Wintersocken->Strumpfhose->Schneeanzug-
>Mütze->Schneebrille->Schuhe->Schneeeisen
Lösung #6: Socken->Unterhose->Wintersocken->Unterhemd->Strumpfhose->Schneeanzug-
>Mütze->Schneebrille->Schuhe->Schneeeisen
Lösung #7: Socken->Wintersocken->Unterhose->Unterhemd->Strumpfhose->Schneeanzug-
>Mütze->Schneebrille->Schuhe->Schneeeisen
```

4. Quellcode

```
#include <assert.h>

#include <string>
#include <fstream>
#include <iostream>

#include <algorithm>
#include <vector>
#include <set>
#include <map>

#include <string.h>

struct Item {
	typedef std::set< const Item * > ConstPointerSet;
	typedef std::vector< const Item * > ConstPointerVector;
	typedef std::set< Item > Set;

	std::string name;
	ConstPointerSet requiredItems;

	Item( const std::string &name ) : name( name ) {
	}

	bool CanBeInserted( const ConstPointerSet &usedItems ) const {
		ConstPointerSet::const_iterator usedIter = usedItems.begin();
```

```
		ConstPointerSet::const_iterator requiredIter = requiredItems.begin();
		while( usedIter != usedItems.end() && requiredIter != requiredItems.end() ) {
			if( *usedIter < *requiredIter ) {
				usedIter++;
			} else if( *requiredIter < *usedIter ) {
				// this means that at least one item is in requiredItems that isn't in usedItems
				// => the difference can't be empty
				return false;
			} else if( *usedIter == *requiredIter ) {
				usedIter++;
				requiredIter++;
			}
		}
		// if the iteration through requiredItems has finished (without returning false), all items are in
usedItems, too
		if( requiredIter == requiredItems.end() ) {
			return true;
		// else only the iteration through usedItems has finished, but the iteration through requiredItems hasn't
and consequently there are still items left over
		} else {
			return false;
		}
	}
};

bool operator <( const Item &a, const Item &b ) {
	return a.name < b.name;
}

/*
Definition:
~~~~~~~~~~

[Items]
A
B
C
...
[Rules]
A B
A C
..

*/
void ParseDefinitionFile( const char *filename, Item::Set &itemSet ) {

	std::ifstream file( filename );
	std::string token;

	file >> token;
	if( 0 != stricmp( token.c_str(), "[items]" ) ) {
		return;
	}
	while( file >> token, 0 != stricmp( token.c_str(), "[rules]" ) && !file.eof() ) {
		itemSet.insert( Item( token ) );
	}
	if( file.eof() ) {
		return;
	}
	bool parseRequiredItem = true;
```

```
	const Item *requiredItem;
	do {
		file >> token;
		Item *item = 0;
		for( Item::Set::iterator iter = itemSet.begin() ; iter != itemSet.end() ; iter++ ) {
			if( stricmp( iter->name.c_str(), token.c_str () ) == 0 ) {
				item = &*iter;
				break;
			}
		}
		if( item == 0 ) {
			std::cout << "Bad token '" << token << "'" << std::endl;
			break;
		}
		if( parseRequiredItem ) {
			requiredItem = item;
		} else {
			item->requiredItems.insert( requiredItem );
		}

		parseRequiredItem = !parseRequiredItem;
	} while( !file.eof() );
}

struct ItemOrder {
	typedef std::vector< ItemOrder > Vector;

	Item::ConstPointerVector order;
};

struct DressingSolutions {
	const Item::Set &itemSet;
	ItemOrder::Vector solutionList;

	DressingSolutions( const Item::Set &itemSet ) : itemSet( itemSet ) {
	}

protected:
	void Solve_r( Item::ConstPointerSet &usedItems, ItemOrder &currentOrder ) {
		if( itemSet.size() == usedItems.size() ) {
			solutionList.push_back( currentOrder );
			return;
		}
		for( Item::Set::const_iterator iter = itemSet.begin() ; iter != itemSet.end() ; iter++ ) {
			if( iter->CanBeInserted( usedItems ) && usedItems.find( &*iter ) == usedItems.end() ) {
				Item::ConstPointerSet newUsedItems = usedItems;
				newUsedItems.insert( &*iter );
				ItemOrder newOrder = currentOrder;
				newOrder.order.push_back( &*iter );
				Solve_r( newUsedItems, newOrder );
			}
		}
	}

public:
	void Solve() {
		Item::ConstPointerSet usedItems;
		ItemOrder order;
		Solve_r( usedItems, order );
	}
```

```
protected:
	void OutputSolution( const unsigned solutionIndex ) const {
		//std::cout << "Lösung #" << (solutionIndex + 1) << std::endl;
		std::cout << "Lösung #" << (solutionIndex + 1) << ": ";
		const ItemOrder &solution = solutionList[ solutionIndex ];
		for( unsigned itemIndex = 0 ; itemIndex < solution.order.size() ; itemIndex++ ) {
			const Item &item = *solution.order[ itemIndex ];
			//std::cout << (itemIndex + 1) << ". " << item.name << std::endl;
			if( itemIndex != 0 ) {
				std::cout << "->";
			}
			std::cout << item.name;
		}
		std::cout << std::endl;
	}

public:
	void OutputFirstSolution() const {
		if( !solutionList.empty() ) {
			OutputSolution( 0 );
		}
	}

	void OutputSolutions() const {
		std::cout << solutionList.size() << " Lösungen gefunden" << std::endl << std::endl;
		for( unsigned solutionIndex = 0 ; solutionIndex < solutionList.size() ; solutionIndex++ ) {
			OutputSolution( solutionIndex );
		}
	}
};

int main( int argc, char *argv[] ) {
	if( argc < 2 ) {
		return -1;
	}
	bool allSolutions = true;
	if( argc == 3 ) {
		if( stricmp( argv[ 2 ], "single" ) == 0 ) {
			allSolutions = false;
		}
	}
	Item::Set itemSet;
	ParseDefinitionFile( argv[1], itemSet );

	DressingSolutions dressing( itemSet );
	dressing.Solve();
	if( allSolutions ) {
		dressing.OutputSolutions();
	} else {
		dressing.OutputFirstSolution();
	}
	return 0;
}
```