The task is to develop a generic class `CAccess` that can find connected places.

We assume a map with places - nodes (e.g. cities, computers, ...) and connections (edges) that connect the places (e.g. roads, railroads, network links, ...). The basic variant assumes there are exactly two places connected by one connection (e.g. a train that only stops at the terminals). The bonus test adds an option to create a connection that connects more than two places (e.g. a train with many stops). The class shall store such map in its member variables. Since the map is general, the data types describing places and connections are not known. Thus the class will be implemented as a generic class where the types of places and connections are generic parameters.

The class will be able to compute the list of places that are accessible from a given place using the connection from the (previously stored) map. The computation will take two optional parameters - the first parameter can be used to limit the maximal number of connections used (e.g. if the limit is 5, we can use up to 5 different connections, thus we can change the connection at most 4 times). Second, the computation will take a filter, the filter will be used to filter out undesired connections in the computation. For instance, we may use the filter to consider only trains faster than a given minimum speed.

The class will provide a method to insert connections and a method to search the connected places. The exact interface is:

default constructor
: prepares an empty `CAccess` instance.

method Add (e, u1, u2)
: this method adds a new connection connecting places `u1` and `u2`. The parameter `e` is the description of the connection, e.g. the speed of the line, type of road, ... The edge is assume two-way edge.

method Find (u[, max, [,filter]])

: this method will search the places that are accessible from place `u`. If called with one parameter, the method will search the accessible places without any restrictions. If there are two parameters, then the second parameter `max` limits the maximum number of connections used in the computation (e.g. `max = 1` searches only places directly connected with `u`, `max = 2` searches for places that are accessible by at most two connections, i.e. the connection is changed at most once, `max = 3` searches for places that are accessible by at most three connections, i.e. the connection is changed at most twice,...). Finally, the third parameter is a filter of the connections. The filter is provided in the form of a pointer-to-function, a functor, or a lambda. If used, the filter shall be called when a connection is about to be used to test whether to actually use it (the filter returns `true`) or not (the filter returns `false`).

: The result of the call is a map of key-value pairs. The keys are the places accessible from `u`, the associated values are integers indicating the minimum number of connections needed to get to that place from `u`. The method shall throw `invalid_argument` exception (the exception is declared in the standard library) if the place `u` is unknown (has never been added in any previous call to `Add`).

method `Add(e, u1, u2, u3, ...)`

> this method is tested in the bonus test only. The method is an extension of the mandatory method `Add` above. A call to this method adds a connection that connects more places - `u1`, `u2`, `u3`, ... In other words, the method is a shortcut to add a direct connection between all pairs of places in the arguments.

Generic parameter `_T` is the data type that represents a place. Your implementation may use the following operations with _T:

- copy constructor,
- operator =,
- relational operators (==, !=, <, <=, > >=),
- destructor,
- output operator <<.
- Further operations may exist, however, they are not guaranteed.

Generic parameter `_T` is the data type that represents a connection. Your implementation may use the following operations with _E:

- copy constructor,
- operator =,
- destructor.
- Further operations may exist, however, they are not guaranteed.

Submit a source file with the implementation of the `CAccess` class template. Use the attached source code as a basis for your implementation. Please preserve the conditional compile directives - if present, the source may be locally tested and submitted to Progtest without manual modifications.

The assessment is divided into mandatory, optional tests, and bonus tests. The mandatory tests use only small maps with small number of places. The optional test tests maps with high number of places, thus the implementation requires some efficient STL data structures to finish in the time limit. A significant penalty will apply if the optional test is not passed.

Use BFS (breadth first search) algorithm to search the map. Further inspiration: PA1, proseminar #11, labyrinth problem.

C++ 11 variadic templates are needed to pass the bonus test. If you decide to skip the bonus test, please leave undefined preprocessor directive `MULTIPLE_STOPS` (if the directive is `#defined` and the implementation of the extended `Add` method is missing, the program will not compile). On the other hand, if you implement the extended `Add` method, enable the directive to actually do the test (if not enabled, the bonus test is skipped and the bonus is not awarded).

The example source code may be found in the attached archive.