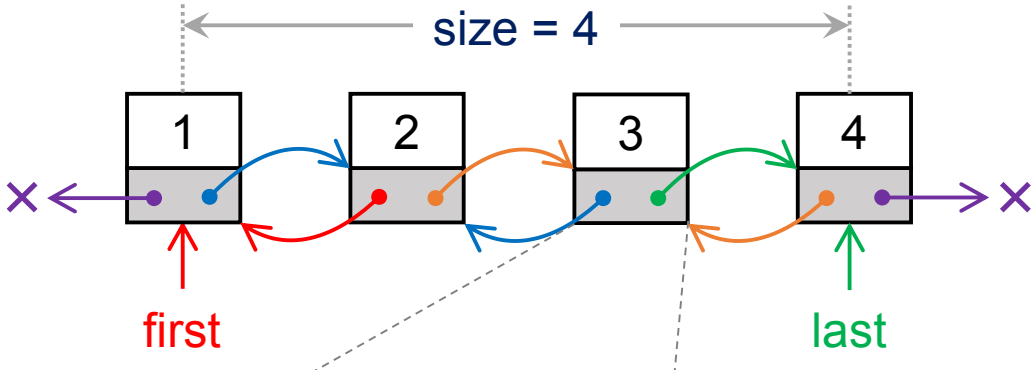
Объектно-ориентированное программирование

Курс «Технология программирования»

Кафедра управления и информатики НИУ «МЭИ»

Осень 2015 г.

Связанный список



Двусвязный список (doubly-linked list).

Элемент (узел, node)

Полезные данные value	
Указатель на предыдущий узел previous	Указатель на следующий узел next

Количество операций (⇒ скорость) при вставке и удалении элемента не зависит от количества имеющихся узлов.

- Один цвет — один адрес (значение указателя), кроме черных и серых (которые не важны).

Определение требований

Совершенно непонятно, как этого добиться.
Реализуем LinkedList только для **double**.

```
LinkedList<double> data { 2, 3, 4 };

data.push_back(5);
data.push_front(1);

data.erase(
    data.find_node_at(3));

cout << data;

}
```

Инициализация списком значений удобна.
Но даже пустому списку нужна инициализация.

Для удаления за $\mathcal{O}(1)$ нужен поиск за $\mathcal{O}(N)$.

При уничтожении списка требуется освободить и память под узлы.

Формализация требований

```
struct LinkedList {
   LinkedList(LinkedList& container);
   ~LinkedList();
   void push_back(double value);
      // push_front(double value), pop_front(), pop_back()
      // определяются аналогично.
   void clear();
   struct Node { … };
   void erase ( const Node* node );
   Node* find_node_at ( size_t index );
   // Вспомогательные члены опущены.
};
ostream& operator<<(ostream& output, const LinkedList& data);
```

Типы данных

- Узел списка:

```
struct Node
{
    double value;
    Node*  previous;
    Node*  next;
};
```

- Связанный список:

```
struct LinkedList
{
    Node* first;
    Node* last;
    size_t size;
    // …
};
```

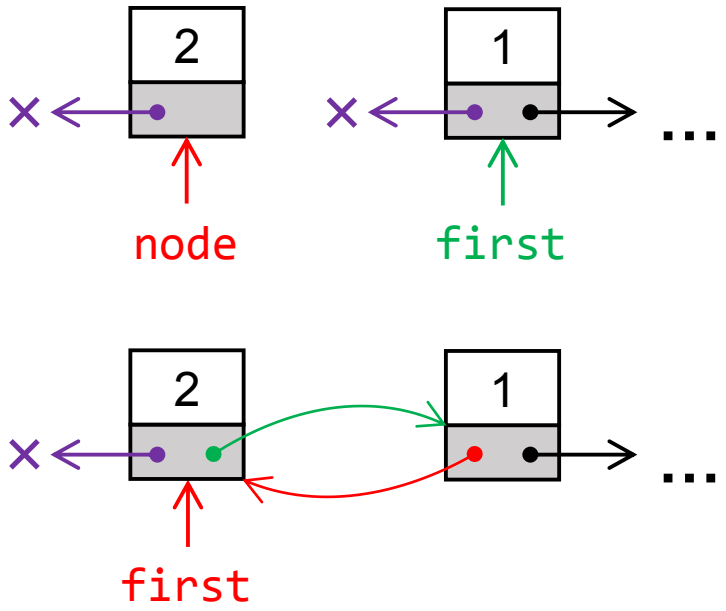
Беззнаковое целое
для размеров, индексов и т. п.

Перед использованием LinkedList (при инициализации переменной) будет вызван конструктор:

```
LinkedList :: LinkedList()
{
    first = nullptr;
    last = nullptr;
    size = 0;
}
```

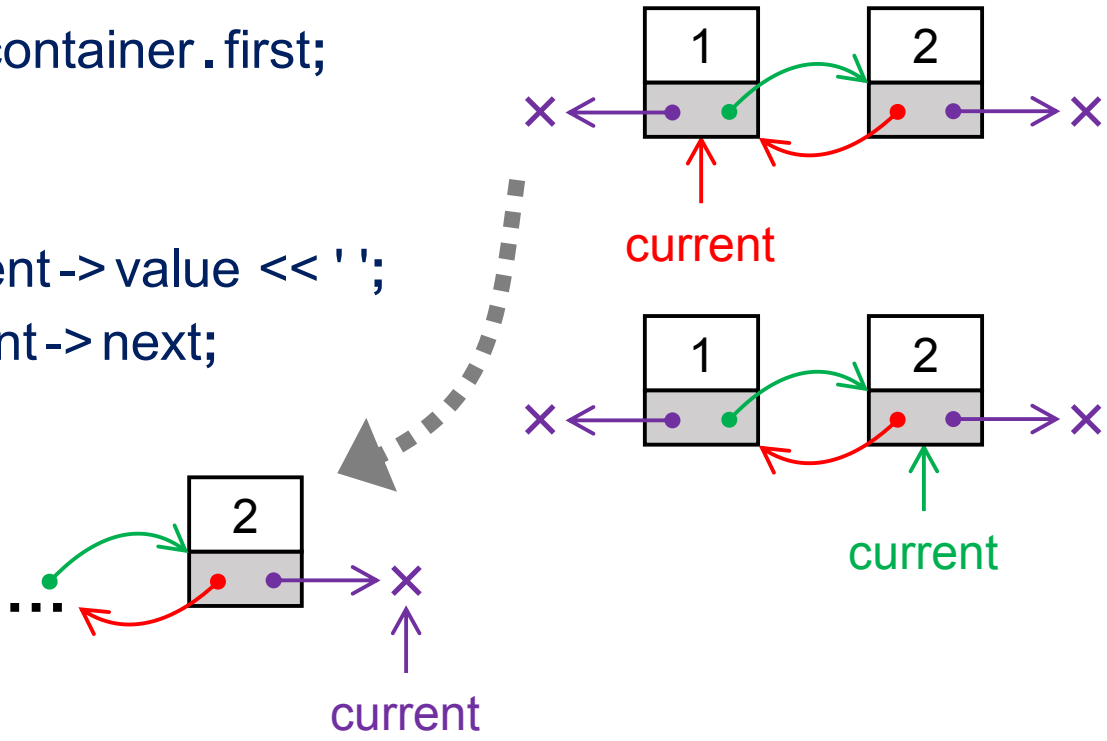
Добавление узла в список

```
void LinkedList :: push_front (const double& value)
{
    Node* node = new Node;
    node->value = value;
    node->previous = nullptr;
    node->next = first;
    if (first != nullptr) {
        first->previous = node;
    }
    first = node;
    if (last == nullptr) {
        last = node;
    }
    ++size;
}
```



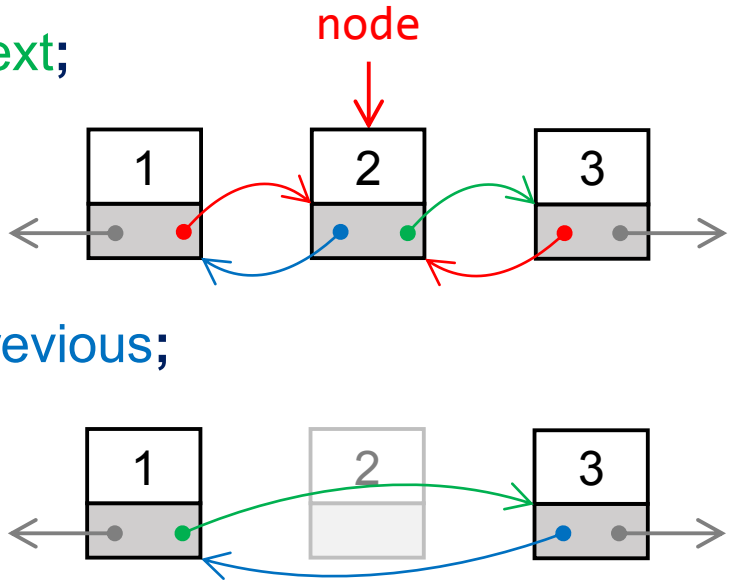
Обход и печать списка

```
ostream& operator<<(
    ostream& output, const LinkedList& container)
{
    Node* current = container.first;
    while (current)
    {
        output << current->value << ' ';
        current = current->next;
    }
    return output;
}
```



Удаление узла из списка

```
void LinkedList :: erase (const Node* node)
{
    if (node->previous)
        node->previous->next = node->next;
    else // node == first
        first = node->next;
    if (node->next)
        node->next->previous = node->previous;
    else // node == last
        last = node->previous;
    delete node;
    --size;
}
```



Поиск узла

```
Node* LinkedList::find_node_at(size_t index)
{
    Node* current = first;
    size_t current_index = 0;
    while (current) {
        if (current_index == index) {
            return current;
        }
        ++current_index;
        current = current->next;
    }
    return nullptr;
}
```

Даже при некорректном индексе
цикл будет остановлен
по достижении конца списка.

Очистка списка

«Чтобы очистить список, нужно,
пока список не пуст,
удалять из него элемент»

```
void LinkedList::clear()
{
    while (size)
        erase(first);
}
```

+ Корректирует size, first, last, поэтому реализация так проста.
− Совершает проверки и коррекции каждый шаг, не очень эффективно.

Тестирование

- Желательно автоматически проверять работу функций.
- Инструменты:
  - **bool** equals (
    const LinkedList& a_list, **const** vector < **double** > a_vector);
  - assert (*условие*) приводит к ошибке, если условие ложно,

    причем печатает условие
    и номер строки в программе.

```
void test_LinkedList_constructor()
{
    LinkedList data;
    assert ( data . size == 0   );
    assert ( equals(data, { }) );
}
```

Test cases (тестовые случаи):

- push_back():
  - в пустой список;
  - в список с элементами.
- erase():
  - начальный элемент;
  - конечный элемент;
  - элемент в середине;
  - **nullptr**;
  - элемент не из списка.

Конструктор и деструктор

```
struct LinkedList
{
    Node* first;
    Node* last;
    size_t size;

     LinkedList();
    ~LinkedList();
};
```

```
{
    LinkedList x;
    // …
}
```

Вызываются автоматически.

Почти то же самое:

```
LinkedList::LinkedList()
    : first{ nullptr }, last{ nullptr }, size{ 0 }
{ }

LinkedList::~LinkedList() { clear(); }
```

```
first = nullptr;
last = nullptr;
size = 0;
```

Инициализация списком

Цель: LinkedList data { 1, 2, 3, 4 };

```
struct LinkedList
{
    LinkedList() :
        first{nullptr}, last{nullptr}, size {0}
    { }

    LinkedList(initializer_list<double> values) :
        LinkedList()
    {
        for (double value : values) {
            push_back(value);
        }
    }
    // …
```

Конструктор по умолчанию сохраняется, он нужен.

Предварительно вызывается конструктор по умолчанию, чтобы обеспечить работу метода push_back().

Данные из списка инициализации копируются в связанный список.

Функции-члены (методы)

Цель:

```
LinkedList data;
data.push_back(1);
data.clear();
```

```
void LinkedList::clear()
{
    while (this->size)
        this->erase(this->first);
}
```

По умолчанию используются члены текущего (**this**) объекта.

```
struct LinkedList
{
    // …
    void clear();
    void erase(const Node* node);
    void push_back(double value);
};
```

```
void LinkedList::clear()
{
    while (size)
        erase(first);
}
```

Вложенные типы

Тип Node не связан с LinkedList, однако:

- Node без LinkedList не имеет смысла;
- у LinkedList, хранящего не **double**, а нечто иное, и Node должны хранить значение другого типа.

```
struct LinkedList {
    struct Node {
        double value;
        // …
    };
    // …
};
```

Node — вложенный тип (nested type).

```
void LinkedList::erase(
    const LinkedList::Node* node)
{
    // …
}
```

За пределами LinkedList нужно пользоваться полным именем Node.

Шаблоны типов

Цель:

- LinkedList < **double** > data**;**
- Не писать LinkedList заново для новых типов данных, хранимых узлами.

Какую идею нужно выразить?

«Для любого типа-параметра T

```
template < typename T >
```

существует структура LinkedList,

```
struct LinkedList {
```

узлы которой

```
    struct Node {
```

содержат значение типа T,

```
        T value;
    };
```

а методы оперируют значениями типа T.

```
    void push_back ( T value );
```

Реализация для любого T одинакова».

Шаблоны типов (продолжение)

```
template<typename T>
struct LinkedList {
    struct Node {
        T value;
        Node* previous, next;
    };
    void erase(const Node* node);
    ostream& operator<<(
            ostream& out,
            const LinkedList& xs);
};
```

Вместо **typename** допустимо **struct** или **class** (разницы нет).

Вне определения LinkedList<T> нужно использовать полное имя с типом-аргументом.

```
template<typename T>
void LinkedList<T>::erase(
    const LinkedList<T>::Node* node)
{
```

Контроль доступа

- Можно ли неправильно использовать LinkedList?

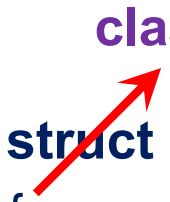
```
LinkedList<double> data;
Node* node = new Node<double>;
node->value = 42;
node->next = nullptr;
data.first = node;
```

  - data.size == 0, хотя элемент добавлен;
  - data.last == **nullptr**, хотя это неправильно;
  - node->previous не заполнено **nullptr**.

- *Если какая-нибудь неприятность может произойти, она обязательно случится.*

закон Мерфи

Спецификаторы доступа



Разрешается использовать всюду в коде.

Разрешается использовать только методам класса LinkedList.

Разница между **struct** и **class** — только уровень доступа по умолчанию:

```
struct X          ⟷   class X
{                     {
   // …                public:
                           // …
```

```
struct X          ⟷   class X
{                     {
private:                   // …
   // …
```

Функции-друзья класса

```
template < typename T >
ostream& operator << (
    ostream& output,
    const LinkedList < T >& container)
{
    Node < T > * current = container . first;
    while (current)
    {
        output << current -> value << ' ';
        current = current -> next;
    }
    return output;
}
```

Класс может разрешить доступ к своим закрытым членам некоторым функциям, объявив эти функции друзьями:

```
class LinkedList
{
    friend
    ostream& operator << (
        ostream& output,
        const LinkedList& xs);
};
```

Методы доступа к данным

- По-английски: getters *(в основном),* accessors, observers.
- Достижение:

```
LinkedList data;
data.first = new Node;  // нельзя
data.size++;            // нельзя
```

- *Всякое решение плодит новые проблемы.*

следствие № 7 из закона Мерфи

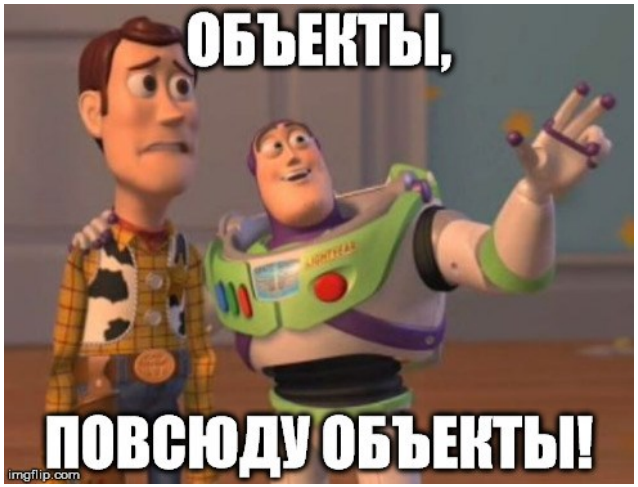
```
cout << data.size;  // нельзя
```

- Нужен метод для доступа (на чтение!) к размеру:

```
public:
    size_t get_size() { return size; }
```

Общий взгляд на объекты

- Каков смысл класса LinkedList?
  - Он представляет связанный список (является моделью связанного списка).
- Какие задачи решает оформление LinkedList в класс?
  - Хранение сообща необходимых данных (состояния).
  - Защита данных от нежелательных изменений.
  - Доступ к необходимым данным извне.
  - Предоставление методов для совершения операций над списком извне (**интерфейса**).
- Являются ли эти задачи частными?
  - Очень многое можно описать как объект или как систему объектов и отношений между ними.



Объектно-ориентированное программирование (ООП)

- **Класс** — модель категории вещей, обладающих определенным набором свойств и операций над ними.
- **Объект** — представитель некоего класса.
- «Программирование — это моделирование».
- **Инкапсуляция** — это…
  - (вообще) заключение данных и функций в единый компонент;
  - (в ООП) ограничение доступа к данным и методам в целях обеспечения согласованного состояния объекта («поддержания инварианта класса»).
- «Три кита» ООП: инкапсуляция, наследование, полиморфизм.

Отношения между объектами

- **Композиция (contains-a)**
  - Один объект является частью другого (и по смыслу, и в памяти).
  - Объекты создаются вместе и уничтожаются вместе.
- **Единоличное владение (exclusive ownership)**
  - Объект-владелец создает и разрушает подвластный объект в отдельной области памяти (управляет временем жизни).
  - Владение можно передать, потеряв его.
- **Совместное владение (shared ownership)**
  - Подвластный объект принадлежит нескольким владельцам и уничтожается, когда их не остается.
- **Осведомленность (reference)**
  - Один объект (A) имеет доступ к другому (B), но не контролирует время жизни B.
  - Объект B не должен быть разрушен раньше A!
    - Или у объекта A должна быть возможность узнать, разрушен ли B.

Композиция и осведомленность

```
class Student {
    string name;
    vector<Mark> marks;

    Group& group;

    Mentor* mentor;

    weak_ptr<Student> partner;
};
```

Композиция:
Имя студента и список его оценок хранятся в составе записи о студенте, создаются и уничтожаются вместе с ней.

Осведомленность:

У студента *всегда* есть группа, которая существует независимо от записи о студенте.

Научный руководитель существует независимо от студента, и его у студента *может не быть* (**nullptr**).

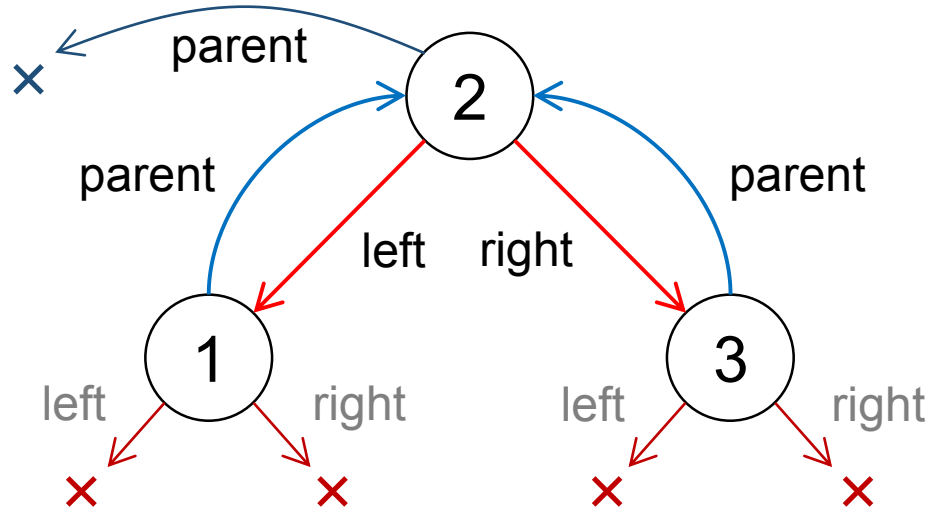
Напарник существует независимо, однако студент *может узнать, существует ли еще* запись о нем.

(Подробнее в лекции 8.)

Владение: узлы дерева

```
struct Node
{
    Node (
        double a_value,
        Node* a_parent )
    :
        value ( a_value ),
        parent ( a_parent )
    { }

    double value;
    Node* parent;
    unique_ptr < Node > left;
    unique_ptr < Node > right;
};
```



- **«Умный» указатель** единоличного владельца на подвластный объект.
- При уничтожении unique_ptr < T > (вместе с владельцем) уничтожается подвластный объект.
- Дочерние узлы **НЕ** хранятся как части родительского

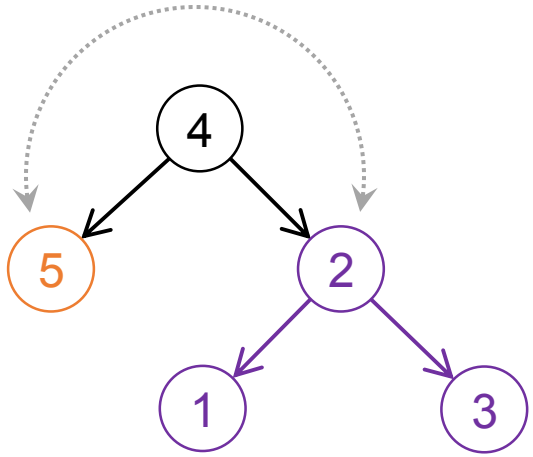
Передача владения

```
Node root ( 4, nullptr );
root . left = make_unique < Node > (2, &root);
root . right = make_unique < Node > (5, &root);
root . left . left =
    make_unique < Node > ( 1, root . left . get() );
root . left . right =
    make_unique < Node > ( 3, root . left . get() );


unique_ptr < Node > holder = move(root . left);
root . left = move(root . right);
root . right = move(holder);

                // swap(root . left, root . right);
```

Выделяет память и конструирует unique_ptr < T >.



Получение простого (raw, «голого», «сырого») указателя из «умного» (**smart pointer**).

Что скрывает unique_ptr<T>?

```
template<typename T>
class unique_ptr
{
public:
    // Управление временем жизни
    // подвластного объекта.
    ~unique_ptr () { delete pointer; }

    // Получить указатель.
    T* get () { return pointer; }

    // Принять во владение
    // новый объект new_pointer.
    void reset (T* new_pointer) {
        if (pointer) delete pointer;
        pointer = new_pointer;
    }

    // Освободить объект.
    T* release () {
        T* result = pointer;
        pointer = nullptr;
        return result;
    }

    // «Притворяться» указателем.
    T& operator*  () { return *pointer; }
    T*  operator->() { return   pointer; }

    // Не допускать копирования,
    // но разрешать перемещение.

private:
    T* pointer; // Подвластный объект.
};
```

- Как реализовать `x = move(y)` через методы `unique_ptr<T>`?

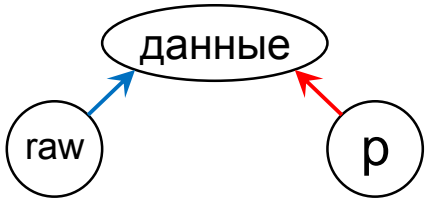
  `x.reset(y.release());`

- Почему не `x.reset(y.get())`?

  Метод `get()` не освобождает подвластный `y` объект:

- `int n = 42;`
  `unique_ptr<int> p(&n);`
  - Обязательно будет уничтожено `n`.
  - Обязательно будет уничтожен объект, подвластный `p` (то есть, `n`).

- `unique_ptr<int> p(new int{42});`
  - Можно.
  - Освобождение памяти — в деструкторе `p`.
  - Используйте `make_unique()`.
- `int* raw = new int{42};`
  `unique_ptr<int> p(raw);`
  - Указатели на один объект:



  - Память будет освобождена при уничтожении `p`.
    - **delete** `raw`?
      - Владеет только `p`.

Нераскрытые вопросы

- Как добавить методы готовому классу?
  - Не изменяя его?
  - Как создать новый класс на основе готового?
  - Если имеющимся методам нужно иное поведение в новом классе?
- Копирование объектов:
  - Можно ли передать LinkedList в функцию по значению?
  - Можно ли вернуть LinkedList из функции?
  - Можно ли присвоить один LinkedList другому?
    - Вопросы владения списка своими узлами.
- Работает ли **for** (**double** value **:** linked_list)?
  - Нет. Почему? Как исправить?
- Нельзя вызвать push_back() у **const** LinkedList — правильно, но нельзя вызвать и size() — почему и как исправить?

Литература к лекции

- *Programming Principles and Practices Using C++:*
  - глава 9 — материал лекции более подробно;
  - пункты 19.3.1, 19.3.2 — шаблоны и их применение.
- *C++ Primer:*
  - раздел 7.1 — классы и ООП;
  - раздел 7.2 — контроль доступа и инкапсуляция;
  - пункт 12.1.5 — «умный» указатель unique_ptr<T>.
  - пункт 16.1.2 — шаблоны классов.
- *Авторский конспект 2013 г.*

  http://uii.mpei.ru/study/courses/sdt-legacy