

Problem 1

Write and test the program described below:

We define a structure describing nodes of a list

```
template <typename T>
struct Node {
    T      data;
    Node*  next;
};
```

(each node holds data of type **T**).

1. Write a function template

```
template<typename T, typename Pred>
Node<T>* arrayToList(const T arr[], size_t size);
```

taking an array and its size. The function should create a list of objects of type **Node<T>** containing (in members **data**) the elements of the array (in the same order). The function should return the pointer to the ‘head’ of the created list.

2. Write the template of a function

```
template<typename T, typename Pred>
Node<T>* extract(Node<T>*& head, Pred predicate);
```

taking a pointer to the ‘head’ of the list just created. The function should extract, to a separate (possibly empty) list, all nodes for which **predicate** applied to the member **data** returns **true**. The function **extract** returns the head of the list containing nodes for which the predicate yields **true**, and, after returning from **extract**, **head** points to the (possibly empty) list of nodes *not* satisfying the predicate (therefore, **head** may be modified by the function — that is why it is passed by reference).

NOTE: the function operates on *existing* **Nodes** only; it cannot create any new objects of structure **Node**.

In an invocation of the function **extract** one can pass, as the second argument, a function pointer, a lambda or a function object (functor).

3. Write a function template

```
void deleteList(Node<T>*& head);
```

which deallocates (using **delete**) all nodes of the list pointed to by **head**. When a node is removed, the function should print data from this node, so we can see that indeed the node is being removed. After returning from the function, **head** should be the empty pointer, as it represents a list which has just become empty.

4. Write a function (also in the form of a template) which prints a list passed to it: all elements in one line, separated by spaces or commas.

The scheme of the program:

```
#include <iostream>
#include <string>

template <typename T>
struct Node {
    T data;
    Node* next;
};

template <typename T>
void showList(const Node<T>* head);

template <typename T>
Node<T>* arrayToList(const T a[], size_t size);

template <typename T, typename Pred>
Node<T>* extract(Node<T>* head, Pred pred);

template <typename T>
void deleteList(Node<T>* head);

bool isLong(const std::string& s) { return s.size() >= 5; }

int main() {
    int tabi[] = { 2, 1, 4, 3, 6, 5, 7, 8 };
    size_t sizei = sizeof(tabi)/sizeof(tabi[0]);
    Node<int>* listAi = arrayToList(tabi, sizei);
    showList(listAi);
    Node<int>* listBi = extract(listAi,
                               [](int n) {
                                   return n % 2 == 0;
                               });
    showList(listBi);
    showList(listAi);
    deleteList(listBi);
    deleteList(listAi);

    std::string tabs[] { "Kasia", "Ola", "Ala",
                         "Zosia", "Ela", "Basia" };
    size_t sizes = sizeof(tabs)/sizeof(tabs[0]);
    Node<std::string>* listAs = arrayToList(tabs, sizes);
    showList(listAs);
}
```

```

        Node<std::string>* listBs = extract(listAs, isLong);
        showList(listBs);
        showList(listAs);
        deleteList(listBs);
        deleteList(listAs);
    }

```

This program should print something like:

```

2 1 4 3 6 5 7 8
2 4 6 8
1 3 5 7
DEL 2; DEL 4; DEL 6; DEL 8;
DEL 1; DEL 3; DEL 5; DEL 7;
Kasia Ola Ala Zosia Ela Basia
Kasia Zosia Basia
Ola Ala Ela
DEL Kasia; DEL Zosia; DEL Basia;
DEL Ola; DEL Ala; DEL Ela;

```
