Zadanie 1

Zdefiniować klasę **Rect** o prywatnych polach opisujących długości boków (**double**). Zdefiniuj

- konstruktor domyślny, tworzący kwadrat o boku 1;
- konstruktor jednoparametrowy, tworzący kwadrat o podanym boku;
- konstruktor dwuparametrowy (dwa boki);
- metody **getA()** i **getB** zwracające odpowiednie boki prostokąta;
- metodę **getDiagonal()** zwracającą długość przekątnej prostokąta;
- metodę **getArea()** zwracającą pole powierzchni prostokąta;
- metodę **isLargerThan(const Rect&)**, która zwraca **true** gdy *ten* prostokąt ma większe pole od tego przekazanego w argumencie, a **false** w przeciwnym przypadku;
- metodę **info()**, która wypisuje informację o prostokącie, na przykład w formie `Rect[2,3]]` (słowo 'Rect' i w nawiasach kwadratowych długości boków).

Przetestuj wszystkie konstruktory i metody w funkcji **main**.

Zadanie 2

Napisz (i przetestuj) szablon klasy **Queue** reprezentującej kolejkę (FIFO) elementów pewnego typu. Implementacja powinna opierać się na strukturze listy jednokierunkowej, dla której pamiętany jest wskaźnik head wskazujący na pierwszy element listy oraz tail wskazujący na element ostatni. Operacje, jakie są określone dla kolejki są następujące:

- konstruktor tworzący kolejkę pustą;
- **put** — dodaje element na końcu kolejki (bez przebiegania przez wszystkie elementy listy!);
- **get** — zwraca element z pierwszego węzła kolejki i usuwa ten węzeł;
- **empty** — odpowiada na pytanie, czy kolejka jest pusta;
- destruktor (który powinien coś pisać, abyśmy widzieli, które elementy są usuwane)..

Program może mieć następującą strukturę

download *QueueTemplate.cpp*

```
#include <iostream>

template<typename T>
class Queue {
    struct Node {
        // ...
    };
```

```
    Node* head;
    Node* tail;
public:
    Queue();
    bool empty() const;
    void put(const T& data);
    T get();
    ~Queue();
};

// implementation

int main() {
    int data1,data2;

    Queue<int>* q = new Queue<int>();
    q->put(1);
    data1 = q->get();
    std::cout << " data1=" << data1 << std::endl;

    q->put(1);
    q->put(2);
    data1 = q->get();
    data2 = q->get();
    std::cout << " data1=" << data1
              << " data2=" << data2 << std::endl;

    q->put(1); q->put(2); q->put(3);
    q->put(4); q->put(5); q->put(6);
    while (!q->empty())
        std::cout << " " << q->get();
    std::cout << std::endl;

    q->put(1); q->put(2); q->put(3);
    delete q;
}
```

i powinien wydrukować coś w rodzaju:

```
data1=1
data1=1 data2=2
1 2 3 4 5 6
Del:1 Del:2 Del:3
```

Zadanie 3

Napisz (i przetestuj) opisany niżej program:

Definiujemy szablon struktury opisującej węzły pojedynczo wiązanej listy

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

Napisz następujące funkcje:

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

template <typename T, typename Pred>
void removeBad(Node<T>*& head, Pred p);

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

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

gdzie

1. **arrayToList** pobiera tablicę elementów typu **T** i jej wymiar. Zadaniem funkcji jest utworzenie listy jednokierunkowej obiektów struktury **Node**, zawierającej w kolejnych węzłach kolejne elementy z przekazanej tablicy (w takiej samej kolejności). Funkcja zwraca wskaźnik do „głowy" utworzonej listy.
2. **removeBad** pobiera wskaźnik do „głowy" listy i predykat i usuwa wszystkie węzły listy zawierające dane dla których predykat zwraca **true**. Wskaźnik na głowę listy przekazany do funkcji jest modyfikowany i staje się wskaźnikiem na głowę nowej listy.
   UWAGA: Funkcja *nie* powinna tworzyć żadnych nowych węzłów. Jeśli wszystkie węzły zawierają „złe" dane, wszystkie powinny zostać usunięte, głowa staje sie wtedy **nullptr**. Przy każdym usuwaniu węzła funkcja powinna drukować wartość danej w nim zawartej, abyśmy widzieli, że rzeczywiście węzły te są usuwane.
3. **showList** drukuje zawartość listy (dane z kolejnych węzłów, w jednej linii, oddzielone znakami odstępu).
4. **deleteList** usuwa wszystkie węzły listy; wskaźnik do „głowy" przesłany jest przez referencję, aby funkcja mogła zmienić jego oryginał (na **nullptr**, co odpowiada liście pustej). Funkcja wypisuje informacje o usuwanych węzłach.

Przykładowy schemat programu:

download *ListsRemove.cpp*

```
#include <iostream>

template <typename T>
```

```
struct Node {
    T data;
    Node* next;
};

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

template <typename T, typename Pred>
void removeBad(Node<T>*& head, Pred p);

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

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

int main() {
    int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    size_t size = std::size(arr);
    Node<int>* head = arrayToList(arr,size);
    showList(head);
    removeBad(head, [](int n) {return n%2 != 0;});
    showList(head);
    removeBad(head, [](int n) {return n < 5;});
    showList(head);
    removeBad(head, [](int n) {return n > 9;});
    showList(head);
    deleteList(head);
    showList(head);
}
```

Program napisany według tego schematu powinien wydrukować:

```
0 1 2 3 4 5 6 7 8 9 10 11 12
DEL 1; DEL 3; DEL 5; DEL 7; DEL 9; DEL 11;
0 2 4 6 8 10 12
DEL 0; DEL 2; DEL 4;
6 8 10 12
DEL 10; DEL 12;
6 8
deleting 6; deleting 8;
Empty list
```
