Script for the vhb lecture

Programming in C++
Part 2

Prof. Dr. Herbert Fischer
Deggendorf Institute of Technology

Table of Contents

5 Linked lists

5.1 Singly linked lists

A major problem with arrays in C++ is that once an array has been created - regardless of whether you have allocated the memory statically or dynamically - it can no longer change its size. However, programs would be highly inflexible if there was not a very elegant solution to this problem. The idea is not to create a specific, predefined set of objects, but to link individual objects using one or more pointers. This creates larger, connected data structures of objects distributed freely. Dynamic data structures are only created at program runtime.

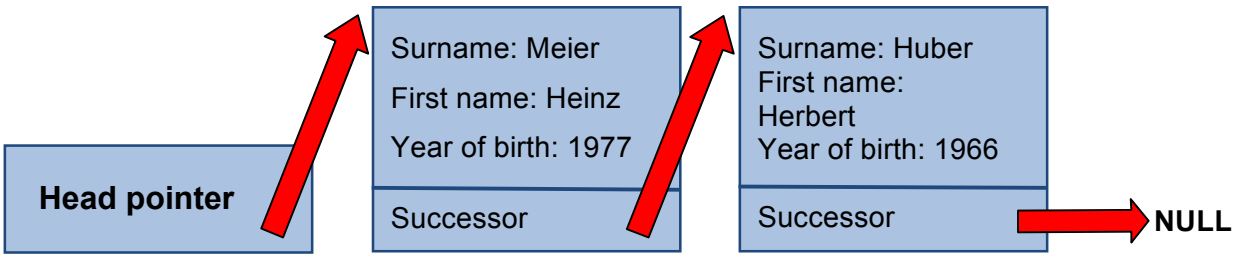
One of the best known or even the best-known dynamic data structure is the singly linked list. It consists of elements that are connected to each other via pointers (usually called successors).

We can build up a list of any number of people by linking individual people together using a concatenation pointer. More precisely, one person uses the concatenation pointer to point to another person.

The elements of a linked list are called nodes. Each node of a list **must** consist of **two components**:

- The actual data content, that is, data members in the example of the *Person* class
- And a "successor" pointer to another object of type *Person*

The "successor" pointer points to the next node of the same type - or to NULL, which marks the end of the list. Thus, each node in this list refers to a subsequent node. Using this concatenation technique, the individual independent objects are combined to a singly linked list.



All we need is a pointer that points to the first node and thus indicates the beginning of the list.
The entire list is represented by this pointer to the first node, which is called header pointer. In general, any number of elements can be concatenated in a list – at the extreme, the list can occupy the entire memory available in your computer.

How do we now have to extend our *Person* class so that a list of persons can be built? We need a new data member of type *Pointer* to the *Person* class to be defined. In C++ this is possible for pointers, although the *Person* class is only just being defined.

5.1.1 Definition of *Person* class as element of a list

```
#include <cstdlib>
#include <iostream>
#include <ctime>
using namespace std;

class Person
{
   friend class ListOfPeople; // friend declaration so that the other class has access to the private elements
   // Data members
   private:
      string surname;
      string firstname;
      int yearOfBirth;
      Person* Successor; // Concatenation pointer
   // Member functions
   public:
      Person (string n, string f, int y): surname(n), firstname(f), yearOfBirth(y) {}

      string getSurname () const { return surname; }
      string getFirstname () const { return firstname; }
      int getYearOfBirth() const { return yearOfBirth; }

      void calculateAge () {
         // Determine current year
         time_t timestamp;
         tm *date;
         timestamp = time(0);
         date = localtime(&timestamp);
         int actYear = date->tm_year+1900;

         cout << "Age: " << actYear- yearOfBirth<< " years old" << endl;
         }
      Person* next () { return Successor; } // Returns the successor of my current object

      void displayData() {
          cout << "Surname: " << surname << ", First name: " << firstname;
          cout  << ", Year of Birth: " << yearOfBirth << endl;
      }
};
```

This enables an object of type *Person* to reference to another object of the same type. This relationship "to itself" is represented in the class diagram by a relationship to itself → reflexive association. This is to illustrate that the *Person* class has the property to refer to objects of the same type.

Through this linking, all persons form a coherent structure - a container for persons. We can consider this container as an independent object and realize a class - let's call it "ListOfPeople". To give this class access to the concatenation pointer of *Person*, we have declared it as "friend" in the *Person* class.

What are the data members of an object of the type "ListOfPeople" and which data members should this list provide? We already know that a single pointer - the head pointer - to the first element can represent the entire list.
This will be the only data member of our list class and the most important member functions are:

- Create list – Constructor for empty list
- Add a new person to the beginning of the list
- Remove a new person from the beginning of the list
- Check if list is empty

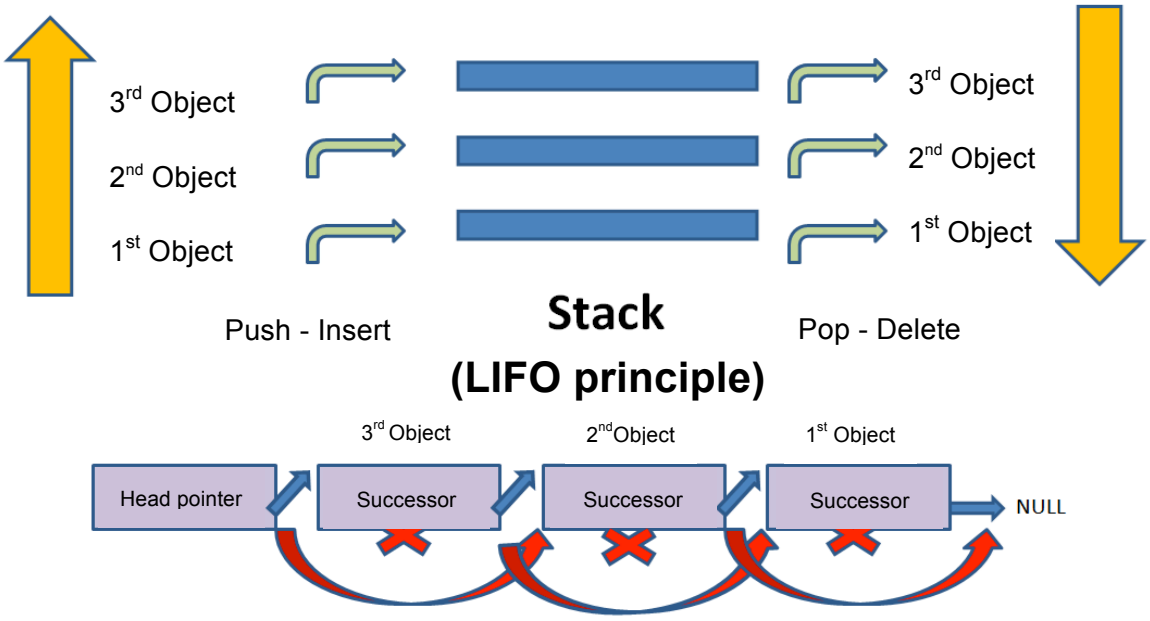
5.1.2 Definition of the *ListOfPeople* class

```
class ListOfPeople
{
   private:
          Person* Headpointer;
   public:
          ListOfPeople():Headpointer(0) {};
          // Insert head pointer at the beginning of the list
          void insert (Person * kp)
          {
                    kp-> Successor = Headpointer; // My successor will be my new beginning
                    Headpointer = kp; // and its successor is the new object I have created
          }
          // Remove a person from the beginning of the list
          Person * remove ()
          {
                    if (Headpointer!= 0)
                              Headpointer = tmp-> Successor;

                    return Headpointer;
          }

          // Access to the first person in the list
          Person * begin () { return Headpointer; }

          // Check if list is empty
          bool empty () const { return Headpointer == 0; } // Returns true if head pointer is 0, otherwise false
};
```

As you can see, the most important functions can be implemented with few lines of code. The default constructor creates an empty list by initializing the header pointer with NULL. When inserting a new element at the beginning of the list, two pointers must be modified:

- the successor of the element to be newly inserted, and
- the header pointer of the list must now point to the new element.

With such a simple program, even any number of people can be administered. There are only some further important functions missing. This list can only administrate existing persons. What function is in charge of creating new persons or of deleting persons that are no longer needed? This list can only handle existing objects that were previously created with the *new* operators and can be released with *delete*. Inserting an object in the list does not create a copy of the object.
We can **only** add new elements to the **beginning** of our list and also delete them from the beginning. A sequential data structure that allows access at only one end is also called a stack. The stack calls the operation of inserting new elements as **push** and deleting them as **pop**. This means that on a stack you can only put something on the top and take something away from the top, i.e. the stack works according to the LIFO principle (last in, first out). We can only edit the nodes in the forward direction because the information about the predecessor element is not available in the other nodes of the list. This is the disadvantage of a singly linked list.



5.1.3 Main program for the administration of people

```
int main ()
{
        ListOfPeople list; // Create empty list
        char ans = 'y';
        string surname="", firstname="";
        int yearOfBirth=0;
        Person * kp;

        // Filling the list
        while (ans != 'n')
        {
                cout << "Surname: " << endl;
                cin >> surname;
                cout << endl << "First name: " << endl;
                cin >> firstname;
                cout << endl << "Year of birth: " << endl;
                cin >> yearOfBirth;
                kp = new Person(surname, firstname, yearOfBirth);
                list.insert (kp);
                cout << "Go on (y/n)" << endl;
                cin >> ans;
        }

        // Display list
        for (kp=list.begin(); kp != 0; kp = kp->next()) // Start at the beginning of the list; Continue until NULL is reached.
                kp->displayData();                      // After each loop pass
                                                        // my new object is its successor
        // Empty list
        while (! list.empty ())                         // As long as my list is not empty
                list.remove ();                         // delete every single object (always the head pointer)

        system("pause");
        return 0;
}
```

5.2 Sequence containers

For singly linked lists, there are already various libraries that can be worked with to use the functionalities of a list. The introductory example uses the keyword *list*. In this program, positive numbers are entered, which are stored in a list and then output. The list is included and used by the existing library <list>.

Example: List

```
#include <iostream>
#include <list> // Library list

using namespace std;

int main()
{
        list<int> l;
        int x;
        cout << "Insert a positive number (Exit with 0):" << endl;
        do
        {
                cin >> x;
                if(x==0)
                   break;
                l.push_back(x); // Insert at the end of the list
        }
        while (true);

        list<int>::iterator i; // Iterator which reads the individual numbers in the list

        for (i=l.begin(); i != l.end(); ++i)      // Start at the beginning of the vector; until the end is reached
                cout << *i << " ";                // Increase the iterator by 1

        cout << endl;
        return 0;
}
```

It is also possible to sort the list. This can be done using the *sort* function that is defined in *list*. Sorting is performed by the command:

```
sort();
```

Therefore, the extended program looks as follows.

Example: Sorting the list

```
#include <iostream>
#include <list>                 // Library vector

using namespace std;

int main()
{
        list<int> l;
        int x;
        cout << "Insert a positive number (Exit with 0):" << endl;
        do
        {
                cin >> x;
                if(x==0)
                   break;
                l.push_back(x); // Insert at the end of the list
        }
        while (true);

        list<int>::iterator i; // Iterator which reads the individual numbers in the vector

        cout << "Before sorting: " << endl;
        for (i=l.begin(); i != l.end(); ++i) // Go through list and output it
                cout << *i << " ";

        cout << endl;

        l.sort(); // Sort the list

        cout << "After sorting: " << endl;
        for (i=l.begin(); i != l.end(); ++i) // Go through new list and output it
                cout << *i << " ";

        cout << endl;
        return 0;
}
```

The most important functions that we can use in a list are specified below:

Operation	**Function**
First node	begin
Last node	end
Insert at the end	push_back
Delete at the end	pop_back
Insert at the beginning	push_front
Delete at the beginning	pop_front
Insert anywhere	insert
Delete anywhere	erase
Empty list	clear
Sort list	sort

Example: Singly linked list using a container

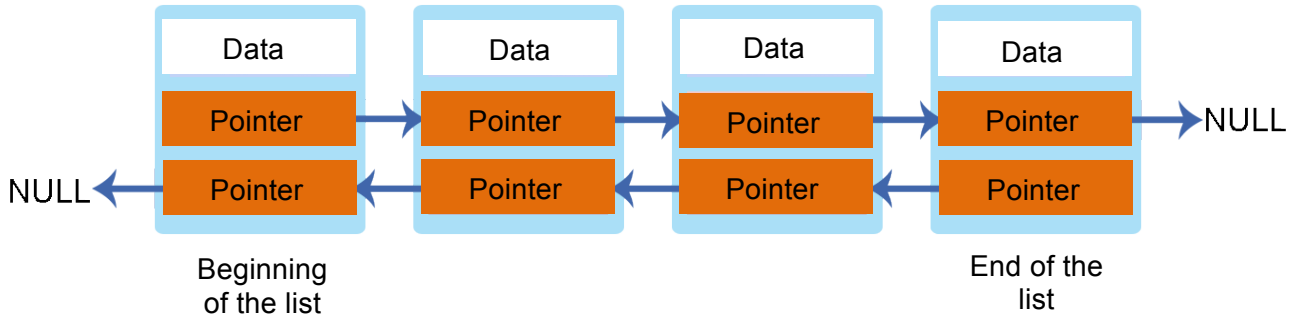
```
#include <iostream>
#include <list>
using namespace std;

void showlist(string, list<int>&);      // Declaration of the showlist function

int main()
{
   list<int> L; // List
   int x;
   cout << "Insert a positive number (Exit with 0):" << endl;
   do
   {
      cin >> x;
      if(x==0)
         break;
      L.push_back(x);                   // Insert at the end of the list
   }while (true);

   showlist("Initiallist:", L);  // Output initial list

   L.push_front(123);                   // Insert 123 at the beginning of the list
   showlist("After inserting 123 at the beginning:", L);

   list<int>::iterator i = L.begin();   // Iterator, assignment to the beginning of the list
   L.insert(++i, 456);                  // Increase iterator by 1, so it is the second node
                                        // Insert in second position

   showlist("After inserting 456 at the second node:", L);

   i = L.end();                         // Iterator, assignment to the end of the list
   L.insert(--i, 999);                  // Decrease iterator by 1 so that it is the penultimate node
   showlist("After inserting 999 at the penultimate node:", L);

   i = L.begin();                       // Iterator, assignment to the beginning of the list
   x = *i;                              // x has the value of the first node
   L.pop_front();                       // Delete first node from list
   cout << "Deleted at the beginning: " << x << endl;
   showlist("After deleting the first node:", L);

   i = L.end();                         // Iterator, assignment to the end of the list
   x = *--i;                            // x has the value of the last node (only i would be outside the list)
   L.pop_back();                        // Delete the last node
   cout << "Deleted at the end: " << x << endl;
   showlist("After deleting the last node:", L);

   i = L.begin();                       // Iterator, assignment to the beginning of the list
   x = *++i;                            // x has the value of the second node
   cout << "Delete: " << x << endl;
   L.erase(i);                          // Delete at the place of the iterator
   showlist("After deleting the second node:",L);

   L.clear();                           // Empty list
   showlist("After deleting the entire list:",L);

   return 0;
}

void showlist(string str, list<int> &L) // Definition of the showlist function: Returns the contents of the list
{
   list<int>::iterator i;
   cout << str << endl << " ";
   for (i=L.begin(); i != L.end(); ++i)
      cout << *i << " ";
   cout << endl;
}
```

5.3 Doubly linked lists, trees, graphs

Complex structures cannot be built with a single concatenation. Example of complex structures: Elements of a tree, branches and leaves, doubly linked list, etc.

5.3.1 Doubly linked lists

The nodes of a doubly linked list not only contain references to the successor, but also to the predecessor node. This makes it possible to search the list in both directions. This also makes it possible to insert a new node before or after another node. The disadvantage of this list is the additional memory space required for one pointer per node. The advantage, on the other hand, is the great flexibility.



Example:

```
#include <cstdlib>
#include <iostream>
#include <ctime>

using namespace std;

class Person
{
   friend class ListOfPeople; // friend declaration
   // Data members
   private:
      string surname;
      string firstname;
      int yearOfBirth;
      Person* Successor; // Concatenation pointer
      Person* Predecessor; // Pointer to the predecessor

   // Member functions
   public:
      Person (string n, string f, int y): surname(n), firstname(f), yearOfBirth(y) {}

      string getSurname () const { return surname; }
      string getFirstname () const { return firstname; }
      int getYearOfBirth () const { return yearOfBirth; }
      Person* getSuccessor () { return Successor; }

      void calculateAge () {
         // Determine current year
         time_t timestamp;
         tm *date;
         timestamp = time(0);
         date = localtime(&timestamp);
```

```
        int actYear = date->tm_year+1900;

        cout << "Age: " << actYear - yearOfBirth << " years old" << endl;
        }
    Person* next () { return Successor; }

    Person* pre() { return Predecessor; }

    void displayData() {
        cout << "Surname: " << surname << ", First name: " << firstname << ", Year of birth: " << yearOfBirth << endl;
    }
};

class ListOfPeople
{
  private:
    Person* Headpointer;   // Marks the beginning of the list -- contains the address of the first node
    Person* Tailpointer;   // Marks the end of the list -- contains the address of the last node
  public:
  // Constructor for an empty list
    ListOfPeople(): Headpointer(0), Tailpointer(0) {}

    // Inserting a person at the beginning of the list
    void insertBeginningList (Person * kp)
    {
        // Header pointer is assigned to the successor
        kp->Predecessor = 0;
        kp->Successor = Headpointer;

        if(Headpointer!= 0)
        {
            (kp->Successor)->Predecessor = kp;
        }
        else {
            Tailpointer = kp;
        }
    Headpointer = kp;
    }
    // Inserting a person at the end of the list
    void insertEndList (Person *kp)
    {
        // If the head pointer is 0, the address of the passed object is assigned to the head pointer. This only happens
once when the list is empty
        if(Headpointer == 0)
            Headpointer = kp;

        // If tail pointer is not 0, the successor of the previous object is updated. This is executed if the list is not empty.
        if(Tailpointer!= 0)
            Tailpointer ->Successor = kp;

        // Define end of list, i.e. assign successor 0
        kp->Successor = 0;

        // Predecessor is assigned the address of the tail pointer. Tail pointer contains the address of the previously
created object.
        kp->Predecessor = Tailpointer;

        // End pointer is assigned the address of the newly inserted object
        Tailpointer = kp;
    }
    // Removing a person from the top of the list
    Person * remove ()
    {
        if(Headpointer!= 0)
            Headpointer = Headpointer -> Successor;

        return Headpointer;
    }
```

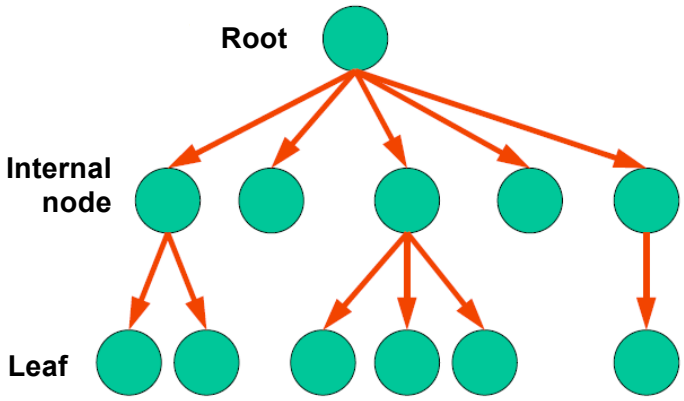
```
    // Access to first person in the list
    Person * begin () { return Headpointer; }

    // Access to last person in the list
    Person * end () { return Tailpointer; }

    // If the list is empty
    bool empty () const { return Headpointer == 0; }
};

int main ()
{
    ListOfPeople list; // Create empty list
    char ans = 'y';
    string surname="", firstname="";
    int yearOfBirth=0;

    Person *kp;

    // Filling the list
    while (ans != 'n')
    {
        cout << "Surname: ";
        cin >> surname;
        cout << endl << "First name: ";
        cin >> firstname;
        cout << endl << "Year of birth: ";
        cin >> yearOfBirth;
        kp = new Person(surname, firstname, yearOfBirth);
        // Elements are inserted at the end of the list
        list.insertEndList (kp);
        // or elements are inserted at the beginning of the list
        //list.insertBeginningList(kp);
        cout << "Go on (y/n)" << endl;
        cin >> ans;
    }

    // Output the list from the beginning
    for (kp=list.begin(); kp != 0; kp = kp->next())
        kp->displayData();

    // Output the list from the end
    for (kp = list.end(); kp!=0; kp = kp->vor())
        kp->displayData ();

    // Empty list
    while(! list.empty())
        list.remove();

    cout << "After emptying the list " << endl;

    // Output the list
    for (kp=list.begin(); kp != 0; kp = kp->next())
        kp->displayData();

    system("pause");
    return 0;
}
```

The example shows one of many options of what is possible with doubly linked lists. The example shows how elements can be inserted at the beginning or end of the list or output from the beginning or from the end of the list.

5.3.2 Trees

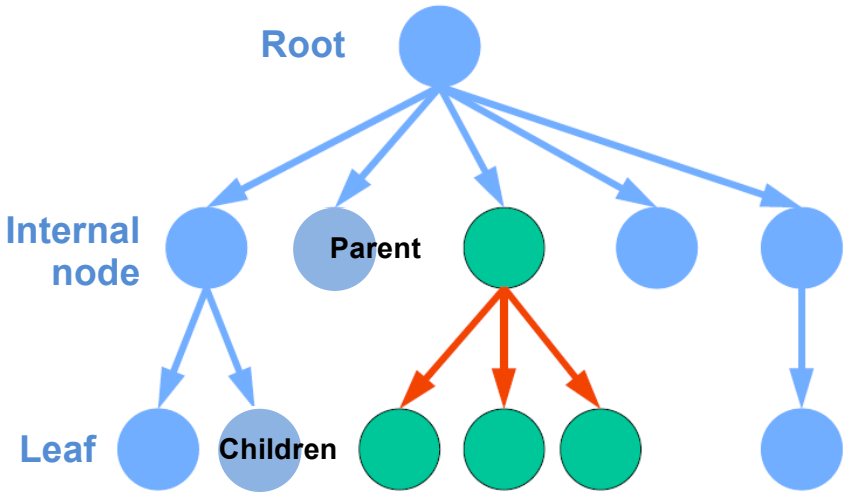
If we allow any number of successors, the result is a so-called tree:

- The elements of a tree are called nodes
- The successors of a node are called children
- Starting from the child, the following always applies:
  - Each child node has exactly one parent node
  - Except for a node that has no parent and is called root
- Nodes that no longer have children are also called leaves



A tree

- is empty or
- consists of a node, which in turn can refer to any number of trees.



Definitions like this are called recursive. Remember the definition of our list elements. This was also a recursive definition, since the individual node contains a reference to similar objects. Recursive algorithms are closely related to this.

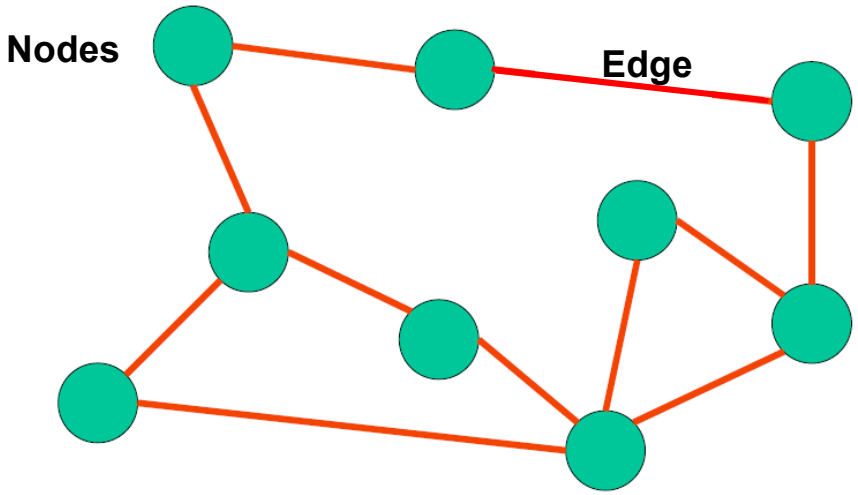
What kind of data can this tree structure be used for?

- The file system of your computer more or less forms such a tree
- The nodes are the data or subdirectories
- Leaves are always formed by files
- The root is the drive, e.g. C

5.3.3 Graphs

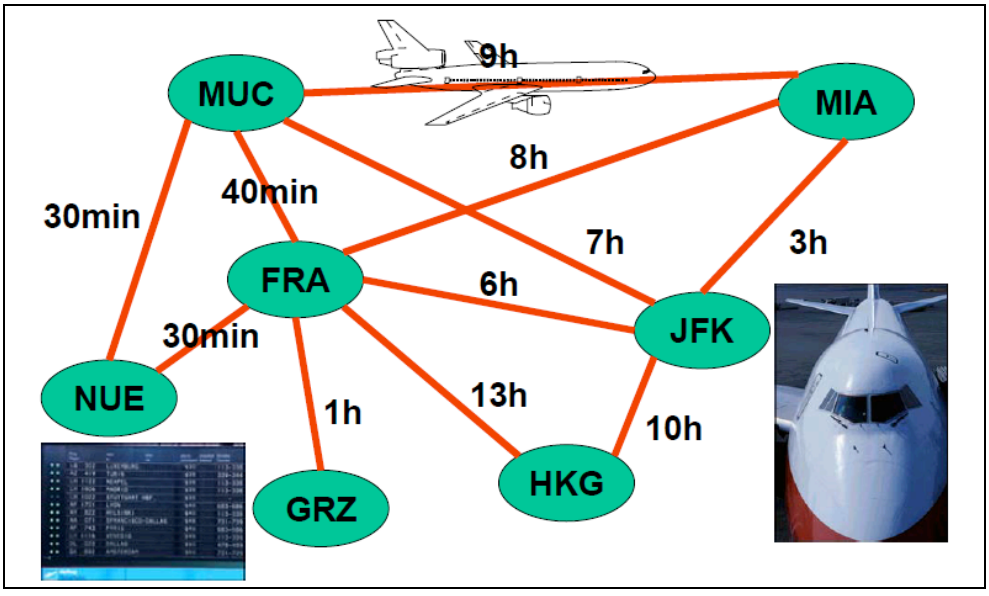
Are there any other data structures that can be formed with the help of pointers?

Probably the most common form of such data structures is called graphs.
A graph is a collection of nodes and edges. Nodes are simple objects. They have names and can hold values or properties. Edges are connections between these nodes.



One example would be direct flight connections in Germany. The airports in Germany form the number of nodes. An edge between two airports only exists if there is a direct flight connection. In addition, we can specify the flight time at the edges.

With this graph we can for instance solve the following interesting task: Find the cheapest connection between two airports, working on the assumption that there is no direct connection between them.



You can now use dynamic data structures in your own C++ programs if you have to manage an unknown number of objects. You have become familiar with various structures of these types:

- Singly linked lists
- Doubly linked lists
- Trees and graphs