СИСТЕМЫ ПРОГРАММИРОВАНИЯ

ВЫЛИТОК
АЛЕКСЕЙ АЛЕКСАНДРОВИЧ

ВМК МГУ

КОНСПЕКТ ПОДГОТОВЛЕН СТУДЕНТАМИ, НЕ ПРОХОДИЛ ПРОФ. РЕДАКТУРУ И МОЖЕТ СОДЕРЖАТЬ ОШИБКИ. СЛЕДИТЕ ЗА ОБНОВЛЕНИЯМИ НА VK.COM/TEACHINMSU.

ЕСЛИ ВЫ ОБНАРУЖИЛИ ОШИБКИ ИЛИ ОПЕЧАТКИ, ТО СООБЩИТЕ ОБ ЭТОМ, НАПИСАВ СООБЩЕСТВУ VK.COM/TEACHINMSU.

БЛАГОДАРИМ ЗА ПОДГОТОВКУ КОНСПЕКТА
СТУДЕНТКУ ФАКУЛЬТЕТА ВМК МГУ
ЧЕРНИКОВУ ПОЛИНУ ГЕОРГИЕВНУ

Оглавление

Лекция 1. Введение в системы программирования 7
Системы программирования 7
Парадигмы программирования 8
Постулаты ООП 10
Декомпозиция задачи 11
Процедурно- и объектно-ориентированная декомпозиция задачи 11
Принципы объектно-ориентированной декомпозиции задачи 12
Синтаксис класса 13
Действия над объектами классов 14
Абстрактный тип данных (АТД) 15
Терминология 15
Некоторые отличия C++ от C 16
Работа с динамической памятью 16
Указатель this 18

Лекция 2. Методы класса 19
Специальные методы класса 19
Правила автоматической генерации специальных методов класса 20
Класс Box 21
Неплоский класс string 22
Переопределение операции присваивания 23
Композиция (строгая агрегация) объектов 24
Ссылочный тип данных. 24
Временные объекты 28

Лекция 3. Конструкторы и деструкторы 30
Порядок выполнения конструкторов и деструкторов 30
Вызов конструктора копирования 30
Друзья класса 33
Свойства друзей класса 33
Использование функций-друзей класса 34
Преимущества использования друзей-класса 35
Перегрузка операций 35

Лекция 4. Перегрузка функций и операций 42
Алгоритм поиска и выбора функции 42
Правила для шага 2 алгоритма выбора перегруженной функции 44
Одиночное наследование 48
Сокрытие имён (hiding) 49
Видимость и доступность имён 49
Вызов конструкторов базового и производного классов 50
Классы student и student5 51

Лекция 5. Динамический полиморфизм 53
- Виртуальные методы 53
- Виртуальные деструкторы 54
- Механизм виртуальных функций 54
- (механизм динамического полиморфизма) 54
- Абстрактные классы 55
- Реализация виртуальных функций 56

Лекция 6. Наследование 60
- Множественное наследование 61
- Правила выбора имён в производном классе 64
- Статические члены класса 65
- Средства обработки ошибок. Исключения в C++ 67
- Перехват исключений 67

Лекция 7. Динамическая идентификация типа 71
- Механизм RTTI (Run-Time Type Identification) 71
- Стандартные исключения 74
- Иерархия классов стандартных исключений 74
- Пример использования стандартных исключений 75
- void f () { 75
- Шаблоны 75
- Алгоритм выбора оптимально отождествляемой функции с учётом шаблонов 78
- Шаблоны классов 79
- Виды отношений между классами 80
- Стандартная библиотека шаблонов STL 83
- Стандартные контейнеры STL 84
- Состав контейнеров 84
- Типы, используемые в контейнерах 85
- Распределители памяти 85
- Итераторы 86
- Методы контейнеров для нахождения значений итераторов концов последовательности элементов 86
- Операции над итераторами 87
- Категории итераторов 88

Лекция 8. Алгоритмы 89
- Группы алгоритмов 89
- Категории итераторов и алгоритмы 90
- Контейнер vector 92
- Контейнер list 93
- Достоинства STL-подхода 96
- Введение в C++11 (стандарт ISO/IEC 14882:2011) 96
- Введения в C++11 rvalue-ссылки 97
- Семантика переноса (Move semantic) 97

Обобщённые константные выражения 99
Вывод типов 99

Лекция 9. Введение в C++11 101
For-цикл по коллекции 101
Улучшение конструкторов объектов 101
Явное замещение виртуальных функций и финальность 102
Константа нулевого указателя 103
Перечисления со строгой типизацией 104
Перечисления с не строгой типизацией 104

Лекция 10. Вычислительные системы 108
Структура вычислительной системы 108
Создание программного продукта (ПП) 108
Каскадно-возвратная модель 109
Итерационная модель 110
Основные компоненты системы программирования 110
Дополнительные компоненты систем программирования 111
Виды систем программирования 112
Интегрированная среда разработки (ИСР) 113
Текстовые редакторы 114
Задачи отладчика в рамках ИСР 115
Стратегии тестирования 115
Способы тестирования 116
Редактор связей 117
Типы библиотек 117
Динамически подключаемые библиотеки (ДБ) 119
Критерии проектирования стандартных библиотек. Требования по составу 119
Требования по свойствам компонентов стандартной библиотеки 120
СП под UNIX. Координатор GNU Make 121
Системы контроля версий 124
Развития систем контроля версий 124

Лекция 11. Трансляторы 130
Основные понятия теории формальных языков 133
Способы описания языков 134
Классификация грамматик и языков по Хомскому 138
Примеры грамматик и языков 140

Лекция 12. Трансляторы (2) 142
Контекстно-свободный класс грамматик 144
Регулярные языки 148

Лекция 13. Регулярные языки 152
Алгоритм построения ДКА по НКА 154
Алгоритм моделирования работы ДКА 155

Недетерминированный разбор 158
Лексический анализатор для М-языка 158
Проектирование структуры классов лексического анализатора М-языка 159

Лекция 14. Регулярные языки (2) 165
Синтаксический анализ 165
Метод рекурсивного спуска (РС-метод) 166
Синтаксический анализатор для М-языка 176

Лекция 15. Анализаторы языка 177
Семантический анализ 178
Семантический анализ для М-языка 179
Контроль контекстных условий в операторах 182
Внутреннее представление программы 184
ПОЛИЗ выражений 185

Лекция 16. ПОЛИЗ 187
Алгоритм Дейкстры перехода в ПОЛИЗ выражений 187
Представление операторов 188
Расширение набора операций ПОЛИЗа 188
Синтаксически управляемый перевод 189
Генератор внутреннего представления программы на М-языке 190

Лекция 17. ПОЛИЗ (2) 193
Интерпретатор ПОЛИЗ для модельного языка 193

Лекция 18. Распределение памяти 197
Классы памяти 197
Общие принципы генерации объектного кода 199
Машинно-независимые оптимизирующие преобразования 200
Машинно-зависимые оптимизирующие преобразования 202
Заключение 204

Лекция 1. Введение в системы программирования

Системы программирования

Иерархия вычислительной системы

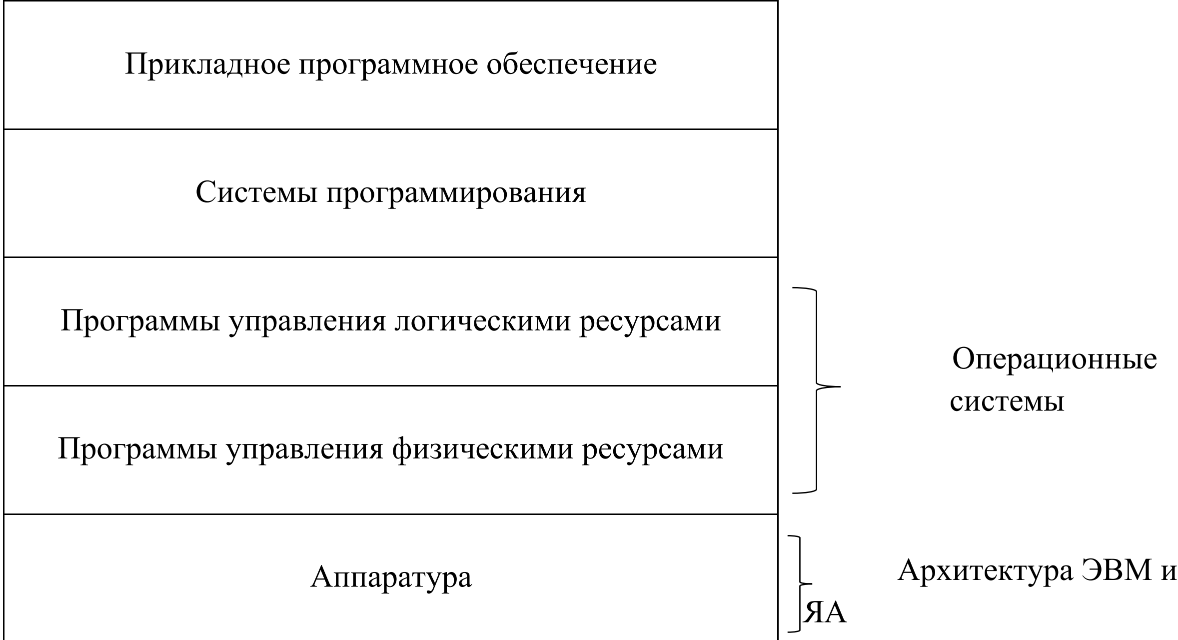


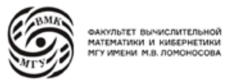
Рис.1.1 Иерархия вычислительной системы

Развитие СП:
- программирование в машинных кодах
- автокоды, языки ассемблера
- трансляторы с языков высокого уровня
- визуальные средства автоматизации и проектирования

Системной программирования (СП) называется комплекс программных средств (инструментов, библиотек), предназначенных для поддержки разработки программного продукта на протяжении всего жизненного цикла этого продукта

Данный курс включает:
- Основные понятия, назначения, структура и функционирование СП
- Принципы ООП на примере языка С++ и СП, поддерживающие ООП
- Элементы истории трансляции.

Список основной литературы курса находится на сайте по ссылке
http:cmcmsu.info/2course/



Язык C++
Автор языка C++ Бьёрн Страуструп предложил этот язык в 80-х годах прошлого века
Первоначально это была дополнительная надстройка в виде макросов на языке C.

Впоследствии язык развивался и были приняты стандарты в 1998, 2011, 2014, 2017г.г. и далее. C++ позволяет справиться с возрастающей сложностью программ в отличие от языка C.
Преимущества C++:
- лучше языка C
- поддерживает абстракции данных
- поддерживает объектно-ориентированное программирование (ООП).

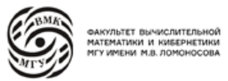
Парадигмы программирования

Все программы состоят из кода и данных, и каким-либо образом концептуально организованы вокруг своего кода и/или данных.

Основные парадигмы (технологии) программирования определяют способ построения программ:
- **процедурно-ориентированная** (где программа – это ряд последовательно выполняемых операций, причём код воздействует на данные, например, в программах на C)
- **объектно-ориентированная** (где программа состоит из объектов – программных сущностей, объединяющих в себе код и данные, взаимодействующих друг с другом через определённые интерфейсы, при этом доступ к коду и данным объекта осуществляется только через сам объект, т.е. данные определяют выполняемый код)
- **функциональная** (программа как набор функций и выполнение программы – это вычисление которые начинаются с выбора главной функции и продолжаются за счёт вызова других функций)
- **логическая** (набор логических утверждений, где есть утверждения, истинность которых мы не знаем и на основе имеющихся утверждений происходит доказательство или опровержение истинности нового утверждения)**.**

Объектная парадигма
Объект = состояние + поведение
Поведение = посылка сообщений себе и другим объектам
Для каждого вида сообщения существуют «обработчики», которые могут модифицировать состояние объекта и посылать сообщения другим объектам.

Первыми языками объектно-ориентированного программирования были Simula и Smalltalk, которые появились в середине 70-х годов прошлого века.



Язык С++ является смешанным, где в основе лежит процедурно-ориентированный стиль языка С.
Рассмотрим пример:

Запись **X** = **X**+**Y** трактуется в объектной парадигме так:
объекту **X** посылается сообщение «складывайся с объектом **Y** и измени своё состояние на новое – результат сложения».

Объекты с одинаковым поведением и возможными состояниями объединяются в *классы*. Классы образуют *иерархии*.
Иерархии – это способ борьбы со сложностью систем.

1. Рассмотрим пример иерархии «часть/целое» на примере автомобиля.

Автомобиль как целое, где выделяем подобъекты (двигатель, колесо).

Автомобиль
- Двигатель
  - поршень
  - камера
- Колесо
  - диск
  - шина

В ООП это называется *структурой объекта*.

2. Рассмотрим пример иерархии «общее/частное» на примере транспортного средства.

Транспортное средство как общее (класс), где выделяем подклассы (грузовое ТС и пассажирское ТС).

Транспортное средство
- Грузовое ТС
  - самосвал
  - бензовоз
- Пассажирское ТС
  - автобус
  - легковой автомобиль

В ООП это называется *структурой классов*.

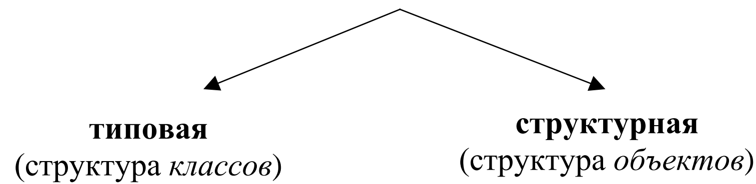
Объектно-ориентированное программирование (ООП) – это метод программирования, основанный на представлении программы в виде совокупности

взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы являются членами определённой иерархии.

Постулаты ООП

Абстракция – центральное понятие ООП.
Абстракция позволяет программисту справиться со сложностями решаемых им задач. Мощный способ создания абстракций:

Иерархическая классификация



типовая
(структура *классов*)

структурная
(структура *объектов*)

Рис.1.2 Иерархическая классификация

Основные механизмы (постулаты) ООП:

- Инкапсуляция
- Наследование
- Полиморфизм

1. **Инкапсуляция –** это механизм:

- связывающий вместе код и данные, которыми он манипулирует;
- защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому.

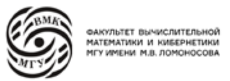
Доступ к коду и данным жёстко контролируется интерфейсом. Основой инкапсуляции является **класс.**
Класс – это механизм (пользовательский тип данных) для создания объектов.
Объект класса – переменная типа класса или экземпляр класса.

Любой объект характеризуется состоянием (значениями полей данных) и **поведением** (операциями над объектами, задаваемыми определенными в классе функциями, которые называются **методами класса**).

2. **Наследование** – это механизм, с помощью которого один объект (производного класса) приобретает свойства другого объекта (родительского **базового класса**).

Наследование позволяет объекту производного класса наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Производный класс конкретизирует, в общем случае *расширяет* базовый класс.



Наследование поддерживает концепцию иерархической классификации. Новый класс необязательно описывать, начиная с нуля, что существенно упрощает работу программиста.

3. **Полиморфизм** – механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

В общем случае концепция полиморфизма выражается с помощью фразы «один интерфейс – много методов». Выбор конкретного действия (метода) применительно к конкретной ситуации возлагается на компилятор. Программисту достаточно запомнить и применять один интерфейс вместо нескольких, что также упрощает его работу.

Различаются следующие виды полиморфизма:

- **статический** (на этапе компиляции, с помощью перегрузки функций);
- **динамический** (во время выполнения программы, реализуется с помощью виртуальных функций);
- **параметрический** (на этапе компиляции с использованием механизма шаблонов).

Декомпозиция задачи

Объектно-ориентированная парадигма предполагает, что мы столкнулись с достаточно объёмной задачей и необходимо провести декомпозицию задачи на подзадачи.

При программировании в объектно-ориентированном стиле на первое место выходит *проектирование* решения задачи, т.е. определение того, какие классы и объекты будут использоваться в программе, каковы их свойства и способы взаимодействия.

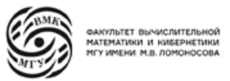
Как правило, при этом необходимо произвести декомпозицию задачи.

Декомпозиция – научный метод, использующий структуру задачи и позволяющий разбить решение одной большой задачи на решения серии меньших задач, возможно взаимосвязанных, но более простых.

Процедурно- и объектно-ориентированная декомпозиция задачи

Процедурно-ориентированная декомпозиция – вычленение из алгоритма решения задачи модулей, выполняющих некоторое самостоятельное действие (оформление некоторых действий в виде отдельных функций и процедур).

Рассмотрим *пример с автобазой*:
мы должны иметь классификацию транспортных средств, также, нам нужно описать объекты, касающиеся водителей, и их классифицировать по степени их квалификации (водитель грузовика, автомобиля легкового класса и т.д.). Далее выделяем некоторые



взаимосвязи, причём, некоторые объекты могут входить сразу в несколько объединений объектов.

Например, один и тоже автобус может ходить по рабочим дням по одному маршруту, а по выходным по другому маршруту, т.е. автобус может входить сразу в два множества.
Сначала нужно все взаимосвязи продумать, спроектировать систему взаимодействующих объектов, и только потом приступать к программированию.
Объектно-ориентированная декомпозиция – выделение элементов, принадлежащих различным абстракциям проблемной области (вычленение объектов проблемной области и определение их свойств).

Пример: студент, ВУЗ

В ВУЗе есть предметы и расписание, студенты взаимодействуют с преподавателями. Необходимо, чтобы студенты и преподаватель оказались в определённое время в определённой аудитории вместе.

Пример: стек/очередь

На языке Паскаль стек мы реализовывали как список из динамических объектов, связанных указателями между собой или в виде массива. При этом стек не был целостным, защищённым объектом, т.к. мы могли, если это массив, «залезть» в любой элемент стека.

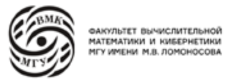
В данном случае, мы должны были работать со стеком только с помощью:
- операции Push положить на верхушку стека элемент и
- операции Pop вытащить с верхушки стека элемент.

Однако, мы могли нарушить эту дисциплину работы со стеком и заглянуть в любой внутренний элемент и даже его изменить.

Язык С и Паскаль не предлагают методов защиты.
При этом если описать стек как *класс* на языке С++, то стек станет обладать свойствами целостности и защищённости. В этом случае мы работаем со стеком через его интерфейс с помощью операций Push и Pop. Возможно ещё использовать операцию Empty, чтобы проверять стек на пустоту.

Принципы объектно-ориентированной декомпозиции задачи

1. Выделяемые элементы не следует делать слишком мелкими – это усложнит процедуры их координации и взаимодействия.
2. Удобно при выделении элементов представлять их в виде черного ящика, внутреннее устройство которого неизвестно, но определены выполняемые им действия и важные для внешнего использования «входы» и «выходы» (набор функций для получения/выдачи информации или изменения состояния элемента, т.н. ***интерфейс*** выделенного элемента).



3. Компоненты, в рамках одного выделенного элемента должны быть концептуально взаимосвязаны.
4. Для удобства и простоты использования выделенных элементов их интерфейс следует стремиться минимизировать.

Синтаксис класса

class имя_класса {
[private:]
    закрытые члены класса (функции, типы и поля-данные)

public:
    открытые члены класса (функции, типы и поля-данные)
protected:
    защищённые члены класса (разрешается при наследовании использовать)
} список_объектов

Описание объектов – экземпляров класса:
    имя_класса список объектов;
        // служебное слово class не требуется

Классы С++ отличаются от структур С++ только правилами определения по умолчанию:
- **прав доступа** к первой области доступа членов класса и
- **типа наследования**:
    для структур – **public**
    для классов – **private.**

Например, как только встречается при описании класса слово **public,** то это означает, что всё, что написано ниже после этого слова является *открытым/публичным* до тех пор, пока не встретиться новое ключевое слово **private** *(личное)***или protected** *(защищённое)*. Можно повторять **public**/ **private/ protected** в любом порядке по несколько раз.

Смысл в следующем, что до конца класса он будет таким как слово в начале *класса*, до тех пор, пока не встретится новое ключевое слово из этих трёх **public**/ **private/ protected**.

Если мы используем ключевое слово **struct** при описании класса, то все элементы класса по умолчанию открыты. Если мы хотим что-то закрыть, то надо перед ними написать **private:**

Отличие между способами класса **struct** проявляется, когда мы будем производить наследование на основе базового класса.

Члены класса

- Члены-данные
- Члены-функции (методы)
- Члены-типы – вложенные пользовательские типы (область видимости - класс)

Правила доступа к членам класса и поиска их имен единообразны для всех членов класса и не зависят от их вида.

Ex.:
```
class X {
        double t:       // Данное
public:
        void f ( );  // метод
        int a;       // данное
enum  { e1, e2, e3 } g;
private:
        struct inner { // вложенный класс
        int i, j;
        void g ( );
        };
        inner c;
  };
…
X x; x.a = 0; x.g = X::e1;
```

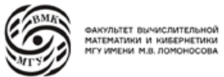
Действия над объектами классов

Над объектами классов можно производить следующие действия:

- присваивать объекты одного и того же класса (при этом производится почленное копирование членов данных)
- получать адрес объекта с помощью операции &
- передавать объект в качестве формального параметра в функцию
- возвращать объект в качестве результата работы функции
- осуществлять доступ к элементам объекта с помощью операции **‘ . ’** , в случае если используется указатель на объект, то с помощью операции **‘ -> ’**
- вызывает методы класса, определяющие поведение объекта.

Пример класса

```
…
class A {
        int a;
```



```
public:
        void  set_a ( int n);
        int get_a ( ) const { return a;}        // Константные методы класса
                                                // не изменяют состояние своего объекта
};

void A::set_a (int n) {
        a=n;
}

int main () {
        A obj1, obj2;
        obj1.set_a (5);
        obj2.set_a (10);
        count << obj1.get_a ( ) <<'\n';
        count << obj2.get_a ( ) << endI;
        return 0:
}
```

Примечание: **::** - операция раскрытия области видимости

Абстрактный тип данных (АТД)

АТД называют тип данных с полностью скрытой (инкапсулированной) структурой, а работа с переменными такого типа происходит только через специальные, предназначенные для этого функции.

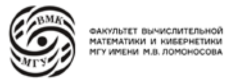
В языке С++ АТД реализуется с помощью классов (структур), в которых нет открытых членов-данных.

Класс А из предыдущего примера на стр.13 является абстрактным типом данных, т.к. все члены-данные (поле а) закрытые.

Терминология

Оператор / Statement – действие, задаваемое некоторой конструкцией языка.
Операция / Operator (для обозначения языка: **+**, *****, **=** и др.) – используются в выражениях.
Определение (описание) переменной / (Definition) – при этом отводится память, производится инициализация, определение возможно только один раз.
Объявление переменной (Declaration) – даёт информацию компилятору о том, что эта переменная где-то описана в программе.

Для преобразования типов используются два термина – **преобразование** (Conversion) и **приведение** (Cast).



Некоторые отличия С++ от С

- Введён логический тип ***bool*** и константы логического типа ***true*** и ***false*** (где 0 – это ложь/ *false*, а 1— это истина/ *true*)
- В С++ отсутствуют типы по умолчанию (например, обязательно int main () {…}).
- Локальные переменные можно описывать в любом месте программы, в частности, внутри цикла **for**. Главное, чтобы они были описаны до их первого использования (соблюдался принцип «сначала опиши, а потом используй»).

По стандарту С++ переменная, описанная внутри цикла **for,** локализуется в теле этого цикла (вне тела существовать уже не будет).

Пример:

for (int i=1; i<10; i++) {…}

- В С++ переработана стандартная библиотека. В частности, в стандартной библиотеке С++ файл заголовков ввода/вывода называется **<iostream>**, введены классы, соответствующие стандартным (консольным) потокам ввода – класс **istream** – и вывода **–** класс **ostream,** а также объекты **cin** (класса **istream**) и **count** и **cerr** (класса **ostream**).

Через эти объекты доступны операции ввода из стандартного потока ввода>**>** (например, **cin** >> x;), и вывода **<<** в стандартный поток вывода (например, count << ‘string << S <<’\n’), при использовании которых не надо указывать никакие форматирующие элементы.

Пример:

Для того чтобы подключить библиотеку в С нужно было написать:

stdlib.h => cstlib

В С++ для этого необходимо:

#include <cstdlib>

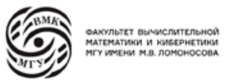
Язык С++ позволяет изменять смысл операций. С++ переписан для потоков так, чтобы удобно писать ввод >>/ вывод <<.

Работа с динамической памятью

int *p,* m;
p = **new int**; *или*
p = **new (nothrow) int**; *или*
p = **new int** (1); *или*
m = **new int** [10]; - для массива из 10 элементов;

Массивы, создаваемые в динамической памяти инициализировать нельзя;

delete p; *или*
delete [ ] m; - для удаления всего массива



Операция **new** размещает объект подходящего типа в динамической памяти. Язык С++ предусматривает отказ выделения памяти. Операция **new** будет выбрасывать объект исключения.

Операция **new** параметр **nothrow** позволяет возвращать нулевой показатель (null) если не удалось выделить память.
Если объект в динамической памяти перестал быть нужным, необходимо освобождать память с помощью операции **delete.**

Значения параметров функции по умолчанию

Пример:

```
void f ( int a, int b=0, int c=1);
```

Обращения к функции:

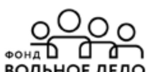
```
f (3)     // a = 3, b = 0, c = 1;
f (3, 4)  // a = 3, b = 4, c = 1;
f (3,4,5) //  a = 3, b = 4, c = 5.
```

Если какой-то из параметров имеет значение по умолчанию, то все параметры, которые расположены правее него, тоже должны иметь значение по умолчанию.

Пространства имён

Пространства имён вводятся только на уровне файла, но не внутри блока.
В стандартной библиотеке уже предусмотрено пространство имён ***std.***

```
namespace std {
	// объявления, определения
}
Ex:	std::cout <<::endI;

namespace NS {
	char name [10]:
	namespace SP {
		int var = 3;
	}
}
Ex:	…NS::SP::var += 2;

#include <iostream>
using namespace std;
```
- эта директива позволяет использовать имена напрямую,



using NS:: name…;   которые прописаны в std

Имена функций между собой не конфликтуют, если они отличаются по количеству или по типу параметров. Это называется *совместное использование*. При этом имена объектов без префиксов не могут быть использованы в данном случае.

Указатель this

Иногда для реализации того или иного метода возникает необходимость иметь указатель на «свой» объект, от имени которого производится вызов данного метода.

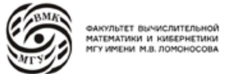
В C++ введено ключевое слово **this,** обозначающее «указатель на себя», которое можно трактовать как неявный параметр любого метода класса:

<имя класса> *** const this;**
***this** – сам объект

Таким образом, любой метод класса имеет на один (первый) параметр больше, чем указано явно.

This, участвующий в описании функции, перегружающий **операцию,** всегда указывает на самый левый (в выражении с этой операцией) операнд операции.

В реальности поле **this** не существует (динамическая память не расходуется), и при сборке программы вместо **this** подставляется соответствующий адрес объекта.



Лекция 2. Методы класса

Специальные методы класса

Конструктор – метод класса, который:
- имеет имя, в точности совпадающее с именем самого класса,
- не имеет типа возвращаемого значения (как пример, **void** не указываем),
- **всегда** вызывается при создании объекта (сразу после отведения памяти под объект в соответствии с его описанием).

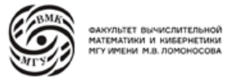
Статические объекты, которые находятся вне функции, существуют от начала и до конца работы программы. Находятся в статической памяти.

Глобальные объекты, которые описываются внутри блока, память под такие объекты отводится в системной области стека программы и существуют до конца блока. При выходе из блока память освобождается, указатель стека передвигается. Из-за этого некоторые фреймы выбрасываются и далее, освободившееся место памяти может быть задействовано под другие объекты.

Деструктор – метод класса, который:
- имеет имя, совпадающее с именем класса, перед первым символом которого прописывается символ **~,**
- не имеет типа возвращаемого значения и параметров,
- **всегда** вызывается при уничтожении объекта (перед освобождением памяти, отведённой под объект).

Пример (рассмотрим заголовки конструкторов):

```
class A {
……
    public:
        A ( );          // конструктор умолчания.
        A (A&y);        //A (const A&y); конструктор копирования (КК)
[explicit] A (int x);   // конструктор преобразования (иногда может вызываться //
                        неявно; explicit означает, что оно может присутствовать
                        или отсутствовать; запрещает компилятору неявное
                        преобразование int в A
    A (int x, int y);   //
    // A (int x = 0, int y = 0);    // заменяет 1-й, 3-ий и 4-ый
                                    // конструкторы
    ~A ( );                         // деструктор
    …..
};
```

```
Int main () {
    A a1, a2 (10), a3 = a2;
    A a4, a5 = A (7);       // Err!, т.к. временный объект не может быть
                               параметром для не константной ссылки в КК
                               О.К., если будет A (const A&y)
    A* a6 = new A (1);
    Где с помощью операции new
```

В С++ временный объект прицеплять можно только к ссылке, указывающей на константный объект (*пример*: A (**const** A&y)).

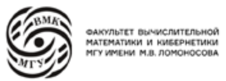
Некоторые конструкторы описывать программисту необязательно, достаточно описать класс, и мы уже имеем несколько автоматически сгенерированных конструкторов.

Правила автоматической генерации специальных методов класса

- Если в классе явно не описан никакой конструктор, то конструктор умолчания генерируется автоматически с пустым телом в **public** области.
- Если в классе явно не описан конструктор копирования, то он **всегда генерируется автоматически** в **public** области с телом, реализующим почленное копирование значений полей-данных параметра конструктора в значения соответствующих полей-данных создаваемого объекта.
- Если в классе явно не описан деструктор, то он **всегда генерируется автоматически** с пустым телом в **public** области.

Если мы вручную написали хотя бы один конструктор (с любым количеством параметров), то тогда мы теряем *конструктор умолчания,* автоматически сгенерированный в классе. В таком случае необходимо вручную прописывать в классе *конструктор умолчания.* Он есть только в том случае, когда других явно описанных конструктов в классе нет.

Автоматически генерируется *конструктор копирования*, который реализует почленное копирование значение полей данных в значения соответствующих полей-данных, создаваемого объекта (т.е. копирует один объект в другой). Он не боится наличия явно описанных других конструкторов.
Мы можем описать в классе какой-нибудь конструктор с одним параметром и не описывать явно конструктор копирования, т.е. можем рассчитывать на автоматически сгенерированный *конструктор копирования,* который переписывает почленно каждое поле бит в бит. Если нас не устраивает такое копирование, то можем явно написать так, как нам нужно в рассматриваемом нами классе.



Деструктор генерируется автоматически в классе с пустым телом, т.е. никаких действий не выполняет. Положено перед уничтожением объекта, чтобы перед выполнением действий вызывался *деструктор.* Деструктору ничего не надо делать, т.к. устранение объектов не приведёт ни к каким последствиям, т.к. он всегда присутствует в классе автоматически сгенерированный.

Класс Box

```
class Box {              //     класс Box – коробка
        int l;           //     length – длина
        int w;           //     width – ширина
        int h;           //     height – высота
public:
        int volume () const {return l * w * h;} () // const не изменяет состояние объектов
        Box (int a, int b, int c) { l = a; w = b; h = c;} // конструктор с тремя параметрами
        Box (int s) { l =  w = h = s;}   // конструктор, с одним параметром, где
                                длина/ширина/высота равны заданному параметру
        Box ( ) { w = h = 1; l = 2;}          // конструктор умолчания
        int get l ( ) const {return l;}       // конструктор get – геттеры, открытые
                                              функции, находящиеся в открытой части
                                              класса, позволяют запросить значение длины/
                                              ширины /высоты
        int get w ( ) const {return w;}
        int get h ( ) const {return h;}
};
```

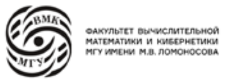
Автоматически сгенерированный конструктор копирования и операция присвоения:

```
Box (const Box & a) {C = a.l; w = a.w; h = a.h;}
```

где, а – это объект, переданный нам по ссылке с параметрами **const** Box не будет изменяться (*l* – приватное поле, на которое ссылается *a* (объект одноклассник), позволяет реализовать конструктор копирования, т.к. мы находимся в одном и том же классе Box).

```
Box & operator = (const Box & a) {l = a.l; w = a.w; h = a.h; return * this;}
```

где операция присваивания (**operator=),** автоматически сгенерированная, возвращает ссылку на Box (на объект, которому только что было что-то присвоено). Это реализовано для того, чтобы как в С можно было выписывать целые цепочки присваивания: x = (y = z) – группировка справа налево у операции присваивания



*** this –** сам объект, в который произошло копирование полей, и он себя возвращает по ссылке в тот контекст, в котором встретилась операция присваивания.

Конструктор копирования и операцию присваивания можно переопределить. Допускается совмещение в одном конструкторе свойств сразу нескольких конструкторов.

Пример:

```
class A {
public:
        A (int a1=0, int a2=0, int a3=0);
}
…..
A x;
A y (1);
A z (1, 2);
A w (1, 2, 3);
```

Неплоский класс string

```
class string {
        char * p;        // здесь потребуется динамическая память, приватное поле p
                         хранит символы строки
        int size;        // поле хранит длину этой строки
public:
        string (const char * str);       // конструктор позволяет по строке сконструировать
                                         объект типа string нашего класса
        string (const string & a);       // конструктор копирования, если уже есть объект
                                         типа string, то можно создать его копию (другой
                                         объект)
        ~string ( ) {delete [ ] p; }     // деструктор, освобождает память
        string & operator = (const string & a); // операция присваивания
…..
};
string :: string (const char * str) {     //
        p = new char [ (size = strlen (str)) + 1] //
        strcpy (p, str);                 // копирование выделенной памяти в строку
}
string :: string (const string & a) {
        p = new char [ (size = a.size) +1];  // a – ссылка на объект, с которого берётся
                                             копия
```

```
        strcpy (p, a.p);
}
```

Пример использования класса string

```
void f {
        string s1 ("Alice");    s1

        string s2 = s1;         s2

        string s3 ("Kate");     s3
        …….
        s3 = s1;
}
{…s1…s2 {s3}…s1…s2}
```

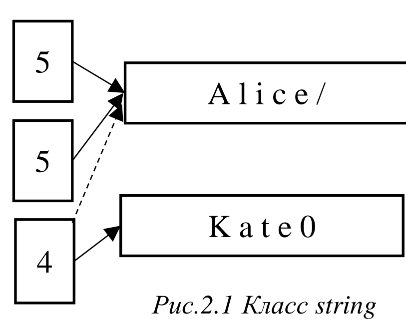


Рис.2.1 Класс string

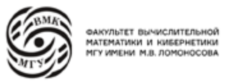
На рисунке 1 показан пример использования класса *string*. Разберём пример подробно.

string s1 ("Alice"); s1 // создали с помощью конструктора преобразования всей строки string объект s1 длины 5, из него идёт указатель (стрелка) на строчку, выделенную в динамической памяти
string s2 = s1; s2 // далее описали все поля из первого объекта и длина и указатель скопируются один в один, и мы получим, что два указателя указывают на одну и ту же строчку
string s3 ("Kate"); s3 // создали с помощью конструктора преобразования всей строки string объект s3 длины 4

Следует избегать поверхностного копирования (см.рис.1 отмечено пунктирной стрелкой). Приведёт к тому, что указатель из объекта s3 будет указывать на объект s1. Строчка *Kate* потеряется и станет мусором, невозможно будет освободить память из-под этой строки. Это плохо, поэтому необходимо явно описать операцию присваивания, которая по-своему устройство очень похожа на операцию копирования.

Переопределение операции присваивания

```
string & string :: operator = (const string & a){
if (this = =& a)                 // this – это адрес текущего объекта, в который
                                 происходит присваивание значения. & a - адрес
                                 объекта, который был передан по ссылке параметра
        return *this;                        // если a=a
delete [ ] p;
p = new char [ (size = a. size) + 1];
strpy (p, a.p);
return *this;
```



}
При этом: s1 = s2 ~ s.1. **operator** = (s2)

Композиция (строгая агрегация) объектов

```
class Point {
        int x;
        int y;
public:
        Point ();
        Point (int, int);
        ….
};
```

```
class Z {
        Point p;
        int z;
public:
        Z (int c) {z = c};
….
};
```

Предположим, что где-то в функции *main* мы сделали следующее описание:
где z – это указатель на объект класса

```
Z * z = new Z (1);          //  Z Point (); Z (1);
delete z;                   // ~Z (); ~Point ();
```

Когда происходит уничтожение объекта с помощью операции **delete,** сначала должен проработать деструктор класса Z, потом деструкторы тех подобъектов, которые есть внутри классов в порядке обратному их описанию. Конструкторы работают в прямом порядке, а деструкторы в обратном.

Использование **списка инициализации** при **описании** конструктора:
В этом списке мы можем явно указать какой именно конструктор должен быть вызван для инициализации того или иного подобъекта класса.

Z :: Z (**int** c) : p (1, 2) {z = c} или
Z :: Z (**int** c) : p (1, 2) z (c) { }

Разберем на примере как описывать:

Z :: Z (**int** c)   - через :: повторить имя класса и потом перед телом p написать : , далее будет находится список инициализации , где можно указать через запятую имена подобъектов и количество параметров.

Ссылочный тип данных.

Ссылочный тип данных задается так: **<тип> &.**

Ссылка (reference) – переменная ссылочного типа. Ссылки – это просто синонимы. Не путать с указателями. С указателями мы работаем через * (звездочку). Указатель – это просто адрес.

Единственная **операция над ссылками – инициализация** (установление связи с инициализатором) при создании, при этом ссылка обозначает (именует) тот же адрес памяти, что и её инициализатор (L-value выражение).

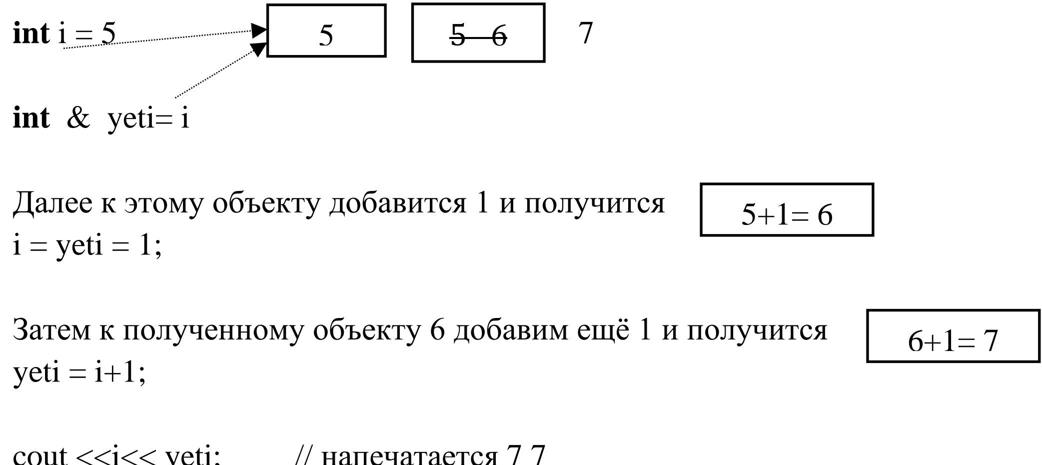
После описания и обязательной инициализации ссылку можно использовать точно так же, как и соответствующий её инициализатор.
Фактически ссылка является синонимом своего инициализатора.

Ссылочный тип данных в C++ используется в следующих случаях:

а). Описание переменных-ссылок (локальных или глобальных)

Например,
int i = 5;   // переменная целого типа получила значение 5
int & yeti = i // ссылка обязательно должна быть инициализирована
// yeti – синоним имени i; & i = & yeti

Предположим, что где-то в памяти расположена переменная *i* и связь с этим объектом в памяти осуществляется как по имени *i,* так и по имени *yeti*. С этого момента к объекту, хранящему целое число, мы можем обращаться и так, и так.



cout <<i<< yeti;   // напечатается 7 7

б). Передача параметров в функции по ссылке

Инициализация формального L-value выражение параметра ссылки происходит в момент передачи фактического параметра (), и далее все действия, выполняемые с параметром-ссылкой, выполняются с соответствующим фактическим параметром.

Пример:

```
void swap (int & x, int & x ){
        int t = x;
        x = y;
        y = t;
}
```

Пример обращения к функции swap:

```
int a = b;
swap (a, b);
```

В C++ удобно передавать параметры по ссылке (обращению). По значению работает конструктор копирования.

c). Возвращение результата работы функции в виде ссылки –
для более эффективной реализации функции, т.к. не надо создавать временную копию возвращаемого объекта и в том случае, когда возвращаемое значение должно быть L-value выражением.

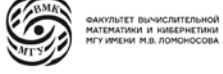
Инициализация возвращаемой ссылки происходит при работе **return** операндом которого должно быть L-value выражение. Необходимо позаботиться, чтобы этот объект продолжал существовать.

Не следует возвращать ссылку *на локальный объект функции,* который перестает существовать при выходе из функции. Локальные объекты при выходе из функции разрушаются, память из-под них освобождается и получается, что ссылку мы вернём, но самим объектом не сможем воспользоваться корректно, т.к. он будет разрушен.

Возвращать следует ссылку на внешний объект, который мы получили по ссылке или на себя на тот же объект (например, *** this** использовали в примере с классом string см. выше) или на динамический объект, который мы создали в динамической памяти, который будет существовать и после возврата из функции. Ссылку на динамический объект можно передавать, при этом, необходимо позаботиться, где потом уничтожить этот объект, чтобы память из-под динамического, когда он станет не нужным была освобождена (использовать **delete**).

Пример:

```
int & f ( ) {     // функция f возвращает ссылку на объект в динамической памяти
        int t = x;
        int * p = new int (5)
        return * p;
}
```

Пример обращения к функции f: **int** &x = f ( );
где x описали как ссылку на **int** и вернули ей результат функции f, т.е. проинициализировали эту ссылку результатом функции f. Это будет объект динамической памяти.

д). Использование ссылок – членов-данных класса

Инициализация поля-ссылки класса обязательно происходит через список инициализации конструктора, вызываемого при создании объекта.

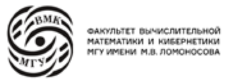
Пример:

```
class A {
    int x;              // где x подобъект стандартного типа
public:
    int & r;            // где r является открытым ссылочным полем
        A () :  r(x) {  //ссылка в списке инициализации, где  r инициализируем
                        объектом x
        x = 3;          // где x внутри конструктора получает значение 3
    }
            A (const A&) ; //!!!
            A & operator = (const A&); //!!!
    ….
};
int main () {   //
    Aa;
    ….
}
```

Для того, чтобы можно было копировать объекты данного класса или присваивать их необходимо самим написать соответствующий конструктор и операцию присваивания, т.к. ссылки невозможно присвоить стандартным образом (автоматически сгенерированные ссылки). Необходимо чтобы адрес **r** указывал на ту же область памяти, что и адрес **x**.
Когда мы опишем в *main* объект **a** *класса* A, то произойдёт следующее:
- сначала должен проинициализироваться подобъект,
- далее сработает конструктор класса A, где поле **x** будет = **3.**

е). Константные ссылки

Использование **ссылок на константу –** формальных параметров функций (для эффективности реализации в случае объектов классов).
Ссылка на константу задаётся так: **const** int & .



Инициализация параметра – ссылки на константу происходит во время передачи фактического параметра, который, в частности, может быть **временным объектом,** сформированным компилятором для фактического параметра-**константы**.

По ссылке в качества параметра можно передавать объекты как константные, так и не константные, при условии, что это не временные объекты и что объект, указанный в качестве фактического параметра, не является константным.

В С++ введено правило, что передавать временные объекты можно только на константную ссылку. Это правило предотвращает возможность менять что-то через ссылку во временном объекте, т.е. после того, как он создан и с него сняли копию, временный объект исчезает.

В случае передачи временного объекта по ссылке в функцию временный объект будет жить до того момента пока из функции мы не выйдем. Разрывается связь между временным объектом и функцией, объект исчезает.

Временный объект можно передавать только со ссылкой на **const** (константный тип)**.**

Пример:

```
struct A {
    int a;
    A (int t = 0) {a = t;}
};
int f (const int & n, const A & ob) {
    return n+ob.a;

}
int main () {
    cout <<f (3,5) <<endf;
….
}
```

Временные объекты

Временные объекты создаются в рамках выражений (в частности, инициализирующих), где их можно модифицировать (применять неконстантные методы, менять значения членов-данных)
Если есть класс А и в нём есть поле а в открытой области **public,** то возможен такой синтаксический вариант:

A (). A = 1;

где будет вызван конструктор класса А и в результате вызова создан временный объект, который можно модифицировать (присвоить значение). При этом **нельзя** инициализировать ссылку на константу временным объектом.

В общем случае «живут» временные объекты до окончания вычислений соответствующих выражений.

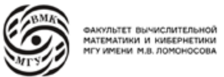
Однако, если инициализировать ссылку на константу с временным объектом (в частности, передавать временный объект в качестве параметра для формального параметра – ссылки на константу), время его жизни продлевается до конца жизни соответствующей ссылочной переменной.

Нельзя инициализировать неконстантную ссылку временным объектом (в частности, неконстантные ссылки – формальные параметры).

Пример:

```
struct A {
    A (int);
    A (const A &);
    };
….  const A & r = A (1);    // если здесь и в КК убрать const,
Aa1 = A (2);                //все эти конструкции будут
Aa2 = 3; ….                 // ошибочными
```

Важно! Компилятор **всегда** сначала *проверяет* синтаксическую и семантическую (контекстные условия) правильность, а затем *оптимизирует*.



Лекция 3. Конструкторы и деструкторы

Порядок выполнения конструкторов и деструкторов

При вызове **конструктора** класса выполняются:

1) Конструкторы базовых классов (если есть наследование)
2) Конструкторы умолчания всех вложенных объектов в порядке их описания в классе
3) Собственный конструктор (все поля класса уже проинициализирован, следовательно, их можно использовать).

Деструкторы выполняются в обратном порядке:

1) Собственный деструктор (при этом поля класса ещё не очищены, следовательно, доступны для использования)
2) Автоматически вызываются деструкторы для всех вложенных объектов в порядке, обратном порядку их описания в классе)
3) Деструкторы базовых классов (если есть наследование).

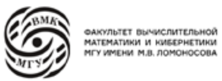
Вызов конструктора копирования

1) Явно:
   мы можем обратиться к конструктору класса A ( ) ; ,
   означает, что создаётся временный объект и его значение инициализируется конструктором умолчания класса A.

   Аналогично можно вызвать конструктор копирования для временного объекта a и он будет инициализирован с помощью конструктора копирования: A (a) ;
   Временные объекты сразу исчезают, как только мы выходим за рамки выражения: все временные объекты, которые в нем возникли, утилизируются. Предварительно перед их исчезновением проработает деструктор.

2) В случае инициализации объектом того же типа:
   Box a (1, 2, 3);
   Box b = a; // a – параметр конструктора копирования, b создаётся с помощью копирования информации из объекта a

3) В случае инициализации временным объектом:
   Box c = Box (3, 4, 5);
   Box (3, 4, 5); - означает вызов конструктора с тремя параметрами



```
// сначала создаётся временный объект и вызывается
// обычный конструктор с тремя параметрами, а затем работает конструктор
// копирования для инициализации объекта c; если компилятор
// оптимизирующий, вызывается только обычный
// конструктор с указанными параметрами.
```

Стандарт языка C++ допускает в таких случаях оптимизацию. Временный объект, созданный с помощью конструктора с тремя параметрами видно, что сразу исчезнет. Компилятор может не создавать его, а сразу инициализировать объект *c* конструктором с тремя параметрами (т.е. пропустить шаг копирования, но он обязательно проверит, что это принципиально возможно, т.е. существует конструктор копирования, который позволит скопировать).

Если конструктор копирования убрать в закрытую часть описания этого конструктора, то в таком случае пользоваться им вне класса невозможно. Компилятор сделает оптимизацию и откажется выполнять это описание, скажет, что здесь ошибка, т.к. конструктор копирования убран в закрытую часть.

4) При передаче параметров функции по значению (для инициализации локального объекта)
   При передаче параметром по значению происходит копирование из точки вызова фактических параметров в функцию, где создаются локальные объекты, соответствующие формальным параметрам и их значения, инициализируются копирующими конструкторами из соответствующих фактических параметров.

5) При возвращении результата работы функции в виде объекта
   Если функция объявлена так, что её результат возвращается не по ссылке, а по значению, то работает конструктор копирования для того, чтобы с объекта, указанного в качестве параметра **return,** или получившегося в результате вычисления выражения, указанного в операторе возврата **return ,** копируется с помощью конструктора копирования в точку вызова функции как результат работы функции.

6) При генерации исключения объекта.

Другие конструкторы также могут вызываться явно:

- конструктор умолчания (вызовется автоматически при создании объекта)
- при создании объектов динамической памяти (выделяем память и над этой памятью работает соответствующий конструктор)
- при композиции объектов (когда сложный объект, то вызываются конструкторы для членов класса)

- при конструировании объекта производного класса на базе другого класса (сначала проработает конструктор базового класса)
- конструкторы преобразования (конструкторы с один параметром, тип которых отличен от ссылки на класс или константы на класс).

Деструкторы можно вызвать только от имени какого-то объекта: *a.*
Всегда вызывается стандартным образом: *a. ~A ( );*
Такая практика опасна, т.к. деструкторы должны вызываться автоматически.

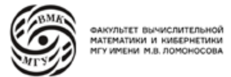
При операции **delete** деструкторы вызываются автоматически, когда мы освобождаем память, захваченную с помощью операции **new.** Вызывается соответствующий деструктор для проработки перед тем, как память будет освобождена.

Принцип работы конструктора и деструктора таков:
если конструктор что-то захватил, например, какую-то строку разместил в динамической памяти, то соответствующий деструктор класса должен это освободить. Если при инициализации мы захватываем какие-то ресурсы, то при работе деструктора мы должны все эти ресурсы освободить, чтобы не замусоривать динамическую память, т.к. в дальнейшем может не хватить для других целей при работе в программы.

При явном вызове деструктора мы заставили его проработать явно один раз, также, деструктор вызовется автоматически ещё один раз, когда объект *a* выйдет из области видимости. Например, при выходе из блока объект *a* был автоматический, значит объект должен исчезнуть и деструктор вызовется второй раз. При повторной работе деструктора могут произойти ошибки.
В общем стандарт языка С++ полагает, что если деструктор явно вызвали, то объект уже не существует. Попытка второй раз применить деструктор к этому объекту может закончить хорошо или плохо (зависит от особенностей реализации).

Иногда явный вызов деструктора бывает полезен. Это делается в том случае, когда объект с помощью специальной формы операции **new** привязывается к определённому участку памяти, которая уже была заранее выделена для размещения объекта. В этом случае деструктор следует вызвать явно, а саму память утилизировать теме же способами, которыми она была выделена.

Итак, закрытые области **private** и **protected** позволяют осуществлять защиту от случайного изменения полей объекта во время использования класса, у которого есть закрытая часть. Из функции, которая не принадлежит классу не имеем права доступа к закрытым полям. Не имеем права вызывать закрытые методы класса. Это предотвращает много случайных ошибок, когда мы допустили опечатку и захотели присвоить что-то закрытому полю. Если бы поле не было закрытым, компилятор пропустил бы такое присваивание, и мы случайно испортили бы состояние объекта.



Если реализацию внутри класса сделали закрытой, то компилятор сразу отловит такую ошибку и при компиляции скажет, что такое присваивание недопустимо (доступ посмотреть значение закрыт).
Инкапсуляция даёт программисту больше удобств, он делает меньше ошибок. В С++ всегда доступна операция взятия адреса.

Друзья класса

Друг класса – это функция, не являющаяся членом этого класса, но имеющая доступ к его **private** и **protected** членам.

Для того, чтобы некоторая функция могла иметь доступ ко всем членам класса её нужно объявить другом. Это осуществляется с помощью служебного слова **friend.** В любой области описания класса **private**/ **protected/ public** и т.д. Пишем заголовок, а перед заголовком служебное слово **friend.**

Своих друзей класс объявляет сам в любой зоне описания класса с помощью служебного слова **friend.**

Функция-друг может быть описана внутри класса.

Если функций, имена которых совпадают с объявленной в классе функцией-другом несколько, то другом считается только та, у которой в точности совпадает прототип.

Другом класса может быть:
- обычная функция: **friend void** f (….);
- функция-член другого класса: **friend void** Y:: f (….);
- весь класс: **friend class** Y; // друзьями становятся все методы класса Y

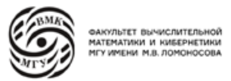
Свойства друзей класса

Дружба не обладает ни наследуемостью, ни транзитивностью.

Примеры:

```
class A{
    friend class B;
    int a;
};
```

```
class B {
    friend class C;
};
```

```
class C {
        void f (A*p) {
                p -> a++;          // ошибка, нет доступа к закрытым членам класса
        }
};

class D:   public B {
    void f (A*p) {
        p -> a++;          // ошибка, нет доступа к закрытым членам класса
    }
};
```

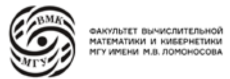
Если мы хотим, чтобы из C или из D можно было пользоваться всеми членами класса A, в таком случае их тоже надо объявить друзьями в классе A.
Тот класс, который предоставляет свои приватные поля другим, он и пишет служебное слово **friend** с указанием какой именно метод другого класса или какая внешняя функция считается его другом, и получает доступ. Поскольку другу можно всё, все поля доступны, даже закрытые, то с особой осторожностью следует относиться к выбору дружественных функций и аккуратно их программировать.

Если класс A объявил другом класс B, то это не значит, что для класса B класс A тоже является другом. Это может быть так если внутри класса B написать **friend class A.** Они становятся взаимными друзьями. Если это не указать, то будет в одну сторону: B может иметь доступ к приватным членам класса A, а наоборот нет.

Использование функций-друзей класса

Примеры:

```
class X {
        int a;                         // закрытое поле a
        friend void fff (X*, int);     // внешняя дружественная функция fff, имеет
                               указатель на объект X*, здесь нет this!
  public:
        void mmm (int);                // открытая функция mmm
};

        void fff  (X*p, int i) {       // указатель слова friend здесь не требуется, т.к. оно
                                       имеет  смысл только внутри класса
        p-> a = i;             // здесь мы видим, что в реализации функции
                               происходит доступ к закрытому члену a (ему что-то
                               присваивается)
```

```
}

void X :: mmm (int i) {        // реализация метода mmm, вынесенная вне класса,
                               имя mmm скрыто внутри класса X,
                               применяем разрешение области видимости mmm
                               (X ::)
        a = i;                 // а описано ранее в классе X
}

void f () {
    X obj;                     // описываем объект класса X
    fff (& obj, 10);           // обращаемся к функции fff, указывая адрес данного
                               объекта
    obj.mmm (10);          // тоже самое происходит если вызвать функцию mmm

}
```

Преимущества использования друзей-класса

1) Эффективность реализации (можно обходить ограничения доступа, предназначенные для обычных пользователей класса).

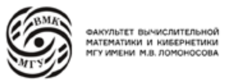
2) Функция-друг нескольких классов позволяет упростить интерфейс этих классов.

3) Функция-друг допускает преобразование своего первого параметра-объекта, а метод класса нет.

Перегрузка операций

- Для перегрузки встроенных операций С++ используется служебное слово **operator.**
- Перегружать операцию можно с помощью:
  - метода класса
  - внешней функций, в частности, функции друга (что менее эффективно)
- Нельзя перегружать:
  '.', **'::'**, **'?: '**, **'.*'**, **sizeof** и **typeid**

Перегрузка (overloading)/ совместное использование означает, что в одной области видимости могут быть несколько функций с одним и тем же именем,



например, **f** или несколько операций **+.** Они могут вызываться для разного типа параметров.

В языке С данную операцию невозможно было выполнить.

Язык С++ помогает нам осуществить статический полиморфизм за счёт перегрузки операций, а также функций. Одним и тем же именем могут называться разные вещи. Например, когда мы читаем книгу по матанализу или линейной алгебре и видим сложение (a + b), то всегда понимаем из контекста, что такое (a + b). Это сложение векторов или матрицы или чисел.

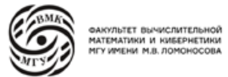
Также и в языке С++ можно перегрузить операцию + таким образом, чтобы выражение (a + b) одинаково записывалось и для матриц, и для чисел, и для чего-то ещё, что можно складывать. При этом метод алгоритм сложения будет вызываться именно тот, который нужен:
- для чисел будет вызываться подпрограмма, которая складывает числа,
- для матриц та, которая складывает матрицы.

Выглядят названия этих методов совершенно одинаково.
Это явление в С++ называется *статическим полиморфизмом*.

В языке С++ можно перегрузить практически все операции за исключением: **‘.’**, **‘::’**, **‘?: ’**, **‘.*’**, **sizeof** и **typeid.**

Рассмотрим *пример 1:*

```
class complex {                 // класс комплексных чисел

    double re, im;              // два закрытых класса типа re и im типа double,
                                означающий вещественно-мнимые части; конструктор
                                умолчания (преобразования)
public:
    complex (double r = 0; double i = 0) {     re = r;
                                               im = i;
};
complex operator+ (const complex & a) {  // имя функции operator+, которую
                                         перегружаем как член класса. Эта операция
                                         один из методов класса.

    complex temp (re + a.re, im + a.im); // опишем локальный объект класса complex и
                                         инициализируем конструктором с двумя
                                         параметрами (суммы вещественных и мнимых
                                         частей)
```



teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

return temp; // возврат в качестве результате. Будет работать конструктор копирования (если сами не написали его, то конструктор копирования сгенерится автоматически)

}….

// **operator double** () {**return** re;} – функция преобразования позволяет превратить комплексные части в вещественные. Конструктор с одним параметром, наоборот, умеет превращать вещественные в комплексные части

};

int main () {

complex x (1, 2), y (5, 8), z; // объекты x, y, z с параметрами

double t = 7.5; // переменная t класса типа **double**

z = x + y; // x + y – комплексные числа, если + описан как метод класса, то компилятор считает это выражением таким способом: преобразует его к виду: z. operator = (**x. operator + (y));**

z = z + t; // O.K.: z. operator + (complex (t)); если есть функция преобразования, то неоднозначность: ‘+’ для **double** или перегруженный

z = t + x; // Err.! – т.к. первый операнд по умолчанию типа complex. где t – это класс типа **double**.

}

Рассмотрим <u>*пример 2*</u> *(как можно операцию + определить с помощью функции друга):*

class complex {

double re, im;

public:

complex (**double** r = 0; double i = 0) {

re = r;

im = i;

};

friend complex **operator+** (**const** complex & a, **const** complex & b); // два параметра в

// операции **+**, передаем их по константной ссылке, не будет

// вызова конструктора копирования при передачи параметров

// возвращать будем по значению, копировать результат в точку вызова

….

};

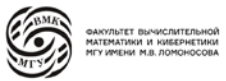
complex **operator+** (**const** complex & a, **const** complex & b) {

```
        complex temp (a.re + b.re, a.im + b.im);      // создаём локальный объект temp,
                                    // складываем соответственные вещественные мнимые
                              // части конструируем нужные значения и возвращаем его

              return temp;            // возвращаем в точку вызова операции +
}
int main (){
        complex x (1, 2), y (5, 8), z;
        double t = 7.5;
        z = x + y;      // O.K. – operator + (x, y): как операция x с двумя параметрами
        z = z + t;      // O.K. – operator + (z, complex (t)):
        z = t + x;      // O.K. – operator + (complex (t), x):  происходит автоматическое
                        // преобразование t к объекту класса complex и сложение с
                        // помощью описанной в классе complex операции
```

Рассмотрим *пример 3* *(как домножить на вещественное число)*

```
class complex {
        double re, im;
public:
        friend complex operator* (const complex & a, double b);
        …
};
complex operator * (const complex & a, double b) {
        complex temp (a.re * b, a.im * b);
        return temp;
int main (){
        complex x (1, 2), z;    // комплексные x и z
        double t = 7.5;         // вещественное t
        z = x * t;              // O.K. – operator * (x, t); операция * описана, где x
                        подходит по 1-му параметру, а  t по 2-му параметру.

        z = t * x;              // Err.! –т.к. нет функции преобразования x -> double, но
                                если бы была, то была бы неоднозначность: * - из double
                                или из complex
}
```

В таких случаях обычно определяют ещё одного друга с прототипом:
complex **operator *** (**double** b**, const** complex & a,).



Замечания по поводу перегрузки операций:

- n-местные операции перегружаются:

1. методом с (n-1) параметром
   Если перегружаем методом, то будет на один параметр меньше. Явно указывается только один параметр.

Например, унарный минус как метод класса: - x; // будет неявный параметр
Бинарный и унарный минус как метод класса различаются наличием параметров.

2. внешней функцией с n параметрами;

- в любом случае сохраняется приоритет, ассоциативность и местность операций;
- операции

=, [ ], ( ) и ->

можно перегрузить **только** нестатическими методами класса, что гарантирует, что первым операндом будет сам объект, к которому операция применяется.

Внешняя функция не является членом класса, всю информацию в виде параметров всех участников следует передать явным образом.

Язык С++ не позволяет менять приоритет, ассоциативность и местность операций.

Особенности перегрузки операций ++ и –

complex x;

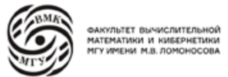
префиксная ++: ++ x; эквивалентно записи x.**operator** ++ ();

Пример как определить операцию для комплексного класса:

```
complex & operator ++ () {
        re = re + 1;
        im = im + 1;
        return *this;
}
```

постфиксная ++: x ++; эквивалентно записи x.**operator** ++ (0);

```
complex operator ++ (int) { // необходимо всегда указывать (int)
        complex c = *this;    // создаём локальный объект с, копируем в него текущий
                              // объект от имени которого вызвана операция ++
```

```
    re = re + 1;        // сам текущий объект увеличиваем его мнимую и
    im = im + 1;        // вещественную часть на 1


    return c;           // по значению возвращаем в complex
}
```

Смысл *постфиксной* операции в том, что в качестве значения возвращается старое значение объекта, а сам объект изменяется, например, увеличивается.

Компилятор семантику языка не проверяет.

Перегрузка операции → (стрелочка)

Операцию → перегружают **методом класса,** объекты которого играют роль «умных» указателей на объекты другого класса.

Операцию → можно считать постфиксной *унарной,* поскольку преобразование объекта класса в указатель не зависит от конкретного поля, на которое он указывает.

В языке C++ можно иметь доступ к методу класса через указатель: *p → f = 1;*

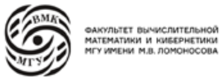
Пример,

```
struct T {int f;}
class Tptr {            // указательный класс для объекта T
    T*pt;               // возможность работы с объектом
public:
    Tptr () {pt = new T,}
    T * operator→ (){
        return pt;
}
    ~Tptr () {delete pt}    // вызов деструктора
};
```

Метод **operator→** () обязан возвращать либо указатель, либо объект класса, для которого также перегружена операция →. Последним в цепочке перегруженных операций → должен быть метод, возвращающий указатель на объект некоторого класса.

Разберём подробнее на *примере*:

```
{Tptr tpt;
tpt → f = 1;
}
```

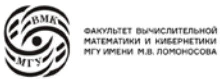



Как компилятор понимает эту конструкцию:
tpt. **operator**→() → f – результатом будет указатель на класс T. Здесь мы защитили себя

Такая перегрузка позволяет осуществлять больше контроля за указателями, сделать так, чтобы случайно не разыменовать пустой указатель, отловить ошибку с помощью умных оболочек.

Пример перегрузки операции ( ) и операции вывода <<

```
class Matrix {                    // опишем матрицу 3 на 3
        double M [3] [3];
public:
        Matrix ( ); //  операция ( ) возвращает ссылку на ту память, где располагается
                        элемент с индексами
        double & operator ( ) (int i, int j) const {
                return M [i] [j];            // возвратили ссылку на элемент матрицы
}
friend ostream & operator << (ostream & s, const Matrix & a)
{
        for (int i = 0; i< 3; i ++) {
                for (int j = 0; j< 3; j ++)
                        s << a (i, j) << ‘ ‘;     // операция << используется для вывода
                s << endl;
                }
                return s;
        }
};
```

Лекция 4. Перегрузка функций и операций

Перегрузка функций возможна только в одной области видимости (методы одного класса и т.д.). Во-первых, выбираются только те перегруженные функции, когда мы видим обращение к функции с какими-то параметрами мы должны из нескольких кандидатов выбрать подходящую.

<u>О перегрузке можно говорить только для функций из одной области видимости!</u>

Алгоритм поиска и выбора функции

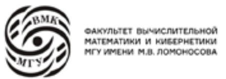
1. Выбираются только те перегруженные (одноимённые) функции, для которых фактические параметры соответствуют формальным по количеству и типу (приводятся с помощью каких-либо преобразований).
   При этом возможно, что тип формального параметра будет не совсем совпадать по типу с фактическим, но есть какие-то механизмы, позволяющие привести тип фактического параметра к типу формального. Если таких механизмов нет или количество не совпадает, то сразу функция отбор не проходит и отсеивается на первом этапе.

2. Для каждого параметра функции (отдельно и по очереди) строится множество функций, оптимально отождествляемых по этому параметру (best matching).
   На втором этапе происходит поиск наиболее подходящей функции по каждому параметру, т.е. отдельно проводим по первому параметру и выбираем победителей, которые лучше подходят по этому параметру, далее отдельно проводим по второму и т.д.
   Конкурс одного параметра называют ещё *best matching* (оптимально отождествляемый по параметру), который мы рассмотри подробнее далее.

3. Находится на пересечении этих множеств:
   - если это ровно одна функция – она и является искомой,
   - если множество пусто или содержит более одной функции, генерируется сообщение об ошибке.
   Третий этап позволяет выбрать приоритет.

Остановимся более подробно на *втором этапе*.
Разберём *пример 1*:

class X {**public**: X (**int**; ….)}; // описан класс X, где есть конструктор преобразования, умеющий по целому числу получить объект класса X



class Y {<нет конструктора с параметрами типа **int >…**}; // не умеет превращать целое в объект класса Y

Рассмотрим функции совместно используемые или перегруженные:

void f (X, **int**); //1-й параметр 'да', 2-й 'да', т.к. у класса X есть конструктор преобразование, который позволяет из 1 можно сделать X

void f (X, **double**); //1-й параметр 'да', 2-й 'нет'

void f (Y, **double**); //отбрасывается на 1-м шаге, т.к. принципиально невозможно преобразовать первый параметр **int** в класс Y

Получается, что есть два победителя по первому параметру и один по второму параметру. Далее пересечение даст нам первую функцию с параметрами X и **int.** Именно она и будет вызвана в контексте f (1, 1);

void g () {…f (1, 1); ….} //

Т.к. в пересечении множеств, построенных для каждого параметра, одна функция f (X, **int**) – вызов разрешим.

Рассмотрим ещё *пример 2*:

struct X {X (**int**); …};

void f (X, **int**); //1-й параметр 'нет', 2-й 'да',

void f (**int;** X); //1-й параметр 'да', 2-й 'нет',

void g () {…f (1, 1); ….} //

Т.к. в пересечении множеств, построенных для каждого параметра, пусто – вызов неразрешим.

Рассмотрим ещё *пример 3*:

Где нет функции **int,** которая одинаково хорошо преобразуется и к **char** и к **double:**

void f (**char**);

void f (**double**); //

void g () {…f (1); ….} // ? неоднозначность, ошибочный вызов

Не всегда просто выполнить шаг 2 алгоритма, поэтому стандартом языка C++ закреплены правила сопоставления формальных и фактических параметров при выборе одной из перегруженных функций.

Правила для шага 2 алгоритма выбора перегруженной функции

Выбираем отдельный параметр и по нему выбираем отдельную функцию среди возможных кандидатов, оставшихся после первого этапа отбора (см. стр. 40).
Если результат достигнут, то следующие шаги не рассматриваются. Если не получился шаг *a*, переходим к b и так далее до тех пор, пока не достигнут точный результат.

a) Точное отождествление
b) Отождествление при помощи расширений
c) Отождествление с помощью стандартных преобразований
d) Отождествление с помощью преобразований, определённых пользователем
e) Отождествление по …

Рассмотрим каждый пункт подробнее.

a) Точное отождествление

- точное совпадение
- совпадение с точностью до **typedef**
- тривиальные преобразования:

T [ ] < -- > T*, // массив указатель на 1-й элемент
T < -- > T&,
T -- > **const** T, // **const** можно добавить в формальном параметре в одну сторону!
T (…) <--> (T*) (…). //

Пример:

void f (**float**);	**void** g () {…	f (1.0);	// f (**double**);
void f (**double**);		f (1.0F);	// f (**int**);
void f (**int**);		f (1);	// f (**float**);
	}		….

Передавая функцию как параметр, мы записываем в качестве параметра только имя функции.

b) Отождествление при помощи расширений

- Целочисленные расширения:

char, short (**signed** b **unsigned**), **enum, bool** --> **int (unsigned int,** если не все значения могут быть представлены типом **int –** тип **unsigned short** не всегда помещается в **int**)

- Вещественное расширение: **float** -- > **double**

Пример:

```
void f (int);

void f (double)
```

```
void g () {
        short aa = 1;
        float ff = 1.0; //
        f (ff);        // f (double)
        f (aa);        // f (int)
}
```

c) Отождествление с помощью стандартных преобразований

- Все оставшиеся стандартные целочисленные и вещественные преобразования, которые могут выполняться неявно, а также преобразование объекта производного класса к объекту однозначного доступного базового класса;

- Преобразование указателей:
  - 0 -- > любой указатель,
  - любой указатель -- > **void***
  - derived* -- > base* // для однозначного и доступного базового класса, если базовый класс однозначный и доступный

Пример:

```
void f (char);
void f (double);

void g () {…f (0);      // неоднозначность, т.к. преобразование int -- > char
                        и int -- > double равноправны
}
```

d) Отождествление с помощью пользовательских преобразований

- С помощью конструкторов преобразования
- С помощью функций преобразования

Пример:

```
struct S {
        S (long);       // long -- > S
```

```
        operator int ();        // S -- > int умеет преобразовать объект класса S в целое
                                                                число
};

 void f (long);              void g (S);              void h (const S&);
 void f (char*);             void g (char*);          void h (char*);

void ex (S & a) {       // рассматриваем тело функции ex, кот. по ссылке передали
                        некоторый объект класса S
        f (a);          // O.K. f ( (long) (a.operator int ()) );     пропускаем три шага,
                                                             далее f (long)  - на шаге d)
        g (1);          // O.K.  g (S (long) 1) );                    т.е. g (S)  - на шаге d)
        g (0);          // O.K.  g ((char*) 0) );                т.е. g (char*))  - на шаге c)!
        h (1);          // O.K.  h (S (long) 1) );               т.е. h (const S&)  - на шаге d)
```

Замечание 1

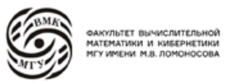
Пользовательские преобразования применяются **неявно** только в том случае, если они **однозначны!**

Пример:

```
class Boolean {
        int b;
  public:
        Boolean operator+ (Boolean);
        Boolean (int i) {b = i ! = 0;}
        operator int () {return b;}
….};
void g () {
        Boolean b (1), c (0);           // O.K.
        int k;
        c = b + 1;              // Er.,  т.к. не может интерпретироваться двояко:
                                b.operator int () + 1 – целочисленный  "+"
                                или b.operator+ Boolean (1) ) –  Boolean "+"
        k = b + 1;              // Er.!,  ---"---
```

Замечание 2

Допускается не более **одного пользовательского** преобразования для обработки одного вызова одного параметра



Пример:

```
class X {public: operator int (); …};
class Y {public: operator X (); …};

void f () {
        Y a;
        int b;
        b = a;  // Er.!,  т.к. требуется operator X (). operator int ()
}
```

Но! **явно** можно делать любые преобразования, явное преобразование сильнее неявного.
На шаге d) мы имеем право воспользоваться только одним пользовательским преобразованием и одним стандартным, которое было на шаге b) или c). При этом нельзя использовать два пользовательских или два стандартных преобразования.

e) Отождествление по …. (многоточию)

Пример 1:

```
class Real {
    public:
        Real (double);
…
};

void f (int, Real);
void f (int, …);            // можно и без ‘,’

void g () {
    f (1, 1);               // O.K. f (int, Real);
    f (1, 1 “Anna”);        // O.K. f (int, ….);

}
```

Набор макросов в С++ позволяет доставать переданные по многоточию параметры.

Пример 2:

Многоточие может приводить к неоднозначности:

```
void f (int);
void f (int …);

void g () {…
        f (1); // Er.! т.к. отождествление по первому параметру даёт обе функции
```

Одиночное наследование

Конструкторы, деструкторы и operator= не наследуются

class < имя derived-cl > : < способ наследования > < имя base-cl> {…};

Пример:

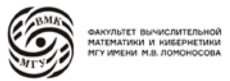
```
struct A { int x; int y;};          struct B : A { int z;}; // наследник B тип
наследования                                                  открытый
                                    struct C : protected A { int z;};    // все открытые
                                                 поля защищены (кроме наследника)
```

```
Aa;                 A *pa;
Bb;                 C c, * pc = &c;
b.x = 1;            pc -> z; // ошибка: доступ к закрытому полю
b.y = 2;            pc -> x; // ошибка: доступ к защищённому полю
b.z = 3;            pa = (A *) pc;
a = b;              pa -> x; // правильно: поле A :: x – открытое
```

```
A a; *pa;
B b; *pb;
pb = &b;
pa = pb;
pb = (B*) pa;
```

Инкапсуляция – с помощью классов, в которых можно описывать приватные члены, закрытые поля и методы.

Статический полиморфизм – осуществляется с помощью перегрузки функций/ совместного использования. Это могут быть или внешние функции перегружены или методы одного и того же класса, которые могут называться одинаково, но могут иметь разного типы параметры.



Наследование – позволяет наследовать функции.

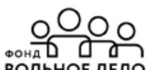
Сокрытие имён (hiding)

```
struct A {
	int f ( int x, int y);	// в классе B будет f скрыто
	int g ();
	int h;
};
struct B : public A {
	int x
	void f (int x);
	void h (int x);
};
….
A a; *pa;
B b; *pb;
pb = &b;
pa -> f (1);		// вызывается B :: f (1)
pb -> g ();		// вызывается A :: g ()
pb -> h = 1;		// Err.!   функция h – (int) не L-value выражение
pa = pb; pa -> f (1);	// Err.!   функция A :: f (1) имеет 2 параметра
pb = &a;		// Err.!   расширяющее присваивание pb = (B*)&a
pb -> f (1);		// возможно Err., если в f(1) используется x из B
```

Видимость и доступность имён

Пример:

```
int x;
void f (int a) {cout << “::f” << a <<endl;}

class A {
	int x;
public:
	void f (int a) {cout << “A::f” << a <<endl;}
};

class B: public A{
public:
```



```
        void f (int a) {cout << "B::f" << a <<endl;};
        void g ();
};
void B :: g () {
        f (1);          // вызов B::f (1)
        A :: f (1);
        :: f (1);       // вызов глобальной void f (int)
        x = 2;          // Err.! – осуществляется доступ к закрытому члену класса A
}
```

Вызов конструкторов базового и производного классов

Пример взаимодействия базового и производного классов:

```
class A {…};

class B: public A{
public:
        B ();                   // конструктор умолчания
        B (const B&);           // есть явно описанный конструктор копирования
        …
};

class C: public A{      // класс C описан открытым способом наследования класса A
public:
                // нет явно описанного конструктора копирования. Если мы ничего не
                пишем из методов в классе, то автоматически генерируется конструктор
                умолчания и копирования, операция присваивания и деструктор
        …
};
int main () {
        B b1;           // описание объекта класса B (выделяется память под объект b1 и
                        запускается конструктор: A (), далее B ()
        B b2 = b1;      // инициализация копирования A (), B (const B&)
        C c1;           // A (), C ()
        C c2 = c1;      // A (const A&); C (const C&);
        …
}
```

Если в производном классе вручную описываем конструктор копирования, то необходимо и в базовом классе вручную описывать его также. В противном случае, в базовый класс будет инициализироваться конструктором умолчания.

Классы student и student5

При наследовании появляется возможность воспользоваться ещё одним видом полиморфизма – *динамическим* полиморфизмом.
Рассмотрим на *примере 1:*

```
class student {
    char * name;
    int year;
    double est;
public:
    student (char * n, int y, double e);
    void print () const;
    ~student ();
};
```

```
class student5: public student {
  char * diplom;
  char * tutor;
public:
  student5 (char*n, double e, char*d, char*t);
  void print () const;
// эта print скрывает print из базового класса
  ~student5 ();
};
```

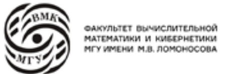
```
student5:: student5 (char*n, double e, char*d, char*t) : student (n, 5, e) {
        diplom = new char [ strten (d) + 1];
        strcpy (diplom, d);
        tutor = new char [strten (t) + 1];
        strcpy (tutor, t);
}
```

```
student5:: ~student5 () {

        delete [] diplom; // деструктор
        delete [] tutor;
}
```

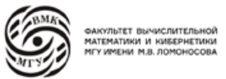
```
void student5:: print () const {
  student :: print ();  // name, year, est
        cout <<< tutor << endl:
}
```

Использование классов student и student5
Пример 2:

```
void f () {
        student s (“Kate”, 2, 4.18), * ps = & gs;       // студент
        student5 gs (“Moris”, 3.96, “DIP”, “Nick”), * pgs = & gs; // студент 5 курса

ps - > print ();            // student :: print ();  напечатает все три параметра студента Kate
```

```
pgs - > print ();        // student5 :: print ();
ps = pgs;                // base = derived – допустимо с преобразованием по умолчанию
ps - > print ();         // student5 :: print () – функция выбирается статически по типу
                                  указателя
```

Лекция 5. Динамический полиморфизм

На стр. 50 и 51 мы рассмотрели примеры на основе динамического полиморфизма.
Как известно, при открытом наследовании мы имеем право указателю на базовый класс присвоить указатель на производный класс.
Например, при этом при попытке вызвать через указатель на базовый класс функции *print* (см. пример 2 на стр.51), у нас вызовется функция print из класса *student* несмотря на то, что указатель в данный момент указывает на объект производного класса:

student5 gs (“Moris”, 3.96, “DIP”, “Nick”), * pgs = & gs; // студент 5 курса

будет вызвано, что он Moris, студент 5 курса, средний балл 3.96. При этом название работы и имя научного руководителя не будет вызвано, т.к. вызовется функция print из базового класса.
Между тем работать с объектами через указатель на базовый класс, с объектами всей иерархии наследования очень удобно. Это реализует концепцию теоретико-множественного подхода. Здесь мы тоже имеем класс студенты с подмножеством студенты 5 курса. Работа со студентами и 5-го курса и не только должна быть унифицирована. Мы пишем код и в зависимости от того, на какого студента указывает указатель вызывалась бы нужная функция print.
Для студентки Kate это должна быть функциям print обычная, а для студента Moris это должна быть функция из класса student5.

В языке С ++ есть механизм, который позволяет достичь подобного полиморфизма.
Вызов будет один и тот же: *ps - > print*, а результат будет разный и метод вызываться будет разный в зависимости от того на какой объект будет указывать указатель *ps*.
Динамическим он будет потому что в момент выполнения программы будет происходить выбор метода , а не в статике как это было при перегрузке функции (с использованием best matching).

Виртуальные методы

Метод называется виртуальным, если при его объявлении в классе используется классификатор **virtual.**
Класс называется **полиморфным,** если содержит хотя бы один виртуальный метод.
Объект полиморфного класса называют **полиморфным** объектом.
Чтобы динамически выбрать функцию *print ()* по типу объекта, на который ссылается указатель, переделываем наши классы таким образом:
давайте опишем функцию *print ()* в классе student как виртуальную (**virtual).**
Оно означает, что функция *print* будет обладать дополнительными свойствами.

```
class student {…
public:
…
        virtual void print () const;
};
class student5: public student {…
public:
…
        [virtual void] print () const;
};
```

Тогда: ps = pgs;
ps - > print (); // student5 :: print () – функция выбирается динамически по типу объекта, чей адрес в данный момент хранится в указателе

Виртуальные деструкторы

Совет: *для полиморфных классов делайте деструкторы виртуальными!*

```
void f () {
        student  * ps = new student5 (“Moris”, 3.96, “DIP”, “Nick”);
….
        delete ps;        // ps – базовый класс, вызовется ~student () и не вся память
                          очистится.
}
```

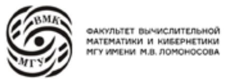
Но если:

```
        virtual ~student (); и
        [virtual] ~student5 ();
```

то вызовется ~student5 (), т.к. сработает динамический полиморфизм.

Механизм виртуальных функций (механизм динамического полиморфизма)

1. Виртуальность функции, описанной с использованием служебного слова **virtual** не проявляется в текущем классе, она начинает работать когда появляется класс, производный от данного, с функцией **с таким же прототипом**.



2. Виртуальные функции выбираются по типу объекта, на который ссылается указатель (или ссылка).

3. У виртуальных функций должны быть одинаковые прототипы. Исключение составляют функции с одинаковым именем и списком формальных параметров, у которых тип результата есть указатель или ссылка на себя (т.е. соответственно на базовый и производный класс).

4. Если виртуальные функции отличаются только типом результата (кроме случая выше), генерируется ошибка.

5. Для виртуальных функций, описанных с использованием служебного слова **virtual,** с разными прототипами работает только механизм сокрытия имён

Абстрактные классы

Абстрактным называется класс, содержащий хотя бы одну **чистую виртуальную функцию.**

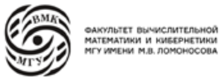
Чистая виртуальная функция имеет вид:

virtual тип_результата имя {список_формальных параметров}= 0;

Пример:

```
class shape {
public:
        virtual double area () = 0;
};
class rectangle; public shape {
        double height, width;
public:
       double area () {
                return   height * width;
        }
};
class circle: public shape {
     double radius;
public:
    double area () {
              return 3.14 * radius * radius;
  }
};
```

```
#define N 100
….
shape* p [N];
double total_area = 0;
…
for (int i = 0; i < N; i++)
     total_area +=p [i] - > area ();
….
```

Замечание: объект типа абстрактный класс создать невозможно.
Абстрактный класс необходим для того, чтобы прояснить какими операциями/свойствами будут обладать объекты-наследники этого класса.

Интерфейсы

С помощью абстрактных классов часто описывают интерфейсы.
Интерфейсами называются абстрактные классы, которые:
- не содержат нестатических полей-данных и
- все их методы являются открытыми чистыми виртуальными функциями.

Реализация виртуальных функций

```
class A{    // класс A
    int a; // поле а имеет скрытое поле pvtbl
public:
    virtual void f ();
    virtual void g (int);
    virtual void h (double);
};

class B : class A{
public:
    int b;
    void g (int);
    virtual void m (B*);
};

Тогда C c;~   a
              pvtbl
              b
              c
```

```
class C : class B{
public:
    int c;
    void h (double);
    virtual void n (C*);
};

vtbl для c~  &A::f
             &B::g
             &C::h
             &B::m
             &C::n
```

```
C c;
A* p = &c;
p - >  g (2); ~ (* (p - >  pvtbl [1]) ) (p, 2); // p = this
```

Виртуальные функции (пример 1)

На базе класса X построен классY

```
class X {
public:
    void g () {
    cout << "X::g\n";
}
    virtual void f () {
        g ();
        h ();
}
    virtual void h () {
        cout << "X::h\n";
    }
};
```

```
class Y:  public X{
public:
    void g () {
    cout << "Y::g\n";
}
    virtual void f () {
        g ();
        h ();
}
    virtual void h () {
        cout << "Y::h\n";
    }
};
```

```
int main () {
        Y b;
        X * px = &b;
        px - > f ();    // Y::g  Y::h  Y::h
        px - > g ();   // X::g  Y::h
        return 0;
}
```

Виртуальные функции (пример 2)

При использовании слова **struct** наследование считается открытым (не нужно указывать **public**).

```
struct A {
    virtual int f (int x, int y) {
        cout << "A :: f (int, int) \n";
        return x+y;
    }
};
    virtual void f (int x) {
        cout << "A :: f () \n";
    }
};
struct B : A {
    void f (int x) {
    cout << "B :: f () \n";
```

```
int main () {
        B b, *pb = &b;
        C c;
        A * pa = &b;
        pa - >  f (1);     // B :: f ();
        pa - >  f (1,2);     // A :: f (int, int);
  // pb - >  f (1,2);   // Err.! – эта f не видна

        A & ra = c;
        ra. f (1,1);         // ra – ссылка;
                             // C :: f( int, int);
        B & rb = c;
        rb. f (0, 0);   // Err.! – эта f не видна
```

```
    }
};

struct C : B {
    virtual int f (int x, int y) {
        cout << “C :: f (int, int) \n”;
        return x+y
    }
};
```

```
    return 0;
}
```

Динамический полиморфизм работает не только через указатель, но и через ссылку тоже.

Виртуальные функции (пример 3)

```
struct B {
        virtual B & f () {cout << “f () from B\n”; return *this;}
        virtual void g ((int x, int y = 7)  {cout << “f () from B:: g\n”;}
};

struct D : B {
        virtual D & f () {cout << “f () from D\n”; return *this;}
        virtual void g ((int x, int y)  {cout << “D :: g y = << y << endl;}
};
        int main () }
        D d;
        B b1, *pb = &d;
        pb - > f ();               // f () from D
                pb - > g (1);              // D :: g y=7 - возьмётся из класса B
                pb - > g (1, 2);           // D :: g y=2
return 0;
    }
```

Если значение параметра y по умолчанию будет в функции g класса D, а в базовом не будет, компилятор на вызов pb - > g (1) выдаст ошибку.

Динамический полиморфизм влияет только на то, как будет вызвана функция. На параметры он не влияет, т.к. параметры выбираются именно те, которые указаны в базовом классе. Передача параметров будет сформирована по образцу той функции, которая есть в базовом классе.

Механизм виртуальных функций позволяет использовать их как обычные, если не вызывать их не через указатели и не через ссылку на соответствующий класс.

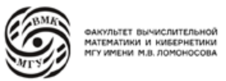
Дело в том, что косвенный вызов: надо пойти в оперативную память, взять память и только потом сделать вызов; а прямой вызов уже команда дешифрована и в процессоре адрес куда следует переходить уже есть. Прямой вызов выполняется быстрее чем косвенный. Это существенная экономия времени.

Если нам очень важна производительность, то мы отказываемся от использования виртуальных функций.

Если важно удобство и лёгкость разработки программы, удобство проектирования, производительность нас устраивает, то пользуемся виртуальными методами.

Наличие обоих методов позволяет нам выбирать между скоростью и удобством программирования.

Хотя виртуальные методы расходуют дополнительную память, но при этом разработка становится более удобной и быстрой, которая позволяет создать новый программный продукт, который можно использовать.



Лекция 6. Наследование

Способ наследования

Наследование способы / ***Зоны описания базового класса***	**public**	**protected**	**private**
Public	public	protected	private *A* ⬆
Protected	protected	protected	private **B** ⬆
Private	*недоступно*	*недоступно*	*недоступно* **C**

Таблица 6.1.Способы наследования

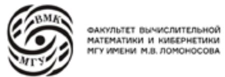
Таблица поясняет, что нам доступно, а что нет, с какими ограничениями мы сталкиваемся, в случае того или иного способа наследования.

Возможные зоны описания базового класса:
- public – что-то может быть открытым (действует по умолчанию ключевое слово struct)
- protected – что-то может быть защищённым (protected всегда нужно указывать явно)
- private – что-то может быть приватным/ закрытым (действует по умолчанию ключевое слово class).

Такими же ключевыми словами называются способы наследования: public, protected и private.
Рассмотрим, *<u>на примере</u> (см. таблицу 1) для класса B, где A – базовый класс, а B его наследник:*

Если в базовом классе A что-то было открытым/ public и способ наследования был открытым, то соответствующие методы и поля также будут открытыми для всех по отношению к классу B.

Если способ наследования protected, то те поля, которые были открытыми в базовом классе A, то становятся защищёнными в наследном классе B: т.е. он может пользоваться любыми методами и полями класса, а что касается остальных, то для всех, кроме потомка класса C, эти унаследованные поля и методы будут private/ закрытыми, т.е. контроль доступа ужесточается, он не может ослабевать).



При приватном способе наследования контроль доступа ужесточается до private, т.е. даже наследнику нельзя будет воспользоваться этими поля, которые класс B унаследовал от класса A при приватном/ private способе наследования.

Рассмотрим зону описания protected. Если что-то было защищено/ protected в классе **A**, то при открытом способе (public) наследования в классе **B** оно так и остаётся защищённым (protected). Т.к. мы не ослабляем контроль, оставляем как есть protected.
Далее способ наследования protected – остаётся таким же protected.
А для зоны описания базового класса private – ужесточается контроль до private. Т.е. то что в классе A было защищённым, в классе B становится при private способе наследования становится приватным.

Что касается приватных/ private полей и методов класса **A**, то каким бы не был способ наследования в классе **B** они недоступны (они/имена видимы в классе наследнике, но пользоваться ими нельзя).

Помимо единичного способа наследования существует множественный способ.

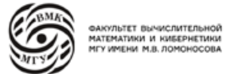
Множественное наследование

class A {….};
class B {….};
class C: **public** A **protected** B {….};
Класс C построен на базе классов C и B, при этом способы наследования обязательно нужно указывать перед каждым классом. Если что-то пропустить, то будет действовать принцип умолчания. Принцип умолчания такой же, как и при одиночном наследовании: при использовании ключевого слова **class** для описания нового класса по умолчанию
все способы private используются.
Если мы хотим что-то изменить, то для базы A пишем **public,** её наследуем открытым способом, а B защищённым способом наследования.

!! Спецификатор доступа распространяется только на один базовый класс: для других базовых классов начинает действовать принцип умолчания.

!! Класс не может появляться как непосредственно базовый класс дважды:

class C: **public** A **public** A {….}; - Err!



Но может быть более одного раза, если наследовать непрямым способом, а с помощью посредника, например, A и B являются наследниками L, а C является сразу наследником и A и B:

class L {**public:** int n; …};

class A : **public** L {…};

class B : **public** L {…};

class C : **public** A, **public** B{…**void** f (); …};

A :: L
Собственно A
B :: L
Собственно B
Собственно C

Рис. 6.2

В таблице 2 показано как устроен объект класса C.
Сначала идёт базовая часть L из класса A, затем какие-то поля, добавленные в классе A, далее вторая база идёт у которой тоже есть вхождение L отличное от первого (база L из класса B, затем какие-то поля, добавленные в классе B). Потом добавляется что-то из класса C, где многоточие мы можем описать какие-то новые поля, которые будут расположены ниже.

Если при *одиночном* наследовании мы имеем: C → B → A, то здесь **решётка смежности** такая:

L < -- A < -- C -- > B -- > L   или

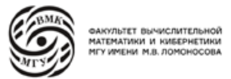
L   L
A   B
C

Рис. 6.3 Решётка смежности

При этом может возникнуть неоднозначность из-за «многократного» базового класса.

О доступе к членам производного класса

void C :: f () {…n = 5; …} // Err.! – неясно, чьё n, *но*;



необходимо указывать точно из какого класса искать n

```
void C :: f () {…A::n = 5; …} // OK.! – либо B:: = 5n;
```

Имя класса в операции разрешения видимости (A или B) – это указание, в каком классе в решётке смежности искать заданное имя.

О преобразовании указателей

Указатель на объект производного класса может быть *неявно* преобразован к указателю на объект базового класса, только если этот базовый класс является **однозначным** и **доступным.**

При множественном наследовании если мы хотим привести указатель на класс C, где C наследник на A и B одновременно к указателю на L (см. пример на стр. 60 таблица 2) , то непонятно какой указатель L: тот, который пришёл как класс A или как базовый класс B.

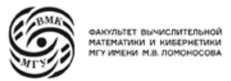
Рассмотрим продолжение предыдущего пример на стр. 60:

```
void g () {
        C * pc = new C;
        L * pi = pc;     // Err.! – L не является однозначным
        pl = (L*) pc;    // Err.! – явное преобразование не помогает,
                         // но возможно:

        pl = (L*) (A*) pc; // pl = (L*) (B*) pc; O.K.!
```

Базовый класс считается **доступным** в некоторой области видимости, если доступны его **public-**члены.

```
    class B {public: int a; …}; // a – открытое поле, наследование приватное, закрыли
                                           //доступ к классу
  class D : public L  {…};

void g () {
        D * pd = new D;
        B * pb = pd;   // Err.! – в g () public-члены B, унаследованные D, //недоступны,
                         такое преобразование может осуществлять только
                       // функция-член D, либо друзья D
}
```

Виртуальные базовые классы



```
class L {public: int n; …};
class A : virtual public L  {…};
class B : virtual public L  {…};
class C : public A, public B{…void f (); …};
```

Теперь **решётка смежности** будет такой (где L в единственном экземпляре входит в класс C):

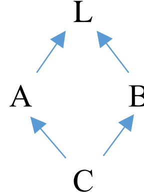


Рис. 6.4 Решётка смежности

и теперь допустимо:

```
void C :: f () {… n = 5; …} // O.K.! – n в одном экземпляре

void g () {
        C * pc = new C;
        L * pl = pc;     // O.K.! – появилась однозначность
}
```

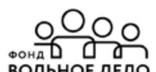
Правила выбора имён в производном классе

1 шаг:	Контроль однозначности (т.е. проверяется определено ли анализируемое имя в одном базовом классе или в нескольких); при этом <u>контекст не привлекается</u>, совместное использование (в одном из базовых классов) допускается
2 шаг:	Если однозначно определённое имя есть имя перегруженной функции, то пытаются **разрешить** анализируемый вызов (т.е. найти best-mashing)
3 шаг:	Если предыдущие шаги завершились успешно, то проводится контроль **доступа**

Неоднозначность из-за совпадающих имён в различных базовых классах

```
class A {
  public:
      int a;
      void (*b) ();
      void f ();
```

```
class B {
      int a;
      void b ();
      void h (char);
  public:
```



```
        void g (); ….
    };
```

```
        void f ();
        int g;
        void h ();
        void h (int);….
    };
```

```
class C : public A, public B {…};
```

Рассмотрим *пример:*

```
void gg (C*pc){
  pc -- > a = 1;          // Err.! – A::a или B::a
  pc -- > b ();           // Err.! – нет однозначности
  pc -- > f ();           // Err.! – нет однозначности
  pc -- > g ();           // Err.! – нет однозначности
                          // контекст не привлекается!
  pc -- > g = 1;          // Err.! – нет однозначности
                          // контекст не привлекается!
  pc -- > h ();           // O.K.!
  pc -- > h (1);          // O.K.!
  pc -- > h ('a');        // Err.! – доступ в последнюю очередь
  pc -- > A::a = 1;       // O.K.! –т.е. снимаем неоднозначность с помощью
                          операции «' '»
```

Статические члены класса

1. Статические члены-данные и члены-функции описываются в классе с квалификатором **static.**
2. Статические члены-данные существуют в одном экземпляре и доступны для всех объектов данного класса.
3. Статические члены класса существуют **независимо от конкретных экземпляров класса**, поэтому обращаться к ним можно ещё до размещения в памяти первого объекта этого класса, а также изменять, используя, например, имя константного объекта класса.
4. **Необходимо** предусмотреть выделение памяти под каждый статический член-данное класса (т.е. описать его вне класса с возможной инициализацией), т.к. при описании самого класса под его экземпляров память под статические члены-данные не выделяется.
5. Доступ к статическим членам класса (наряду с обычным способом) можно осуществлять через имя класса (без указания имени соответствующего экземпляра) и оператор разрешения области видимости «::».

Рассмотрим сравнительный пример на нашей аудитории:

Аудитория – это класс
Студенты с конспектами – это объекты
Обычные поля – это наш конспект
Доска – это наше статическое поле, она общая на вест класс. Это поле существует в единственном экземпляре. Оно не хранится внутри объекта.
Имя класса :: **static**
Пример:

```
class A{
public:
        static int x;
        static void f (char c);
};

int A::x;        // !!! – размещение статического объекта в памяти

void g () {
        …
        A::x = 10;
        ….
        A::f ('a');
        ….
}
```

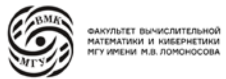
Особенности использования статических методов класса

- Статические методы класса используются в основном для работы с глобальными объектами или статическими полями данных соответствующего класса.

- Статические методы класса не могут пользоваться не статическими членами-данными класса.

- Статические методы класса не могут пользоваться указателем *this,* т.е. использовать объект от имени которого происходит обращение к функции.

- Статические методы класса не могут быть виртуальными и константными (inline – могут).



Итак, в статическом методе нет текущего объекта, он работает от имени всего класса. Информация об объекте не передаётся. Не может быть виртуальным, т.к. у виртуального метода обязательно должна быть информация о текущем объекте. У статического метода может быть параметр или параметры. Одним из этих параметров в явном виде мы можем указать объект, с которым предлагается дальше статическому методу дальше работать. Таким способом обычные поля объекта нестатические может получить доступ.

Средства обработки ошибок. Исключения в С++

Обработка **исключительных ситуаций** в С++ организуется с помощью ключевых слов: *try, catch и throw*.

Операторы программы при выполнении которых необходимо обеспечить обработку исключений выделяются в *try-catch* блок.

Если ошибка произошла внутри *try*-блока (в частности, в вызываемых из *try*-блока функциях), то соответствующее исключение должно генерироваться с помощью оператора *throw,* а перехватываться и обрабатываться в теле одного из обработчиков *catch,* которые располагаются непосредственно за *try*-блоком.
Исключение – объект некоторого типа, в частности, встроенного.

Операторы, находящиеся после места генерации ошибки в *try*-блоке, игнорируются, а после обработки исключения управление передаётся первому оператору, находящемуся за обработчиками исключений. *Try-catch* блоки могут быть вложенными.

Общий синтаксис *try-catch* блока:

```
try {
… throw исключение; ….
}
catch (type) {----/* throw; */}  //  происходит копирование объекта исключения
catch (type arg) {----/* throw; */} //
….
catch (…) {----/* throw; */}
```

Перехват исключений

С каждым try-блоком может быть связано несколько операторов **catch.** Они просматриваются по очереди сверху вниз.

Какой именно обработчик **catch** будет использоваться зависит от типа сгенерированного исключения.

Выбирается первый обработчик с типом параметра, **совпадающим** с типом исключения. **Ловушки с базовым типом** (или с указателем или ссылкой на базовый тип) **перехватывают все исключения с производным типом** (или его адресом), т.е. производные типы должны стоять раньше базовых типов.

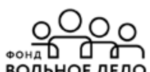
Если исключение перехвачено каким-либо обработчиком **catch,** аргумент (arg) получает его значение, которое затем можно использовать в теле обработчика. Если доступ к самому исключению не нужен, то в операторе **catch** можно указывать только его тип.

Существует специальный тип обработчика, перехватывающего любые исключения **catch** (…) {----}.
Естественно, он должен находиться в конце последовательности операторов **catch.**

Если для сгенерированного исключения в текущем **try**-блоке нет подходящего обработчика, оно перехватывается объемлющим try-блоком (main → f() →g( )→h( )).

Если же подходящего обработчика так и не удалось найти, может произойти **ненормальное (аварийное)** завершение программы. При этом вызывается стандартная библиотечная функция **terminate (),** которая в свою очередь вызывает функцию **abort** (), чего лучше избегать.
Рассмотрим *пример 1:*

```
class A {
public:
        A () {cout << * Constructor of A\n*;}
        A () {cout << * Destructor of A\n*;}
};

class Error {};           // class Error имеет место
class Error _of_A: public Error {};
void f () {
        Aa; // объект значения A
        throw 1;
cout << “This message is never printed <<;
}
int main () {
        try {
                        f ();
                        throw Error _of_A() ;
                        }
                        catch (int) {cerr << *Catch of int\n*;} // попадаем в ловушку с
                                                                // параметром int
                        catch (Error _of_A) {cerr << *Catch of Error_of_ A\n*;}
                        catch (Error) {cerr << *Catch of Error\n*;}
```



```
                    return ();
}
```

Результат программы (на основании рассмотренного выше примера 1)
Constructor of A
Destructor of A // т.к. f обработчика нет, поиск идёт дальше, но при выходе из f
// вызывается деструктор локальных объектов

Catch of int

Если поменять строки внутри try, получим:
Catch of Error_of_A

Если закомментировать строку
// **catch** (Error _of_A) {cerr << *Catch of Error_of_ A\n*;}

получим
Catch of Error

Пример использования классов исключений

```
class MathEr {…virtual void ErrProcess();…};  // базовый класс математическая ошибка
class Overflow : public MathEr {…void ErrProcess();…};
class ZeroDivide : public MathEr {…void ErrProcess();…};
…
```

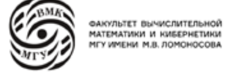
Через параметры конструктора исключения можно передавать любую нужную информацию.

Если использовать виртуальные функции, то можно после try-блока задать единственный обработчик **catch,** имеющий параметр типа базового класса, но перехватывающий и обрабатывающий любые исключения:

```
try { …
}
catch (MathEr & m) {… m. ErrProcess ();…} // ловушка со ссылкой на базовый класс
```

Организованная таким образом обработка исключений позволяет легко модифицировать программы.

Исключения, генерируемые в функциях



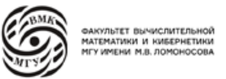
В заголовке функции можно указать типы исключений (через запятую), которые может генерировать функция (эту возможность удобно использовать при описании библиотечных функций):

Тип_рез имя_функции (список_аргум) ***[const] throw*** *(список_типов){…}*

Если список типов **пустой**, то функция не может генерировать **никаких** исключений.
Если же функция все-таки сгенерировала не декларированное исключение, вызывается библиотечная функция **unexpected (),** работающая аналогично функции **terminate ().**

Использование аппарата исключений – единственный безопасный способ нейтрализовать ошибки в конструкторах и деструкторах, поскольку они не возвращают никакого значения, и нет другой возможности отследить результат их работы.

Если деструктор, вызванный во время свёртки стека, попытается завершить свою работу при помощи исключения, то система вызовет функцию **terminate (),** что крайне нежелательно. Отсюда важное требование к деструктору: ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы.



Лекция 7. Динамическая идентификация типа

Рассмотрим на *примере*:
Если у нас есть класс A базовый и его наследником является класс B (наследование открытым способом), то:

A	Aa; A * pa = 8a; // где 8 – означает присвоить
⬆	
B	Bb; B * pb = 8b;
	pa = pb // Err! pb = pa – Указателю на производную нельзя явно присвоить указатель на базовый класс

Механизм RTTI (Run-Time Type Identification)

Механизм динамической идентификации типа (PTTI) позволяет во время исполнения узнать и безопасным образом преобразовать указатели, если подходящий объект имеется в качестве указываемого.
Операция type_id позволяет идентифицировать точный тип при наличии указателя на полиморфный класс.
Полиморфный класс – это класс, когда есть хотя бы одна полиморфная функция.

Механизм RTTI состоит из трёх частей:

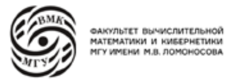
1. Операция **dynamic_cast –** в основном предназначена для получения указателя на объект производного класса при наличии указателя на объект полиморфного базового класса;

2. Операция **typied –** служит для идентификации точного типа объекта при наличии указателя на полиморфный базовый класс;

3. Структура **type_info –** позволяет получить дополнительную информацию, ассоциируемую с типом.

Для использования RTTI в программу следует включить заголовок <typeinfo>. Без него все эти операции недоступны.

Остановимся подробнее на каждой из трёх частей механизма RTTI.



(1) Операция **dynamic_cast** реализует приведение типов (*указателей или ссылок*) полиморфных классов в динамическом режиме.
Синтаксис использования *dynamic_cast:*

dynamic_cast < целевой тип> (выражение)

Если даны **два полиморфных класса B и D** (причём D – производный от B), то *dynamic_cast* всегда может привести D* к B*.

Также *dynamic_cast* может привести D* к B*, но только в том случае, если объект, на который указывает указатель, действительно является объектом типа D (либо производным от него)!

При неудачной попытке привидения типов результатом выполнения *dynamic_cast* является 0, если в операции использовались указатели.
Если же в операции использовались ссылки, генерируется исключение типа **bad_cast.**

Рассмотрим пример:
Пусть Base – полиморфный класс, а Derived – класс, производный от Base.

```
Base * bp, b_ob;
Derived * dp, d_ob;

bp = &d_ob;
dp = dynamic_cast < Derived* > (bp);
if (dp)
        cout << «Приведения типов не произошло»;
```

(2) – (3) Информацию о типе объекта можно получить с помощью операции ***typeid.***
Синтаксис использования операции ***typeid:***

typeid (выражение) или
typeid (имя_типа)

Операция ***typeid*** возвращает ссылку **на объект класса type_info,** представляющий либо тип объекта, обозначенного заданным выражением, либо непосредственно заданный тип.

В классе *type_info* определены следующие открытые члены:
bool operator = = (**const** type_info & объект); // для сравнения типов

bool operator ! = (**const** type_info & объект); // для сравнения типов
bool before (const type_info & объект); // для внутреннего использования
const char * name ( ); // возвращает указатель на имя типа

Оператор ***typeid*** наиболее полезен, если в качестве аргумента задать указатель полиморфного базового класса, т.к. с его помощью во время выполнения программы можно определить тип реального объекта, на который он указывает. Тоже относится и к ссылкам.

typeid часто применяется к разыменованным указателям (*typeid* (*p)). Если указатель на полиморфный класс p == NULL, то будет сгенерировано исключение типа ***bad_typed.***

Пример:

```
class Base {
        virtual void f ( ) {…};
};
class Derived1: public Base {…
class Derived2: public Base {…
};
int main ( ) {
        int i;
        Base *p, b_ob;
        Derived1 ob1;
        Derived2 ob2;
        cout << «Тип i - » << typeid ( i ) , name () << endl;
        p = & b_ ob;
        cout << "p указывает на объект типа" << typeid ( * p ) , name ( ) << endl;
        p = & ob1;
        cout << "p указывает на объект типа" << typeid ( * p ) , name ( ) << endl;
        p = & ob2;
        cout << "p указывает на объект типа" << typeid ( * p ) , name ( ) << endl;
        if (typeid (ob1) = = typeid (ob2))
            cout << "Тип объектов ob1 и ob2 одинаков\n";
        else
            cout << "Тип объектов ob1 и ob2 не одинаков\n";
        return ( );
}
```

Стандартные исключения

Текст по генерации стандартных исключений вставляется компилятором. Стандартные исключения объединены в иерархию классов, в вершине которой находится стандартный абстрактный библиотечный класс ***exception,*** описанный в <stdexcept> :

```
class exception {
public :
        exception ( ) throw ();
        exception (exception &) throw ();
        exception const & operator = (const exception &) throw ();
        virtual ~exception () throw ();
        virtual const char * what () const throw ();
        ….
};
```

Иерархия классов стандартных исключений

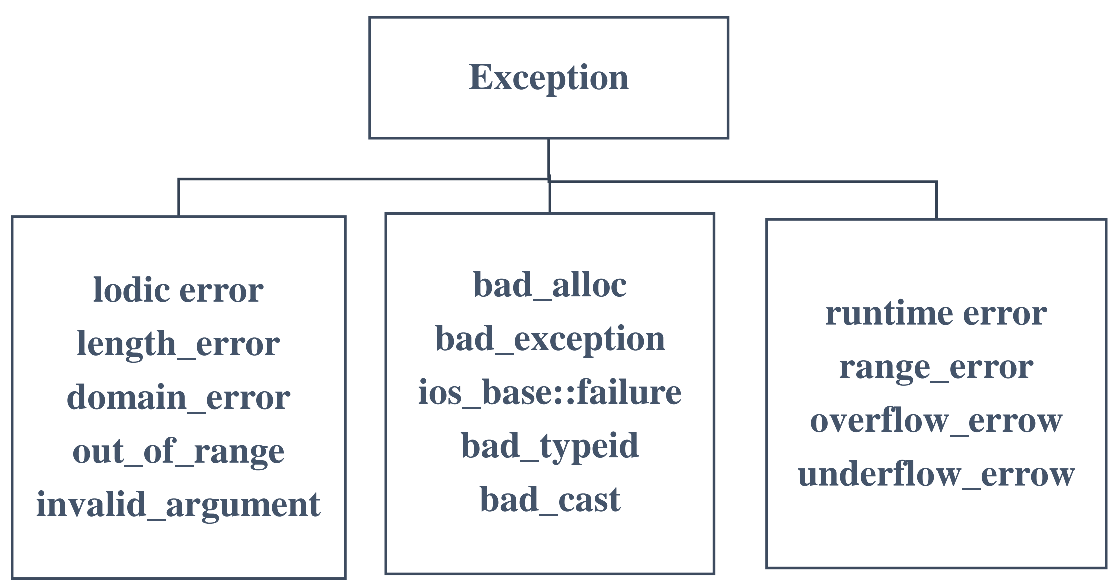


Схема 7.1 Иерархия классов стандартных исключений

Из этих классов исключений мы рассматриваем только исключения:

bad_cast и **bad_typeid,** генерируемые соответственно при неверной работе операций dynamic_cast и typeid, и расположенные в файле <typeinfo>,

out_of_range, генерируемое методом at () контейнеров STL, и расположенное в файле <stdexcept>,
bad_alloc, генерируемое операцией **new** при невозможности выделения динамической памяти и расположенное в файле <new>.

Чтобы операция **new** при ошибке выделения динамической памяти возвращала 0, надо использовать следующую её форму:

```
T * p = new (nothrow) T;
```

Пример использования стандартных исключений

```
void f () {
    try {….
        // использование стандартной библиотеки
    catch (exception & e) {
        cout << "Стандартное исключение" << e.what () <<'\n'; // e.what
                                                  // виртуальный метод
    }
    catch (…) {               // ловушка с многоточием всегда ставим в конце
        cout << "Другое исключение" <<'\n';
        …
    }
}
```

Иерархию классов стандартной библиотеки можно брать за основу для своих исключений.

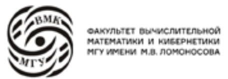
Шаблоны

1. Механизм шаблонов реализует в C++ **параметрический полиморфизм.**

2. Шаблон представляет собой предварительное описание функции или класса, конкретное представление которых зависит от параметров шаблона.

3. Для описания шаблонов используется ключевое слово **template,** вслед за которым указываются аргументы (параметры шаблона), заключённые в угловые скобки.

4. Параметры шаблона перечисляются через запятую, и могут быть:
   a) объектами следующих типов:
      - целочисленного
      - перечислимого



- указательного (в т.ч. указатели на члены класса)
- ссылочного;

b) именами типов (перед именем) типа надо указывать ключевое слово **class** или **typename**).

5. Параметры-объекты являются **константами**, их нельзя изменять внутри шаблона.

Шаблоны функций

template < список_параметров_шаблона>
тип_результата имя_функции (список_аргументов_функции) {/*….*/}

Обращение к функции-шаблону: имя_функции<список_фактич._пар-ров_шаблона> (список_фактич._аргументов_функции)

Пример:

```
template <class T>          // функция суммирования элементов массива
T sum (T array [ ], int size) {
        T res = 0;       // где res - накопитель
        for (int i = 0; I <size; i++>) res += array [ i ];
        return res;
}
```

Использование шаблона для массивов типа **int** [10]:

```
int iarray [10];
int i_sum;
// ….
i_sum = sum <int> (iarray, 10);
```

Можно задать **size** в виде параметров шаблона:

```
template <class T, int size>
T sum (T array [ ]){/*…*/}
```

Тогда вызов sum будет таким:

```
i_sum = sum <int, 10> (iarray);
```

В случае шаблонных функций компилятор иногда может угадать на какой тип надо настроить шаблон и сгенерировать соответствующую этому типу функцию, т.е. необязательно всегда указывать в угловых скобках < > параметры шаблона.

Неявное определение параметра-типа шаблона

Пример 1

```
class complex
```

```
{…. public:
        complex (double r = 0, double i = 0);
        operator double ();  …..
};
template <class T>
T f ( T& x, T& y) {
    return x > y ?  x : y;
}
double f (double x, double y){
    return x > y ?  -x : -y;                /
}
int main ( ) {
    complex a (2, 5), b (2, 7), c;
    double x  = 3.5, y = 1.1;
    int i, j = 8, k = 10;

    c = f (a, b);     // f <complex> (a, b)
    x = f (a, y);     // f (a, y)
    i = f (j, k);     // f <int> (j, k)
    return 0;

}
```

Пример 2

```
template <class T>
T max ( T& x, T& y) {
    return x > y ?  x : y;
}

int main ( ) {
double x  = 1.5, y = 2.8; z;
int i = 5, j = 12, k;
char * s1 = "abft";
char * s2 = "abxde"; * s3;

z = max (x, y);                  // max <double>
k = max <int> (i, j);            // max <int>
// z = max (x, i);        // Err.! – неоднозначный выбор параметров
z = max <double> (y, j);
s3 = max (s2, s1);          // max <char*>, но происходит сравнение
                            адресов
```

```
        return 0;

        }
```

Пример 3

```
template <class T> T m1 (T a, T b) {
        cout << *m1_1\n*;
        return a < b ?  b : a;
}

template <class T, class  B> T m1 (T a, T b, B c) {
        cout << *m1_2\n*;
        c = 0;   return a < b ?  b : a;
}

int m1 (int a, int b) {
        cout << *m1_4\n*;
        return a < b ?  b : a;
}

int m1 (int a, double b) {
        cout << *m1_5\n*;
        return a;
}

template <class T, class  Z > T m1 (T a, Z b) {
        return a < b ?  b : a;
}

int main ( ) {                                  // Если убрать первый шаблон:
        int i;
        m1 < int > (2, 3);              // m1_1         //m1_3
        m1 < int, int > (2, 3);         // m1_3         //m1_3
        m1 < int > (2, 3, i);           // m1_2         //m1_2
        m1  (1, 1);                     // m1_4         //m1_4
        m1  (1.3, 1);                   // m1_3         //m1_3
        m1  (1.3, 1.3);                 // m1_1         //m1_3
        return 0;
}
```

Алгоритм выбора оптимально отождествляемой функции с учётом шаблонов

- Для каждого шаблона, подходящего по набору формальных параметров, осуществляется формирование специализации, соответствующей списку фактических параметров.
- Если есть два шаблона функции и один из них более специализирован (т.е. каждый его допустимый набор фактических параметров также соответствует и второй специализации), то далее рассматривается только он.
- Осуществляется поиск оптимально отождествляемой функции из полученного набора функций, включая определения обычных функций, подходящие по количеству параметров. При этом если параметры некоторого шаблона

функции были определены путём **выведения** по типам фактических параметров вызова функции, то при дальнейшем поиске оптимально отождествляемой функции к параметрам данной специализации шаблона нельзя применять никаких описанных выше преобразований, кроме преобразований **точного** отождествления.

- Если обычная функция и специализация подходят хорошо, то выбирается обычная функция.
- Если полученное множество подходящих вариантов состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

Шаблоны классов

Шаблоны создаются для классов, имеющих общую логику работы.
Для определения шаблона класса перед ключевым классом **class** помещается **template-** классификатор.

template < список_параметров_шаблона_типа > **class** имя_класса {/*….*/};

Конкретный экземпляр шаблона класса (объект класса) можно создать так:

имя_класса < список_фактических_параметров > объект;

Для шаблонов класса никакие фактические параметры по умолчанию не выводятся.

Функции-члены класса-шаблона автоматически становятся функциями-шаблонами.

Шаблоны методов

Можно описывать шаблонные методы в классах, не являющихся шаблонами.

Запрещено определять шаблоны для виртуальных методов, из-за возникающих больших накладных расходов на возможную перестройку таблиц виртуальных методов при компиляции.

Шаблонный класс stack

```
template <class T, int max_size >
	class stack {
	T s [max_size];
	int top;
public:
	stack ( ) {top = 0;}
	void reset ( ) {top = 0;}
```

```
        void push (T i);
        T pop ( );
        bool is_empty () {return top == 0;}
        bool is_full () {return top == max_size;}
};
```

```
template <class T, int max_size >
void stack <T, max_size> : : push (T i){
    if (! is_full () ){
        s [top] = i;
        top++;
    }
    else
        throw "stack_is_full";
}
```

```
template <class T, int max_size >
T stack <T, max_size> : : pop ( ) {
    if (! Is_empty () ){
    top --;
    return s [top];
    }
    else
        throw "stack_is_empty";
}
```

Недостаток такого метода: скомпилированная программа растёт по объёму.

Виды отношений между классами

Часто при проектировании программ, разрабатываемых в объектно-ориентированном стиле, взаимосвязь используемых в них классов и объектов представляют в виде UML (Unified Modeling Language).

Классы изображают в виде прямоугольника, состоящего из трёх частей:
- сверху – имя класса,
- в середине – члены-данные, возможно, с указанием типов,
- внизу – прототипы методов класса.

Имена абстрактных классов и чистых виртуальных функций выделяются курсивом.

Перед описанием имени членов классов или метода можно указать спецификатор доступа с помощью значков:
+ (**public**),
- (**private)** или
(**protected**).

Для статических членов класса после спецификатора доступа указывается символ $.

Большинство объектно-ориентированных языков программирования (ООЯП) поддерживают следующие отношения между классами:

- Ассоциация
- Наследование
- Агрегация
- Использование
- Инстанцирование

Рассмотрим подробнее каждый класс.

1. Ассоциация.

Ассоциация – отношение, показывающее, что два класса концептуально взаимодействуют друг с другом.

Отношение ассоциации удобно представлять в виде ER-диаграмм (entry – relationships – сущность – связь) в основном используемым при разработке реляционных баз данных. Связи отображаются сплошными линиями без направления.

Виды связей, представляемых ER-диаграммами:

1 — 1
1 — N
N — N

Различают обязательное и необязательное участие сущностей в установленных между ними связях.

Примеры:

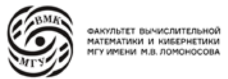
- Система автоматизации работы магазина (количество покупателей в магазине может быть от 0 до N и они могут купить товаров от 0 до N). Существует естественный предел сколько может быть в магазине одновременно покупателей и сколько они могут приобрести товаров.

покупатель 0…N ----- 0…N товары

- Минимум вершин, которое может быть в многоугольнике – три. Диапазон от 3 до N.

многоугольник 1 ------ 3… N вершина

Рис.7.2 Примеры



2. Наследование.

Наследование открытым способом позволяет реализовать отношения Часть – Общее (“is a”).

Отношение задаётся в виде стрелки с не закрашенным треугольником на конце, которая указывает на базовый класс.

Примеры:

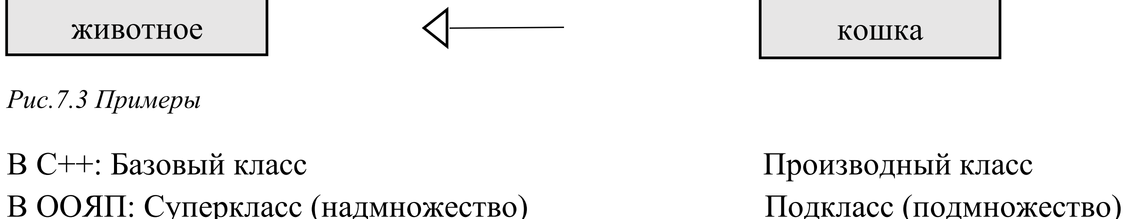


Рис.7.3 Примеры

В C++: Базовый класс — Производный класс
В ООЯП: Суперкласс (надмножество) — Подкласс (подмножество)

Агрегация.

Агрегация включает отношения Часть – целое (“has a”). Бывает строгая и нестрогая.

Строгая агрегация называется – **композиция**, обозначается с закрашенным ромбиком на конце стрелки.

Нестрогая агрегация – **агрегация** (при этом один объект может быть включён в разные объекты одновременно).

Композиция обозначается стрелкой с закрашенным ромбом на конце, направленной на включающий класс.

Примеры:

Строгая агрегация

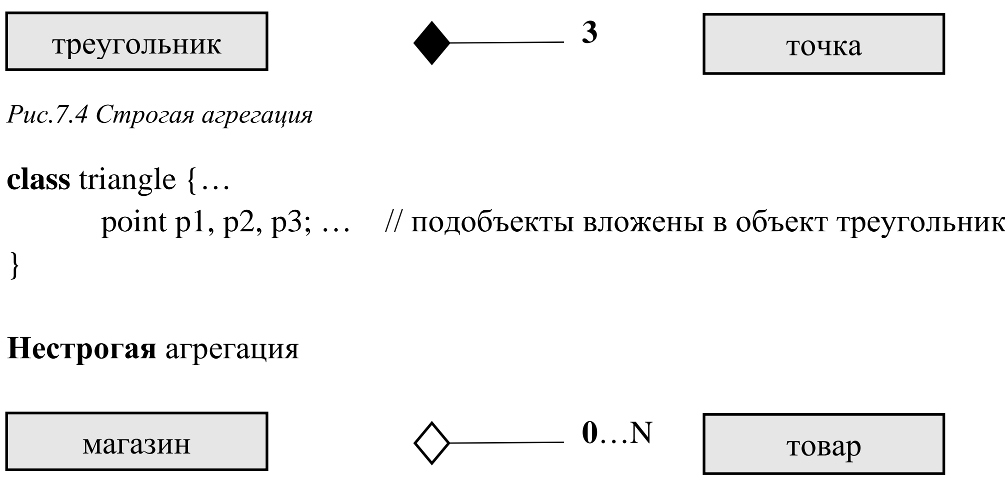


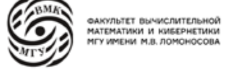
Рис.7.4 Строгая агрегация

```
class triangle {…
        point p1, p2, p3; …    // подобъекты вложены в объект треугольник
}
```

Нестрогая агрегация

магазин — 0…N — товар

Рис.7.5 Нестрогая агрегация



```
class shop {…
        goods * g; …
}
```

3. **Использование и Инстанцирование.**

Отношение **использования** возникает, когда

- в прототипе метода одного класса используется имя другого класса;
- в теле метода одного класса – локальный объект другого класса;
- в теле метода одного класса вызывается функция другого класса.

Использующий класс называют client, а используемый *supplier*.
Отношения использования обозначается пунктирной стрелкой, указывающей на класс *supplier*.
Примеры:

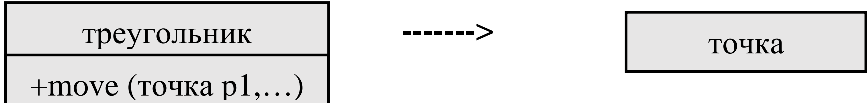


Рис.7.6 Отношения использования

Инстанцирование – связь между шаблоном класса и классом - результатом генерации по шаблону.

В UML инстанцирование обозначается стрелкой, идущей от шаблона класса к конкретной его реализации.

Стандартная библиотека шаблонов STL

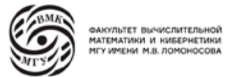
STL (Standart Template Library) является частью стандарта C++.
Ядро STL состоит из четырёх основных компонентов:

- контейнеры,
- итераторы,
- алгоритмы,
- распределители памяти.

Рассмотрим подробнее каждый из вышеперечисленных компонентов.

1. **Контейнер –** это класс, который предназначен для хранения объектов какого-либо типа.

   Примеры известных ранее контейнеров:
   - таблица идентификаторов,
   - массив,
   - дерево,
   - список,



- ассоциативный список, например, список, хранящий фамилии людей и номера их телефонов, ключом которого является фамилия, если она уникальна.

Стандартные контейнеры STL

vector <T>	-	динамический массив
list <T>	-	линейный список
stack <T>	-	стек
queue <T>	-	очередь
deque <T>	-	двусторонняя очередь
priority_queue <T>	-	очередь с приоритетами
set <T>	-	множество
bitset <T>	-	множество битов (массив из N бит)
multiset <T>	-	набор элементов, возможно, одинаковый
map <key, val>	-	ассоциативный список
multimap < key, val >	-	ассоциативный список для хранения пар ключ/значение, где с каждым ключом может быть связано более одного значения

!! Важно знать, как можно обращаться с этими контейнерами.

Состав контейнеров

В каждом классе-контейнере определён набор методов для работы с контейнером, причём все контейнеры поддерживают **стандартный набор базовых операций.**

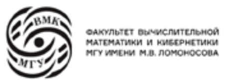
Базовая операция контейнера (базовый метод) – метод класса, имеющий во всех контейнерах одинаковое имя, одинаковый прототип и семантику (их примерно 15-20).

Например,
функция **push-back ()** помещает элемент в конец контейнера,
функция **size ()** выдаёт текущий размер контейнера.

Базовыми методами можно пользоваться одинаково независимо от того, в каком конкретно контейнере находятся элементы, можно также менять контейнеры, не меняя тела функций, работающих с ними.

Операции, которые не могут быть эффективно реализованы для всех контейнеров, не включаются в набор общих операций.

Например, обращение по индексу введено для контейнера **vector**, но не для **list.**



Типы, используемые в контейнерах

Каждый контейнер в своей **public** части содержит серию **typedef**, где введены стандартные имена типов, например:

value_type	-	тип элемента
allocator_type	-	тип распределителя памяти
size_type	-	тип, используемый для индексации
iterator	-	итератор
const_ iterator	-	константный итератор
reverse_ iterator	-	обратный итератор
const_ reverse_ iterator	-	обратный константный итератор
pointer	-	указатель на элемент
const_ pointer	-	указатель константный на элемент
reference	-	ссылка на элемент
const_ reference	-	ссылка на константный элемент

Эти имена определяются внутри каждого контейнера так, как это необходимо, т.е. скрывают реальные типы, что, например, позволяет писать программы с использованием контейнеров, ничего не зная о реальных типах.

В частности, можно составить код, который будет работать с любым контейнером.

Распределители памяти

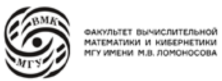
Каждый контейнер имеет **распределитель памяти (allocator),** который используется при выделении памяти под элементы контейнера и предназначен для того, чтобы освободить пользователей контейнеров, от подробностей физической организации памяти.

Стандартная библиотека обеспечивает стандартный распределитель памяти, заданный стандартным шаблоном **allocator** из заголовочного файла **<memory>**, который выделяет память при помощи операции **new ()** и по умолчанию используется всеми стандартными контейнерами.

Класс **allocator** обеспечивает стандартные способы выделения и перераспределения памяти, а также стандартные имена типов для указателей и ссылок.

Пользователь может задать свои распределители памяти, предоставляющие альтернативный доступ к памяти.

Стандартные контейнеры и алгоритмы получают память и обращаются к ней через средства, обеспечиваемые распределителем памяти.



Класс allocator

```
template <class T> class allocator {
public:
        typedef T * pointer;
        typedef T & reference;
        …..
        allocator ( ) throw ( );
        …..
        pointer allocator ( size_type n);        // выделяет память для n объектов типа T
        void deallocate (pointer p, size_type n);     // освобождает память, отведённую под
                                                      // n объектов типа T
        void construct (pointer p, const T & val);    // инициализирует * p значением val
        void destroy (pointer p);                     // вызывает деструктор * p, память при
                                                      // этом не освобождается
```

Итераторы

Итератор – это класс, объекты которого выполняют такую же роль по отношению к контейнеру, как указатели по отношению к массиву. Указатель может использоваться в качестве средства доступа к элементам массива, а итератор – в качестве средства доступа к элементам контейнера.

Итераторы «склеивают» ядро STL в одну библиотеку.
Итераторы поддерживают абстрактную модель данных как последовательности объектов.

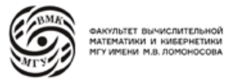
Понятие «нулевой итератор» не существует, а при организации циклов происходит сравнение с концом последовательности.

Каждый контейнер обеспечивает свои итераторы, поддерживающие стандартный набор итерационных операций со стандартными именами и смыслами.

Итераторные классы и функции находятся в заголовочном файле **<iterator>.**

Методы контейнеров для нахождения значений итераторов концов последовательности элементов

- **iterator begin ();** - возвращает итератор, который указывает на первый элемент последовательности
- **const_iterator begin () const;**
- **iterator end ();** - возвращает итератор, который указывает на элемент, следующий за последним элементом последовательности
- **const_iterator end () const;**
- **reverse_iterator rbegin (); -** возвращает итератор, который указывает на первый элемент в обратной последовательности



- **const_ reverse_iterator rbegin () const;**
- **reverse_iterator rend ();**
- **const_ reverse_iterator rend () const;**

Прямые итераторы

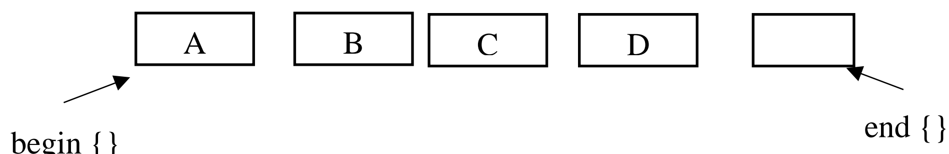


Рис.7.7 Прямые итераторы

На рисунке 7.7 стр.87 есть контейнер, в котором хранятся значения A, B, C, D, где begin укажет на начальный, end на следующий за последним.

Обратные итераторы

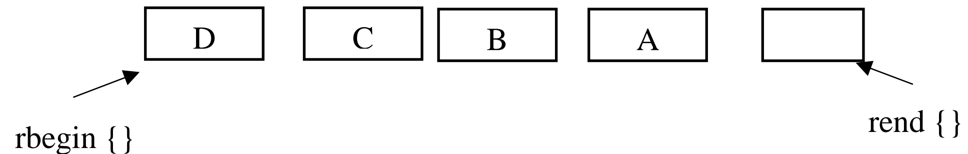


Рис.7.8 Обратные итераторы

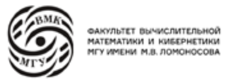
На рисунке 7.8 стр.87 есть контейнер, в котором хранятся значения, где rbegin укажет на D (переворачиваем последовательность), rend на A, следующий за последним в перевёрнутой последовательности этих элементов.

Операции над итераторами

Пусть p – объект типа итератор.
К каждому итератору можно применить, как минимум, три ключевые операции:

- * p – на который указывает итератор
- p++ – переход к следующему элементу последовательности
- == – операция сравнения.

Пример:
iterator p = v. begin (); – такое присваивание верно независимо от того, какой контейнер v.



Теперь p – первый элемент контейнера v.

Замечание:

при проходе последовательности как прямым, так и обратным итератором переход к следующему элементу будет p++ (а не p –!).

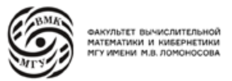
Не все виды итераторов поддерживают один и тот же набор операций.

Категории итераторов

В библиотеке STL5 категорий итераторов:

1. **Вывода** (output – запись в контейнер)
   (*p =, ++)
2. **Ввода** (intput – считывается из контейнера)
   (= *p, →, ++, ==, !=)
3. **Однонаправленный** (forward)
   (* p =, = * , →, ++, ==, !=)
4. **Двунаправленный** (bidirectional)
   (* p =, = *p, →, ++, ==, !=) – list, map, set
5. **С производным доступом** (random, access)
   (* p =, = *p, →, ++, ==, !=, [], +, -, +=, - =,<, >, <=, >=) – vector, deque

Каждая последующая категория является более мощной, чем предыдущая.



Лекция 8. Алгоритмы

Алгоритмы STL (их всего 60) – реализуют некоторые распространённые операции с контейнерами, которые не представлены функциями-членами каждого из контейнеров (например, просмотр, сортировка, поиск, удаление элементов…).

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций.

Операции, реализуемые алгоритмами, являются универсальными для любого из контейнеров и поэтому определены вне этих контейнеров.

Реализация алгоритмов не использует имён никаких конкретных контейнеров, а все действия над контейнером производятся через универсальные имена итераторов.

Зная, как устроены алгоритмы, можно писать свои собственные алгоритмы обработки, которые не будут зависеть от контейнера.

Все стандартные алгоритмы находятся в пространстве имён std, а их объявления – в заголовочном файле **<algorithm>.**

Группы алгоритмов

1. **Немодифицирующие алгоритмы –** извлекают информацию из контейнера, но не модифицируют сам контейнер (ни элементы, ни порядок их расположения).

   Примеры:
   Find () – находит первое вхождение элемента с заданным значением
   Count () – количество вхождений элемента с заданным значением
   For_each () – применяется некоторая операция к каждому элементу (не связано с изменениями)

2. **Модифицирующие алгоритмы –** изменяют содержимое контейнера. Либо меняются сами элементы, либо их порядок, либо их количество.

   Примеры:
   Transform () – применяется некоторая операция к каждому элементу (каждый элемент изменяется)
   Reverse () – переставляет элементы в последовательности
   Copy () – создаёт новый контейнер

3. **Сортировка.**

   Примеры:
   Sort () – простая сортировка
   Stable_sort – сохраняет порядок следования одинаковых элементов (например, это бывает существенно при сортировке по нескольким ключам)
   Merge – склеивает две отсортированные последовательности.

Категории итераторов и алгоритмы

По соглашению, при описании алгоритмов, входящих в STL, используются стандартные имена формальных-параметров-итераторов.

В зависимости от названия итератора в прототипе алгоритма должен использоваться уровня «не ниже чем». Т.е. по названию параметров шаблона можно понять какого рода итератор нам нужен, то есть к какому контейнеру применим этот алгоритм.

Пример:

Шаблонная функция **find () с тремя параметрами** (итератор, с какого начинается поиск, каким заканчивается и искомый элемент).

Для реализации целей функции достаточно итератора ввода (из контейнера).

```
Template <class InputInterator, class T>
InputInterator  find (InputInterator first,  InputInterator last, const T& value) {
                while (first != last && * first ! = value)
                first ++;
                return first;
}
```

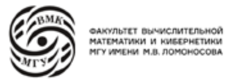
Однако, категория итераторов не принимает участия в вычислениях. Этот механизм относится исключительно к компиляции.

Typename

1. Ключевое слово **typename** используется при описании параметра-типа шаблона наряду с ключевым словом **class.**

Например,

```
template <template T>
void f (T a) { ...}
```

2. Если используемое в шаблоне имя типа зависит от параметров шаблона, необходимо использовать ключевое слово **typename.**

Например,

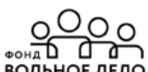
```
template <class T>
void f (vector <T> & v)
{
        vector <T> :: iterator i = begin ();                // Err.!
        typename vector <T> :: iterator i = begin ();       // O.K.
        …..
}
```

Пример шаблонной функции, использующей тип, вложенный в класс-параметр шаблона

```
struct X {
    enum {e1, e2, e3} g;
    struct inner {
                int i, j;
        void g ( ) {cout << "ggg\n";}
    };
    inner c;
};
    template <class T>
    void f (typename T:: inner  t) {t.g ();}

    int main (){
        Xx;
        x.g = X ::e1;
        x.c.g ();
        X:: inner   iii;
        iii.i = 7;
        f <X> (iii);
        return ();
}
```

Пример шаблонной функции для контейнеров STL

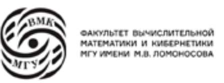
Поиск заданного элемента в контейнере, начиная с последнего (просмотр контейнера от конца к началу) обычно производится так:



```
template <class C>
 typename C :: const_iterator  find_last
        (const C & c, template C :: value_type v)
{
                typename C :: const_iterator  p = c.end ()
                while (p !=c.begin () )
                        if (* -- p == v)
                                return p;
}
```

Контейнер vector

```
template <class T, class A = allocator <T> > class vector {
…..
public:
// Типы – typedef ….. – см.выше
// Итераторы ….. – см.выше
//
// Доступ к элементам

reference operator [ ] (size_type n); // доступ без проверки диапазона
const_reference operator [ ] (size_type n) const;

reference at (size_type n); // доступ с проверкой диапазона (если индекс выходит за пределы диапазона, возбуждается исключение out_of_range)
const_reference at (size_type n) const;

reference front ( ); // первый элемент вектора
const_reference front ( ) const;

reference back (); // последний элемент вектора
const_reference front ( ) const;

// Конструкторы, деструкторы, operator=

explicit vector (const A &=A ()); // создаётся вектор нулевой длины

explicit vector(size_type n; const T & value = T (); const A & = A ());
// создаётся вектор из n элементов со значением value  или с «с нулями» типа T, если второй параметр отсутствует; в этом случае конструктор умолчания в классе T обязателен)
```

template <class I> vector (I first, I last, const A& = A ());
// инициализация вектора копированием элементов из [first, last), I – итератор для чтения

vector (const vector <T, A> & obj); // конструктор копирования

vector & operator = (const vector <T, A> & obj);

~ vector ();

// Некоторые функции-члены класса vector

iterator erase (iterator i) - удаляет элемент, на который указывает данный итератор. Возвращает итератор элемента, следующего за удалённым.

iterator erase (iterator st, iterator fin); – удалению подлежат все элементы между **st** и **fin,** но **fin** не удаляется**.** Возвращает **fin.**

iterator insert (iterator i, const T & value = T ()); – вставка некоторого значения value перед **i.** Возвращает итератор вставленного элемента.

void insert (iterator i, size_type n, const T & value = T ()); – вставка **n** копий элементов со значением **value** перед **i.**

void push back (const T & value); – добавляет элемент в конец вектора.

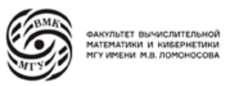
size_type size () const; – выдаёт количество элементов вектора.

bool empty () const; – возвращает истину, если вызывающий элемент пуст.

void clear ()– удаляет все элементов вектора.

Контейнер list

template <class T, class A = allocator <T> > class list{
…….
public:
// **Типы**
// …..
// **Итераторы**
// …..



//
// **Доступ к элементам**
//
reference front ();   // первый элемент списка

const_reference front ( ) const;

reference back ();   **//** последний элемент списка

const_reference back ( ) const;
// Конструкторы, деструкторы, operator=

explicit list (const A &=A ()); // создаётся список нулевой длины

explicit list (size_type n; const T & value = **T (); const A & = A ());**
// создаётся список из n элементов со значением **value** или с «с нулями» типа T, если второй параметр отсутствует

template <class I> list (I first, I last, const A& = A ());
// инициализация списка копированием элементов из [first, last), I – итератор для чтения

list (**const list <T, A> & obj);** // конструктор копирования

list & operator = (**const vector <T, A> & obj);**

~ list ();

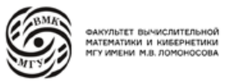
// Некоторые функции-члены класса list

iterator erase (iterator i); - удаляет элемент, на который указывает данный итератор. Возвращает итератор элемента, следующего за удалённым.

iterator erase (iterator st, iterator fin); – удалению подлежат все элементы между **st** и **fin,** но **fin** не удаляется**.** Возвращает **fin.**

iterator insert (iterator i, const T & value = **T ()); –** вставка некоторого значения value перед **i.** Возвращает итератор вставленного элемента.

void insert (iterator i, size_type n, const T & value = **T ()); –** вставка **n** копий элементов со значением **value** перед **i.**



void push_back (const T & value); – добавляет элемент в конец списка.

void push_front (const T & value); – добавляет элемент в начало списка.

void pop_front ( ); – удаляет первый элемент списка.

void pop_back ( ); – удаляет последний элемент списка (не возвращает значение!).

size_type size ( ) const; – выдаёт количество элементов списка.

bool empty ( ) const; – возвращает истину, если вызывающий список пуст.

void clear (); – удаляет все элементов списка.
}
!! В отличие от вектора у списка нет квадратных скобок, т.к. эта операция не может быть использована эффективно.

Пример использования STL

Функция, формирующая по заданному вектору целых чисел список из элементов вектора с чётными значениями и распечатывающая его:

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

void g (vector <int> & v, list <int? & lst){
        int i;
        for (i = 0; i <v.size ( ); i++)
                if (! (v[ i ] % 2))
                        lst.push_back(v[ i ]);
        list <int> :: const_iterator p = lst.begin ();
        while ( p! = lst.end ()){
                cout << *p << endl;
                p++;
        }
}
```

Достоинства STL-подхода

- Каждый контейнер обеспечивает стандартный интерфейс в виде набора операций. Так что один контейнер может использоваться вместо другого, причём это не влечёт существенного изменения кода.
- Дополнительная общность использования обеспечивается через стандартные итераторы.
- Каждый контейнер связан с распределителем памяти-аллокатором, который можно переопределить с тем, чтобы реализовать собственный механизм распределения памяти.
- Для каждого контейнера можно определить дополнительные итераторы и интерфейсы, что позволит оптимальным образом настроить его для решения конкретной задачи.
- Контейнеры по определению однородны, т.е. должны содержать элементы одного типа, но возможно создание разнородных контейнеров для указателей на общий базовый класс.
- Алгоритмы, входящие в состав STL, предназначены для работы с содержимым контейнером. Все алгоритмы представляют собой шаблонные функции, следовательно, их можно использовать для работы с любым контейнером.

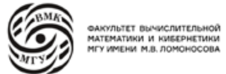
Введение в C++11 (стандарт ISO/IEC 14882:2011)

Вне рассмотрения в рамках курса остаются нововведения для проведения работы с шаблонами:

- введение понятий лямбда-функций и выражений
- внешние шаблоны
- альтернативный синтаксис шаблонных функций
- расширение возможностей использования угловых скобок в шаблонах
- typedef для шаблонов
- шаблоны с переменным числом аргументов
- статическая диагностика
- изменения в STL
- регулярные выражения.

Не рассматриваются также новые понятия

- тривиального класса
- класса с простым размещением
- ***explicit*** перед функциями преобразования
- новшества в ограничениях для ***union***



- новые строковые литералы
- новые символьные типы ***char16_t*** и ***char32_t*** для хранения UTF-16 и UTF-32 символов
- некоторое другое

  ! Ознакомиться с указанной информацией следует самостоятельно если нужно.

Полностью новый стандарт поддерживают компиляторы g++, начиная с версии 4.7….

Для компиляции программы в соответствии с новым стандартом в командной строке в качестве опции компилятору g++ надо указать:

```
-std=c++0x  ( или в новых версиях -std=c++11):
g++ ……. -std=c++0x
( или g++ …….-std=c++11)
```

Введения в C++11 rvalue-ссылки

В C++11 появился новый тип данных **rvalue-ссылка:**

<тип> && **<имя> = <временный объект>;**

В C++11 можно использовать перегруженные функции для неконстантных временных объектов, обозначаемых посредством **rvalue-ссылок**.

Например,

```
class A; …. A a; ….

void f (A & x);          ~ f (a);       // где & обычная ссылка

void f (A && y);         ~ f (A ());    // где && rvalue-ссылка в качестве
                                                 формального параметра
….
A && rr1 = A();      // A && rr2 = a; // Err! – нельзя поступать как с обычными
                                                 ссылками
int && n = 1+2;
n++;
…..
```

Семантика переноса (Move semantic)

При создании/ уничтожении временных объектов **неплоских** классов, как правило, требуется выделение динамической памяти, что может отнимать много времени.

Однако, можно оптимизировать работу с временными объектами неплоских классов, если не освобождать их динамическую память, а просто перенаправлять указатель на неё в объект, который копирует значение временного объекта неплоского класса (посредством поверхностного копирования).
При этом после копирования надо обнулить соответствующие указатели у временного объекта, чтобы его конструктор её не зачистил.

Это возможно сделать с помощью перегруженных конструктора копирования и операции присваивания с параметрами – **rvalue - ссылками**. Их называют конструктором переноса (move constructor) и операцией переноса (move assignment operator).

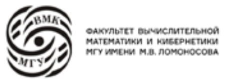
При этом компилятор сам выбирает нужный метод класса, если его параметром является временный объект.

Рассмотрим пример:

```
class Str {
        char * s;
        int len;
public:
        Str (const char * sss = NULL);    // обычный конструктор неплоского класса
        Str (const Str &);                // традиционный конструктор копирования
        Str (Str && x) {                  // move constructor
                s = x.s;
                x.s = NULL;
                len = x.len;
}
Str & operator = (const Str & x);         // обычная перегруженная операция =
Str & operator = (Str && x){              // move assignment operator
                s = x.s;
                x.s = NULL;
                len = x.len;
                return *this;             // возвращаем объект по ссылки
}
~ Str ();                                 // традиционный деструктор неплоского класса
Str operator + Str x
```

Использование **rvalue - ссылок** в описании методов класса Str приведёт к более эффективной работе, например, следующих фрагментов программы:

```
…Str a (“abc”), b (“def”), c;
   c = b+a;    // Str operator = (Str &&); результат временный объект класса Str
```

```
    …

    Str f (Str a) {
        Str b;  ….return a;
    }
… Str d = f (Str (“dd”));  …      // Str (Str &&); конструируется временный объект
                                   (“dd”) по значению
```

Обобщённые константные выражения

Введено ключевое слово **constexpr,** которое указывает компилятору, что обозначаемое им выражение является константным, что в свою очередь позволяет компилятору вычислить и использовать как константу.

Пример:

```
constexpr int given5 () {
    return 5;
}
int mas [givenn5 () + 7];      //создание массива из 12 элементов. В C++11 так можно.
```

Однако, использование **constexpr** накладывает жёсткие ограничения на функцию:

- она не может быть типа **void**
- тело функции должно быть вида **return** *выражение*
- *выражение* должно быть константой
- функция, специфицированная **constexpr,** не может вызываться до её определения.

В константных выражениях можно использовать не только переменные целого типа, но и переменные других числовых типов перед определением которых стоит **constexpr.**

Пример:

```
constexpr double a = 9.8;
constexpr double b = a/6;
```

Вывод типов

Описание ***явно инициализируемой*** переменной может содержать ключевое слово **auto:** при этом типом созданной переменной будет тип инициализирующего выражения.
Auto – это спецификатор для обозначения автоматического класса памяти. Это все локальные объекты, которые мы описываем внутри блока/ функции. Располагаются на

системном стеке и имеют автоматический класс памяти. При выходе из блока автоматически удаляются.

Пример:
Пусть ft (…) – шаблонная функция, которая возвращает значение шаблонного типа, тогда при описании

```
auto var1 = ft (…);
```

переменная var1 будет иметь соответствующий шаблонный тип.
Возможно также:

```
auto var2 = 5;          // var2 имеет тип int
```

Для определения типа выражения во время компиляции при описании переменных можно использовать ключевое слово **decltype.**

Пример:

```
int v1;
decltype (v1) v2 = 5;          // тип переменной v2 такой же как у v1
```

Вывод типов наиболее интересен при работе с шаблонами, а также для уменьшения избыточности кода.

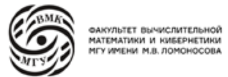
Пример: Вместо

```
for (vector < int >:: const_iterator itr = myvec.cbegin(); itr! = myvec.cend(); ++itr)
…
```

можно написать:

```
for ( auto itr = myvec.cbegin(); itr! = myvec.cend(); ++itr)
….
```

Эти новшества облегчают написание кода.



Лекция 9. Введение в C++11

For-цикл по коллекции

Введена новая форма цикла for, позволяющая автоматически осуществлять перебор элементов коллекции (массивы и любые другие коллекции, для которых определены функции begin () и end ()).

Пример:

```
int arr [5] = {1, 2, 3, 4, 5};
for (int &x : arr) {
        x* = 2;
}…
```

При этом каждый элемент массива увеличится вдвое.

```
for (int x : arr) {
        cout << x<<’ ‘;
```

Улучшение конструкторов объектов

В отличие от старого стандарта новый стандарт C++11 позволяет вызывать одни конструкторы класса (так называемые делегирующие конструкторы) из дргих, что в целом позволяет избежать дублирования кода.

Пример:

```
        class A {
                int n;
        public:
                A (int x) n (x) { }
                A ( ) A (14) { }
};
```

Стало возможно инициализировать члены-данные класса в области их объявления в классе.

Пример:

```
class A {
                int n = 14;
        public:
                explicit A (int x): n (x) { }
                A ( ) A{}
};
```

Любой конструктор класса A будет инициализировать n значением 14, если сам не присвоит n другое значение.

Замечание:

Если до конца проработал хотя бы один делегирующий конструктор, его объект уже считается **полностью созданным.** Однако, объекты производного класса начнут конструироваться только после выполнения всех конструкторов (основного и его делегирующих) базовых классов.

Явное замещение виртуальных функций и финальность

В C++11 добавлена возможность (с помощью спецификатора **override**) отследить ситуации, когда виртуальная функция в базовом классе и в производных классах имеет разные прототипы, например, в результате случайной ошибки (что приводит к тому, что механизм виртуальности для такой функции работать не будет.)

Кроме того, введён спецификатор **final,** который обозначает следующее:
- в описании классов – то, что они не могут быть базовыми для новых классов,
- в описании виртуальных функций – то, что возможные производные классы от рассматриваемого не могут иметь виртуальные функции, которые бы замещали финальные функции.

Замечание:

Спецификаторы **override** и **final** имеют специальные значения только в приведённых ниже ситуациях, в остальных случаях они могут использоваться как *обычные идентификаторы.*

Пример 1:

```
struct B {
        virtual void some_func ();
        virtual void f (int);
        virtual void g () const;
};

struct D1 : public B {
        virtual void sone_func () override;          // Err!: нет такой функции в B
        virtual void f (int) override;               // OK!
        virtual void f (long) override;              // Err!: несоответствие типа параметра
        virtual void f (int) const override;  // Err!: несоответствие квалификации функции
        virtual  int f (int) override;               // Err!: несоответствие типа результата
        c final;                          // OK!
        virtual void g (long)                        // OK! Новая виртуальная функция
```

Пример 2:

```
struct D1 : B2 {                    // см. пример 1 стр.99
    virtual void g () const;        // Err!: замещение финальной функции
};
struct f final {
    int x.y;
};
struct D : F              // Err!: наследование от финального класса
    int z;
};
```

Константа нулевого указателя

В C++11 NULL – это эквивалент константы 0, что может привести к нежелательному результату при перегрузке функций:

```
void f (char *);
void f (int);
```

При обращении f (NULL) будет вызвана f (**int**); что, вероятно, не совпадает с планами программиста.

В C++11 введено новое ключевое слово ***nullptr*** для описания константы нулевого указателя:

```
std :: nullptr_t nullptr;
```

где тип *nullptr_t* можно неявно конвертировать в тип любого указателя и сравнивать с любым указателем.

Неявная конверсия в целочисленный тип **недопустима**, за исключение **bool** (в целях совместимости).

Для обратной совместимости константа 0 также можетиспользоваться в качестве нулевого указателя.

Пример:

```
char * pc = nullptr;        // OK!
int * pi = nullptr;         // OK!
bool b = nullptr;           // OK: b = false;

int i = nullptr;            // Err!:
f (nullptr) ;               // вызывается f (char*), а не f (int)
```

Перечисления со строгой типизацией

В С++:
- перечислимый тип данных фактически совпадает с целым типом,
- если перечисления заданы в одной области видимости, то имена их констант не могут совпадать.

В С++11 наряду с обычным перечислением предложен также способ задания перечислений, позволяющий избежать указанных недостатков. Для этого надо использовать объявление **enum class** (или как синоним **enum struct).** Например,

enum class E {V1, V2, V3, = 100, V4/*101*/};

Элементы такого перечисления нельзя неявно преобразовать в целые числа (выражение E :: V4 == 101 приведёт к ошибке компиляции).

Перечисления с не строгой типизацией

В С++11 тип констант перечислимого типа необязательно **int** (только по умолчанию), его можно задать явно следующим образом:

enum class E2 : **unsigned int** {V1, V2}; // значение E2 :: V1 определено, а V1 – не определено

enum class E3 : **unsigned long** {V1 = 1, V2}; // в целях обеспечения обратной совместимости определены и значение E3 :: V1, и V1

В С++11 возможно предварительное объявление перечислений, но только если указан размер перечисления (явно или неявно):

enum E1; // Err!: низлежащий тип не определён
enum E2 : **unsigned int;** **//** OK!
enum class E3 : **//** OK! низлежащий тип **int**
enum class E4 : **unsigned long;** **//** OK!
enum class E2 : **unsigned short;** // Err!: ранее объявлен с другим низлежащим типом

sizeof для членов данных классов без создания объектов

В С++11 разрешено применять операцию **sizeof** к членам-данным классов независимо от объектов классов.

Пример:

```
struct A {
        some_type_a;
};
sizeof (A :: a) …  // OK!
```

Кроме того, в C++11 узаконен тип **long long int.**

Рассмотрим ещё один пример с использованием наследования.

```
class A {
virtual void f () {cout <<’’f() from  A’;}
class B; public A {
virtual void f () {cout <<’’l() from  B’;}
                                    // Определим {B b; - объект b класса B
                                    b/F ();
```

Unspecified behavior (неспециализированное поведение)

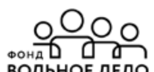
а + в
(а + в) + с
 а = (в = с)

Underfined behavior (UB)

Операция сдвига вправо: 4 >> -2; , где мы пытаемся сдвинуть на -2, то стандарт в данном случае ничего не гарантирует. Какая-то реализация может считать эту программу корректной и исполнить её (например, двигать в другую сторону). А другая программа ещё на этапе компиляции может зафиксировать в данной программе ошибку.
Если мы хотим сделать программу независимой от компиляторов, то надо избегать свойств UB, которые дают неопределённое поведение.

Рассмотрим *пример:*

```
struct X {
X (int);
n x (i);
{ X b;
char x buf = new char [struct (X)];
X * p = new (buf) X (222);
```



```
….
P –> X::` X ();//
B~ X ();  //  может быть UB
delete () buf;
}
```

Рассмотрим *пример с областью видимости:*

```
namespace A {        int x;              A::x;
        int y;                           B::x;
        void f ();                       double z;
        void f (char);                   double x;
}                                        int f;
namespace B {        int x;              void f (double);
        int y;                           z; f(); x; f(2.0);
        void f ();                       using namespace A;
        void f (char);                   x;              // double
}                                        y;              // из A
                                         f (‘a’);        // из A
                                         using namespace B;
                                         x;              // double
                                         B::x;           // int из B
                                         f ();           // f из текущего блока
                                         f (‘a’);        // ошибка, неоднозначность
```

Как открыть файл поток:

```
std :: ifstream  f1 (“file1.txt”);        // поток ввода
std :: ofstream  f2 (“file2.txt”);        // поток вывода
```

Более подробно о работе с потоками можно прочитать в справочнике или дополнительной литературе.

Мы рассмотрели средства, которые можно применить в объектно-ориентированном программировании С++.
В С++ появились три вещи, где управление без нашего ведома передаётся в другую часть программы:

1. Конструкторы и деструкторы – они вызываются автоматически и управление передаётся на соответствующий конструктор, когда мы описываем объект с инициализацией и когда он уничтожается, то перед его уничтожением автоматически вызывается деструктор.

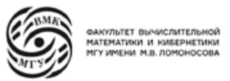
2. Виртуальные функции – при вызове виртуальной функции через указатель на базовый класс управление может перейти в функцию заместитель производного класса.
3. Исключение – управление передаётся нестандартным образом, когда объект летит вниз по коду или вперёд по коду к ловушке (трай-блоки). В том выражении, где функция была вызвана из этой функции в месте значения вылетаю исключения и продолжают распространяться в новом контексте вызова функции.

Всё это позволяет писать компактные программы и тратить меньше усилий. Наследование позволяет использовать уже готовые классы и на их базе создавать новые, добавляя необходимое.

Стандартная библиотека шаблонов удобна в использовании. Но за корректность шаблонов отвечает программист самостоятельно.

Язык С++ широко используется в программировании для написания прикладных программ, встроенных систем и т.д.

С++ сохраняет все преимущества языка С, а также предлагает новый подход к программированию.



Лекция 10. Вычислительные системы

Структура вычислительной системы

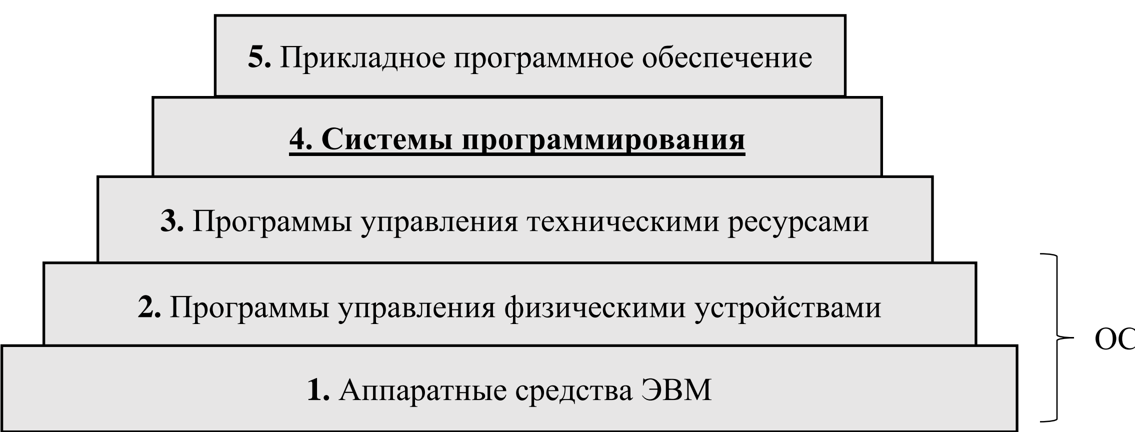


Рис.10.1 Структура вычислительной системы

Система программирования (СП) – это комплекс программных инструментов и библиотек, который поддерживает ***весь*** технологический цикл создания программного продукта (ПП).

ПП – программа, оформленная, документированная и специфицированная таким образом, что её можно использовать независимо, отчуждённо от автора программы.

Создание программного продукта (ПП)

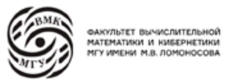
1. **Анализ требований**

Уточняются, формализуются и документируются требования заказчика к ПП, в результате создаются *внешняя спецификация ПП*, т.е. характеристика программы с точки зрения заказчика.
Часть требований к ПП можно формально записать с помощью языков спецификаций (SDL, ….), таблиц решений, функциональных диаграмм.

2. **Проектирование**

Выделяются отдельные модули, определяется их иерархия и сопряжение между ними. В результате создаётся общая *схема иерархии и внешняя спецификация отдельных модулей.*
По внешним спецификациям разрабатывается внутренняя структура каждого модуля – выбирается алгоритм работы модуля и способ внутреннего представления данных.

3. **Кодирование**
4. **Компоновка и интеграция**
5. **Тестирование и отладка**
   Верификация (ПП работает согласно спецификации).



Валидация (ПП пригоден для использования).
Тестирование:
- ручное означает запуск тестирования вручную,
- автоматизированное означает использование специальных средств для генерации тестов и для прогона автоматически сразу серии тестов (получить результат какие тесты не прошли);
- функциональное,
- интеграционное,
- модульное;
- регрессионное.

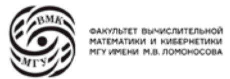
Формальное доказательство корректности работы, включая испытание на аппаратуре, для которой разрабатывался ПП,

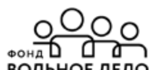
6. **Документирование**
   Необходимо чётко описать что/как устроено.
7. **Внедрение**
   Установка ПП на технику заказчика и настройка.
8. **Сопровождение**
   Постоянная поддержка в случае, если выявляются ошибки или остались непонятные вещи для выяснения.

Каскадно-возвратная модель

Рис.10.2 Каскадно-возвратная модель

С любого этапа можно вернуться на предыдущий этап или на несколько этапов назад.





Итерационная модель

Создаётся прототип с уменьшенной функциональность, отражается самое важное. Создаётся полный цикл от анализа требований до документирования.
Позволяет на следующем витке добавлять функциональность. При этом уже имеется работающая версия, которую можно показать заказчику и вносить при необходимости изменения, полученные от заказчика.

Рис.10.3 Итерационная модель

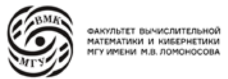
Основные компоненты системы программирования

1. **Транслятор** (переводит программы с языка программирования на машинный язык, что и позволяет выполнить их на ЭВМ).
2. **Макрогенератор** или макропроцессор (работает непосредственно перед транслятором, используется для получения исходной программы).
3. **Редактор текста** (используется для составления программ на языке программирования).
4. **Редактор связей** или компоновщик (предназначен для связывания между собой (по внешним данным) объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав СП).
5. **Отладчик** (используется для проверочных запусков программ и исправления ошибок).

6. **Библиотеки стандартных программ** (облегчает работу программиста, используются на этапе трансляции и исполнения).

Дополнительные компоненты систем программирования

a) **Системы контроля версий** для версионирования исходного текста ПП.
b) **Средства конфигурирования**
    - помогают создавать **различные конфигурации** ПП в зависимости от конкретных параметров системного окружения, в котором ПП будет функционировать и от возможных различий отдельных версий ПП;
    - поддерживают информацию обо всех предполагаемых и выполненных **изменениях** ПП;
    - обеспечивают координированное **управление** развитием функциональности и улучшения характеристик системы.
c) **Средства тестирования** (помогают при составлении набора тестов).
d) **Профилировщик.**
   Профилирование – определение в процентах времени, затрачиваемого на выполнение отдельных фрагментов программы, как правило, для линейных участков кода (фрагментов программы, где нет передачи управления). Профилировщик часто используется для выявения мест для дальнейшей оптимизации программы.
e) **Справочная система** (содержит справочные материалы по языку программирования и компонентам СП).
f) **Инструменты для статического анализа кода**
    - Производят анализ логики работы программы без её исполнения (работают с исходным текстом программы).
    - Основное применение – поиск мест, где может содержаться логическая ошибка (lint и аналоги).
    - Также используются для организации навигации по коду (генерация т.ню тэгов), полуавтоматического рефакторинга и прочее.
g) **Средства навигации по коду**
   В простейшем варианте (ctags) – анализ исходного текста, поиск в нём символов (определений функций, классов) и формирование указателя найденных символов для использования в текстовом редакторе. В более сложных случаях отыскиваются также отношения наследования, места использования символа в коде и прочее.
h) **Инструменты подготовки документации**
   Используются для автоматической генерации списков классов, функций и т.п. по исходному коду. При этом автоматически извлекаются



комментарии к коду, и на выходе генерируется документация к коду, которая может компоноваться с концептуальной документацией на ПП и его подсистемы.

i) **Управление разработкой**
Планирование, отслеживание замечаний.

Виды систем программирования

Рассмотрим виды систем программирования по стратегии интеграции:

1. Наборы независимых инструментов
2. Интегрированные системы программирования

Стратегии трансляции

- Компиляторы и ассемблеры
- Интерпретаторы (не создают объектный файл; интерпретируют текст и сразу выполняет)
- Смешанная стратегия (байт-код, JIT-компиляция; трансляции на более низкий язык)

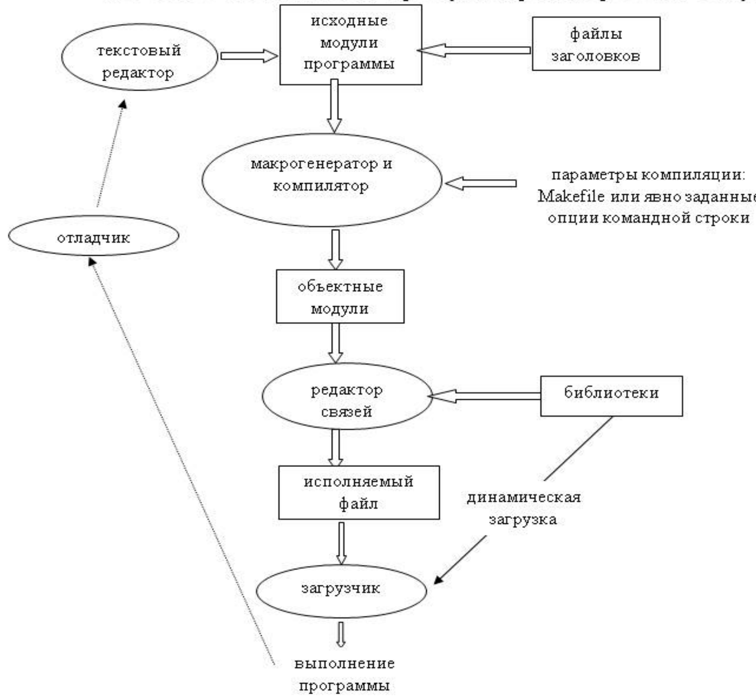
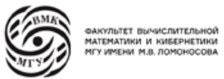


Рис.10.4 Общая схема функционирования основных компонентов СП на базе компилятора



Рассмотри пример схемы на рис.10.4 стр.112 на базе интерпретатора.

Рис.10.5 Общая схема функционирования основных компонентов СП на базе интерпретатора

Интегрированная среда разработки (ИСР)

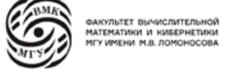
ИСР (IDE, integrated development environment) – комплекс средств, поддерживающих полный жизненный цикл программного продукта (ПП).

Простая ИСР содержит минимальный набор компонентов:

- Текстовый редактор
- Компилятор
- Редактор связей
- Отладчик.

Состав продвинутой ИСР

- **Модуль системы контроля версий** – все объекты, с которыми идёт работа в рамках ИСР, хранятся в репозитории системы контроля версий;
- **Графические средства** анализа и проектирования, обеспечивающие создание и редактирование иерархически связанных диаграмм, образующих модели ПП, а также графический пользовательский интерфейс, обеспечивающий взаимодействие с функциями API;



- **Средства разработки приложений,** включая инструменты кодогенерации;
- **Компилятор;**
- **Текстовый редактор;**
- **Средства статического анализа кода** с поддержкой навигации по коду и полуавтоматического рефакторинга;
- **Средства документирования;**
- **Средства тестирования и отладки;**
- **Средства** управления **проектной деятельностью**, в т.ч. интеграция с системами отслеживания замечаний;
- **Средства обратного конструирования** позволяют восстановить логику и структуру программы по её исходным текстам, с целью модификации или перенесения на другую платформу.

Текстовые редакторы

1. **Пакетные –** редакторы, управляемые макросредствами. Обычно обрабатывают в потоке несколько файлов, осуществляя некоторые замены с помощью макросредств.

2. **Диалоговые –** это редакторы, в которых программист создаёт программу.
   Они могут быть:
   - строчные (необходимо давать команду какую строку нужно редактировать, далее переходить в неё и редактировать);
   - экранные (позволяют средствами клавиатуры и мыши осуществлять удобную навигацию по тексту).

Возможности текстового редактора в ИСР

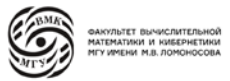
1. Подготовка текста программы (обычные действия по созданию, редактированию, сохранению файла с текстов программы).

2. Многооконный интерфейс, управление окнами и вкладками.

3. Закладки, настраиваемые сочетания клавиш, шаблоны фрагментов текста, программное управление самим редактором.

4. Интеграция с компилятором и средствами статического анализа кода.

5. Интеграция с отладчиком:
   - отображение контрольных точек останова при отладке,



- отображение текущего значения объекта, при наведении курсора на идентификатор.

Дополнительные возможности текстового редактора в ИСР

Интеграция с компилятором и/или средствами статического кода;

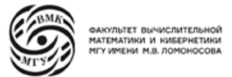
- визуализация текста с выделением лексем (синтаксическая подсветка элементов языка);
- дополнение кода, интерактивная подсказка;
- всплывающая подсказка об атрибутах идентификаторов, если на них установить курсор, отображение ошибок, обнаруженных на этапе компиляции в тексте программы;
- навигация по коду – переход к определению имени, поиск мест использования имени, поиск имени, навигация по иерархии наследования;
- рефакторинг кода – от простейших случаев: переименование идентификатора, генерация заглушки – до сложных: выделение базового класса, перенос некоторого метода вверх или вниз по иерархии

Задачи отладчика в рамках ИСР

1. Пошаговое выполнение программы (шаг = строка; с трассировкой внутри вызываемой функции и без неё);
2. Выполнение программы до строки, в которой в редакторе стоит курсор;
3. Выделение выполняемой в данный момент строки;
4. Приостановка выполнения программы. При этом:
   - можно запросить значение переменно;
   - можно заказать вычисление некоторого выражения;
     можно изменить значение переменной и продолжить выполнение программы (но не всякий отладчик позволяет изменять программный код, т.е. поддерживает частичную перекомпиляцию);
5. Расстановка/ снятие точек останова, которые визуализируются на текстовом редакторе;
6. Выдача всей информации в терминах исходной программы.

Стратегии тестирования

Стратегия тестирования – это метод, используемый для отбора текстов, которые должны быть включены в тестовый комплект.
Стратегия является эффективной, если тесты, включённые в неё с большей вероятностью, обнаружат ошибки тестируемого объекта. Эффективность стратегии



зависит от комбинирования природы тестов и природы ошибок, на поиск которых эти тесты направлены.

Стратегия поведенческого теста основана на технических требованиях. Тестирование, выполняемое с помощью стратегии поведенческого теста, называется *поведенческим тестированием, функциональным тестированием или тестированием из чёрного ящика.*

Стратегия структурного теста определяется структурой тестируемого объекта. Тестирование, выполненное с помощью стратегии структурного теста, называется также *тестированием белого/ прозрачного ящика.* Стратегия структурного теста требует полного доступа к структуре объекта, т.е. к исходной программе.

Стратегия гибридного теста является комбинацией поведенческой и структурной стратегий. Модули и низкоуровневые компоненты часто тестируются с помощью структурной стратегии. Большие компоненты и системы в основном тестируются с помощью структурной стратегии. Гибридная стратегия полезна на всех уровнях.

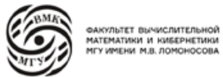
Способы тестирования

1. Тестирование проводится не только на той стадии разработки программ, которая специально для этого предназначена, но и на предшествующих стадиях – при автономной отладке программ, ещё до объединения их в единый программный комплекс. Такое тестирование называется ***модульным***. Его обычно проводят сами разработчики, которые проверяют точное соответствие программы выданной им спецификации.

2. ***Интеграционное тестирование*** призвано проверить все аспекты работы программы от правильности взаимодействия внутренних программных компонентов до правильности взаимодействия программного комплекса с его пользователями.

3. Во время ***пользовательского тестирования*** результаты работы программы проверяются с прикладной точки зрения.

4. ***Нагрузочное тестирование*** даёт возможность проверять безопасную и эффективную работу созданной программы в нормальных и пиковых режимах её использования. Функциональность на этом этапе проверяется только в смысле её влияния на важнейшие технические параметры программы, например, на время реакции системы на запрос пользователя.



5. Важной в тестировании является возможность проведения **регрессионного тестирования.** Регрессионные тесты, повторяемые после каждого исправления программы, позволяют убедиться, что функциональность программы, не связанная с внесённым исправлением, не затронута этим исправлением и не утрачена из-за него.

Редактор связей

Редактор связей (компоновщик) предназначен для связывания между собой (по внешним данным) объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав СП.

Редактор связей выполняет следующее:

- Связывает между собой по внешним данным объектные модули, порождаемые компилятором и составляющие единую программу,
- Связывает файлы статически подключаемых библиотек с целью получения единого исполняемого модуля,
- Готовит таблицу трансляции относительных адресов для загрузчика,
- Готовит таблицу точек вызова функций динамически подключаемых библиотек.

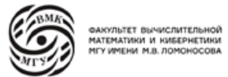
Типы библиотек

Библиотеки являются существенной частью систем программирования.
В настоящее время можно выделить три типа библиотек:

1. Библиотеки функций (или подпрограмм)
2. Библиотеки классов
3. Библиотеки компонентов.

(1) Библиотеки функций

Библиотеки функций во многом определяют возможности систем программирования в целом. Чем больше выбор библиотечных функций СП (система программирования) предоставляет пользователю, тем лучше позиции она имеет на рынке средств разработки программного обеспечения.
Различают:



- *Библиотеки для языка программирования* (например, функции ввода-вывода, работа со строками);
- *Библиотеки для решения задач в конкретной проблемной области* (например, функции, реализующие алгоритмы линейной алгебры).

Библиотеки функций представляют собой **откомпилированные объектные модули**, а необходимые фрагменты библиотеки функций включаются в исполняемый файл на этапе работы редактора связей.

(2) Библиотеки классов

Библиотеки классов также являются важной частью современных систем программирования, базирующихся на ООЯП.

Недостаток библиотеки классов – все её классы должны быть написаны на том же языке программирования, на котором пишется программа, куда интегрируются библиотечные классы.

В библиотеке классов различают:

- Конкретные классы
- Абстрактные классы, иерархии классов
- Шаблоны классов, иерархии шаблонов классов.

Библиотеки классов включаются в программу на этапе компиляции и компилируются со всей программой вместе.

(3) Библиотеки компонентов

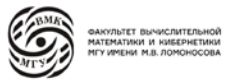
Библиотеки компонентов – это библиотека готовых откомпилированных программных модулей, предназначенных для использования в качестве составных частей программ, и которыми можно манипулировать во время разработки программ.

Компоненты бывают **локальные** (находящиеся на той же ЭВМ, где создаётся ПП) и **распределённые** (расположенные на сервере и доступные по сети ЭВМ).

Примеры технологий, использующих библиотеки компонентов:

- Технология CORBA (Common Object Request Broker Architecture) от международной группы OMG позволяет использовать программные компоненты, размещённые как локально, так и дистанционно. Использование CORBA – компонент не зависит от языка, на котором они были написаны.

- Технология COM (Common Object Model) от компании Microsoft под ОС Windows позволяет использовать локально размещённые компоненты



независимо от языка их реализации. Её развитие привело к распределённой архитектуре DCOM (Distributed COM), а затем к ActiveX.

- Технология Java Beans от Sun Microsystems позволяет использовать компоненты, написанные на языке Java. Так как реализация Java-машины существует почти для всех ОС, отсутствует жёсткая привязка к конкретной ОС.

Динамически подключаемые библиотеки (ДБ)

ДБ в отличии от статических библиотек подключаются к программе не во время компиляции программы, а непосредственно в ходе её выполнения.
На этапе компоновки программы редактор связей, встречая вызовы функций ДБ, вместо процедуры связывания формирует таблицу точек вызова функций ДБ для последующей операции динамического связывания. Таким образом, процесс полной компоновки завершается уже на этапе выполнения целевой программы.

Преимущества динамических библиотек:

- Не требуется включать в программу объектный код часто используемых функций, что существенно сокращает объем кода;

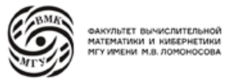
- Различные программы, выполняемые в некоторой ОС, могут пользоваться кодом одной и той же библиотеки, содержащейся в ОС;

- Изменения и улучшения функций библиотек сводятся к обновлению только содержимого ДБ, а уже существующие тексты программ не требуют перекомпиляции (этот же факт может оказаться недостатком, если при модификации функции меняется логика их работы, поэтому использование ДБ накладывает определённые обязательства как на разработчика программы Б так и на создателя библиотеки).

Как правило, динамически подключаются системные функции ОС и общедоступные функции программного интерфейса (API).
Можно создавать свои динамические библиотеки при разработки прикладных программ.

Критерии проектирования стандартных библиотек. Требования по составу.

Стандартная библиотека должна:
- Обеспечивать поддержку свойства языка (например, управление памятью, предоставление информации об объектах во время выполнения программ);



- Предоставлять информацию о зависящих от реализации аспектах языка (например, о максимальных размерах целых значений);

- Предоставлять функции, которые не могут быть написаны оптимально для всех вычислительных систем на данном языке программирования (например, soft) или memmove () – пересылка блоков памяти);

- Предоставлять программисту нетривиальные средства, на которые он может рассчитывать, заботясь о переносимости программ (например, средства работы со списками, функции сортировки, потоки ввода/ вывода);

- Предоставлять основу для расширения собственных возможностей, в частности, соглашения и средства поддержки, позволяющие обеспечить операции для данных, имеющих определяемые пользователями типы, в том же стиле, в котором обеспечиваются операции для встроенных типов (например, ввод/ вывод).

- Служить основой и теоретическим базисом других библиотек.

Требования по свойствам компонентов стандартной библиотеки

Компоненты стандартной библиотеки должны:

- Иметь ***общезначимый*** характер (структура данных и алгоритмы для работы с ними – стек, очередь, список, …, сортировка, поиск, копирование, ...), быть важными и удобными для использования всеми программистами;

- Быть настолько ***эффективными***, чтобы у пользователей библиотеки не возникло потребности заново программировать библиотечные средства;

- Быть ***независимыми от*** конкретных ***алгоритмов*** или предоставлять возможность указать алгоритм в качестве параметра;

- Оставаться элементарными, чтобы не терять эффективность из-за излишних усложнений или попыток совместить различные функции в одной;

- Быть ***безопасными*** устойчивыми к неправильному использованию, использование библиотеки не должно провоцировать ошибки, а наоборот, снижать их вероятность);

- Обладать достаточной полнотой ***(завершённостью),*** чтобы ни у кого не возникло желания что-то заменить или доопределить;

- Обладать удобной и безопасной системной умолчаний;

- Поддерживать общепринятые стили программирования;

- Обладать способностью к расширению, чтобы работать с типами, определяемыми пользователем, было также хорошо, как и со встроенными (базовыми) типами ***(сочетаемость с базовыми типами данных и базовыми операциями).***

СП под UNIX. Координатор GNU Make

Make существенно упрощает процесс сборки проектов.
Make отслеживает изменившиеся файлы и перекомпилирует при обращении к нему только их и файлы, связанные с ними по компиляции.

Информация о зависимостях по компиляции и необходимые команды по компиляции содержатся в файле *Makefile (makefile* или в файле с соответствующей структурой, имя которого задаётся при обращении к *Make: Make* -f <имя_файла>*)*, который должен находиться в текущей директории.

Makefile состоит из последовательности записей вида:

```
Цель: зависимости_по_компиляции
        команда ОС UNIX
        …….
        команда ОС UNIX
    ……..
```

Цель – имя целевого файла ли название действия.

Если обращение к Make происходит без параметра, то выполняются действия по достижению первой цели, если же параметр есть, то Make достигает цель, имя которой совпадает с именем параметра.

Если цель – имя файла, то Make автоматически по дате модификации файлов, указанных среди файлов-зависимостей по компиляции, определяет, какие из них должны быть перекомпилированы и выполняет соответствующие команды.

В Makefile должны быть указаны зависимости и команды для получения как промежуточных, так и исполняемых файлов.

Пример 1: Makefile (для модельного SQL-интерпретатора)
client: client.o

```
        cc -o client client.o

server: server.o   parse.o   getlex.o   table.o
         cc -o server.o   parse.o   getlex.o   table.o

table.o:  table.c   table.h
        cc -c   table.c

parse.o:    parse.c   parse.h   getlex.h   table.h
        cc -c   parse.c

getlex.o: getlex.c   parse.h  getlex.h
        cc -c   getlex.c

server.o: server.c   parse.h   getlex.h
         cc -c server.o

client.o: client.c
        cc -c client.c

clean:
        rm*.o

all: client  server
```

Некоторые дополнительные возможности создания Make-файлов

В начале файла можно вводить макросы для обозначения каких-либо часто повторяющихся фрагментов текста файла в виде:

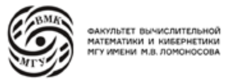
имя = текст

затем там, где нужно, использовать введённые имена таким образом:

$ (имя)

Некоторые предопределённые макроопределения:

$@ - полное имя текущей цели
$+ - имя текущей цели без типа файла (суффикса)
$? - список зависимостей, которые обновились с момента предыдущего обновления цели



$< - полное имя исходного файла, к которому применяется правило трансформации.

В язык для Makefile введены некоторые предопределённые правила суффиксов по умолчанию для стандартного получения результирующих файлов по исходным файлам.

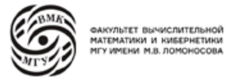
Например, правило трансформации **.c.o** означает, что для того, чтобы получить файл с расширением .o (если есть файл :c) надо выполнить указанную команду.

Строка, начинающаяся символом # является комментарием.
Пустые строки игнорируются.
Любая строка, оканчивающаяся символом “ \ ” продолжается на следующую строку.

gcc -MM позволяет сгенерировать фрагмент make-файлов с зависимостями модулей.

Пример 2: Makefile (для модельного SQL-интерпретатора)

```
cc = gcc
server_o =  server.o   parse.o   getlex.o   table.o

client: client.o
        $ (cc)  -o client client.o

server.o: $ (server_o)
         $ (cc) -o   server $ (server_o)
.c.o:
        $ (cc) -c $ *.c

tables .c:      tables .h
parse. c:       parse.h   getlex.h   table.h
getlex. c:      parse.h   getlex.h
table. c:       parse.h   getlex.h

clean:
        rm*.o

all: client  server
```

Системы контроля версий

Система контроля версий в самом общем понимании осуществляет отслеживание версий (ревизий) некого набора объектов (например, файлового дерева).

Применительно к процессу разработки ПП говорят:

- Об **управлении исходным кодом** (source control), при этом отслеживаются ревизии исходного кода ПП и других ресурсов, необходимых для сборки ПП.
- Или **об управлении конфигурациями** (configuration management), при этом отслеживаются ревизии всего окружения, в том числе, сопутствующих материалов, таких как файлы данных и документация.

Система контроля версий позволяет фиксировать ревизии исходного кола ПП, возвращаться к любой из них, отслеживать авторство изменений, производить анализ истории изменений и т.д.

Системы контроля версий

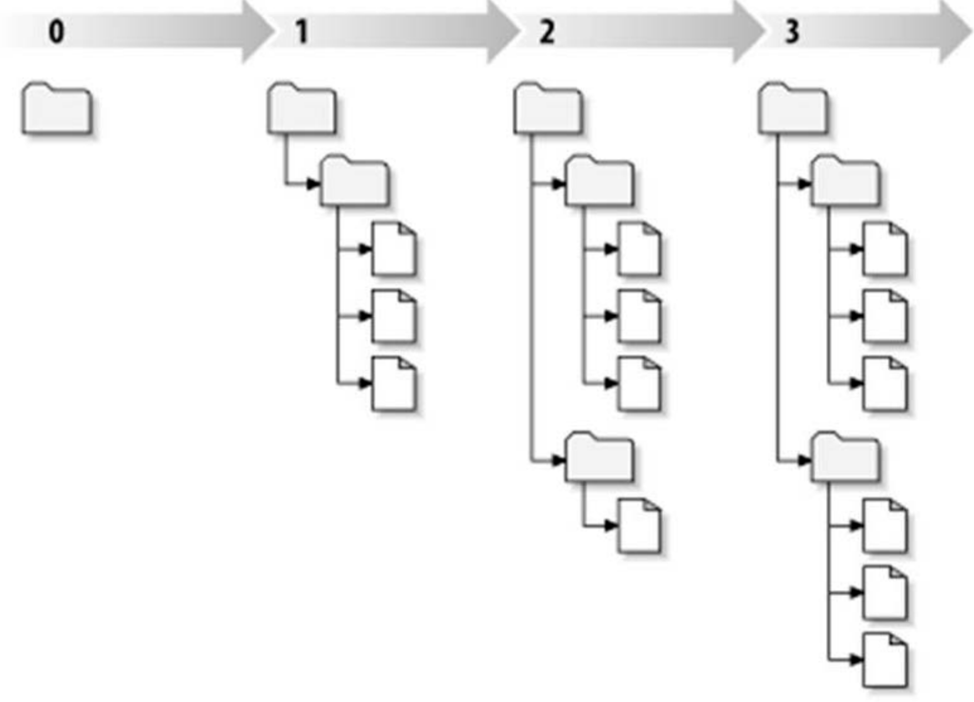
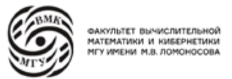


Рис.10.6 Система контроля версий

Развития систем контроля версий

- Старейшие системы:



SCCS (1972), RCS (1982)
  - Отслеживается один файл на одной машине
  - Рядом с файлом хранится история всех его ревизий.
- Проектные клиент-серверные системы:
  CVS (1990), SVN (2000)
  - Поддерживается отслеживание файлового дерева
  - Фиксируется ревизия дерева в целом
  - История версий хранится в репозитории (на сервере)
  - Работа с кодом ведётся в рабочих копиях.
- Распределённые системя:
  BitKeeper (1998), Darcs (2002), Git (2005), Mercurial (2005)
  - Каждая рабочая копия может иметь собственный репозиторий
  - Работа с репозиторием не требует доступа к серверу
  - Возможна децентрализованная разработка.

Контроль версий: основные понятия

Сущности

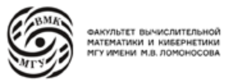
- **Дерево**
- **Ревизия**
- **Набор изменений** (changeset)
- **Ветка**
- **Репозиторий**
- **Рабочая копия**

Операции

- **Фиксация** (commit)
- **Обновление** на ревизию
- **Ветвление**
- **Слияние** (merge)
- **Передача изменений (pull/push)**

Пример 1: **История изменений дерева**

Рис. 10.7 История изменений дерева



Пример 2: **Ветки**

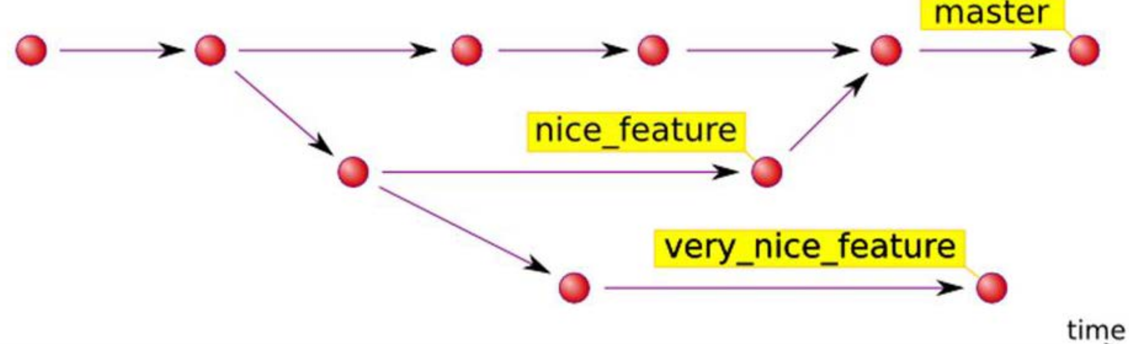


Рис. 10.8 Ветки

Контроль версий: классификация

По расположению репозитория:

- Централизованные (CVS, SVN)
  Распределённые (Darcs, Git, Mercurial)
- Комбинированные (Bazaar)

По объекту отслеживания:

- Отслеживающие ревизии (CVS, SVN)
- Отслеживающие наборы изменений
  - наборы изменений организованы в ациклический орграф (Git, Mercurial)
  - наборы изменений организованы как набор патчей (Darcs)

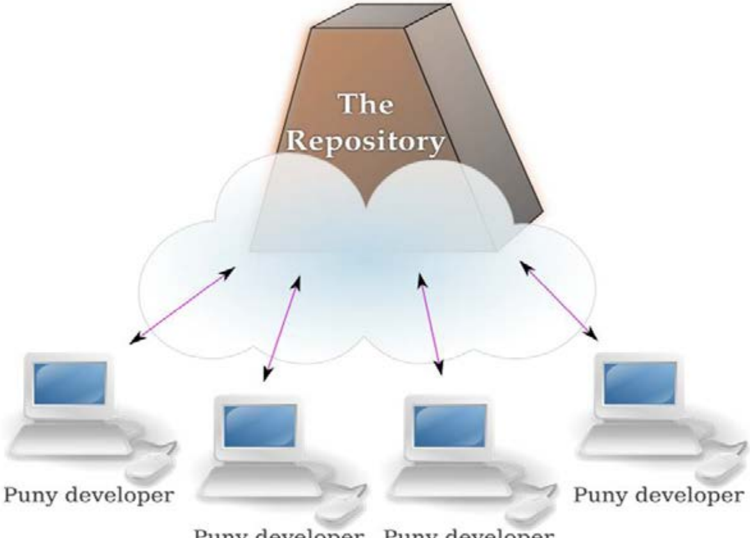
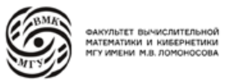


Рис.10.9 Централизованные системы



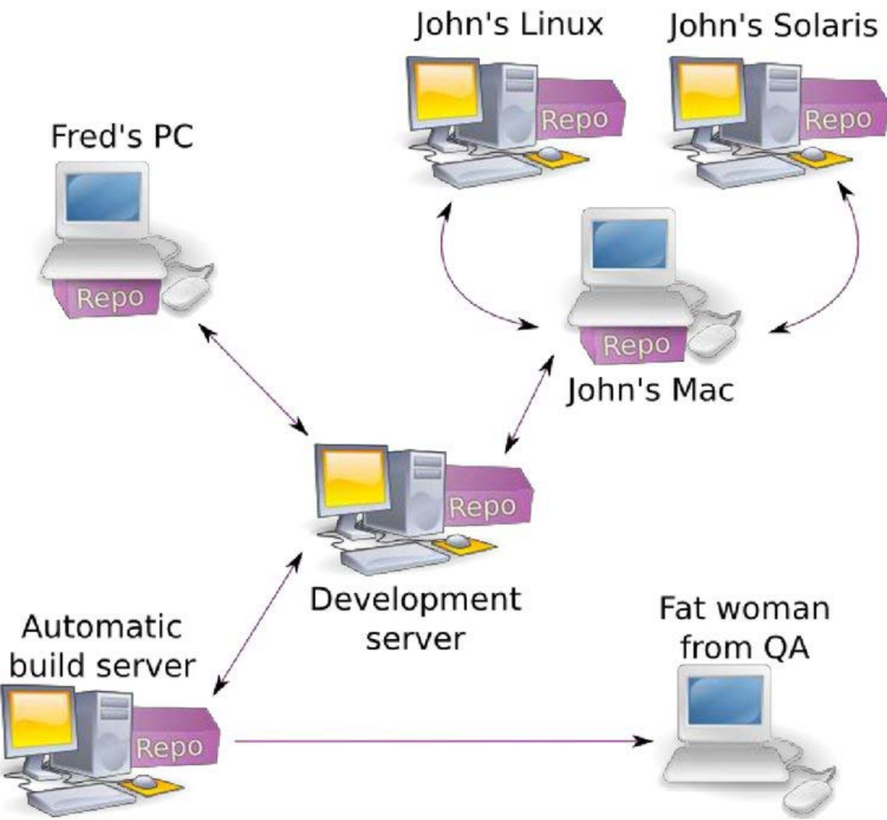


Рис.10.10 Распределённые системы

Примеры работы с системами контроля версий

1. **Линейная работа.**
   Последовательная фиксация прогресса работы над ПП.
2. **Совместная линейная работа.**
   Совместная работа с фиксацией по принципу «кто успел» и последующим слиянием изменений.
3. **Ветки разработки**, в которых новая функциональность дорабатывается до готовности к выпуску версии ПП.
4. **Ветки для тестирования.**
5. **Ветки для сопровождения старых версий.**
6. **Метки** (тэги) для фиксации значимых ревизий (например, версий ПП).
7. **Анализ истории** (в том числе аннотирование кода).

Режим аннотирования кода

```
[1586]  110 class ReportModule(Component):
[1]     111
[1860]  112     implements(INavigationContributor, IPermissionRequestor, IRequestHandler,
        113                IWikiSyntaxProvider)
[1586]  114
[6901]  115     items_per_page = IntOption('report', 'items_per_page', 100,
        116         """Number of tickets displayed per page in ticket reports,
        117         by default (''since 0.11'')""")
        118
        119     items_per_page_rss = IntOption('report', 'items_per_page_rss', 0,
        120         """Number of tickets displayed in the rss feeds for reports
        121         (''since 0.11'')""")
[11096] 122
[1586]  123     # INavigationContributor methods
        124
        125     def get_active_navigation_item(self, req):
        126         return 'tickets'
        127
        128     def get_navigation_items(self, req):
[4143]  129         if 'REPORT_VIEW' in req.perm:
[5776]  130             yield ('mainnav', 'tickets', tag.a(_('View Tickets'),
[4787]  131                                          href=req.href.report()))
[1586]  132
[11096] 133     # IPermissionRequestor methods
[1860]  134
[11096] 135     def get_permission_actions(self):
```

Рис.10.11 Режим аннотирования кода

Синим цветом отмечены наиболее старые изменения, оранжевым цветом отражены последние (свежие) изменения.

Популярные современные системы

1. Git

- высокая скорость
- Github

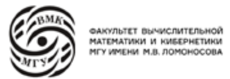
2. Mercurial

- простота в использовании
- кроссплатформенность

3. Subversion (SVN)

- централизованная система.

Case-средства (Computer Aided Software Engineering) – программные средства, поддерживающие полуавтоматическую разработку комплексного ПП на всех стадиях его жизненного цикла.



Основные характерные особенности Case-средств:

- Наличие мощных **графических средств** для описания и документирования ПП, обеспечивающие удобный интерфейс с разработчиком и развивающие его творческие возможности;
- **Интеграция** отдельных компонент Case-средств, обеспечивающая управляемость процессом разработки программной системы;
- Наличие средств автоматического или автоматизированного кодирования и документирования ПП.

Примером наиболее известного Case-средства является объектно-ориентированное Case-средство **Rational Rose** (компании Rational Software corporation).

В основе работы Rational Rose лежит построение диаграмм и спецификаций унифицированного языка моделирования **UML** (Unified Modeling Language) определяющих архитектуру проекта, его статистические и динамические аспекты.

UML – язык для определения, представления, проектирования и документирования программных систем различной природы.

Некоторые дополнительные возможности современных систем программирования

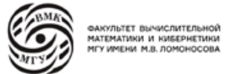
Существуют множество других программных средств, помогающих в проектировании, модификации и кодировании программ.

Например,

- Системы, преобразующие программы на процедурном (императивном) ЯП в программы на ООЯП (поскольку ОО программы считаются более простыми в сопровождении, это бывает полезно);
- Системы, анализирующие исходный код, с целью получения высокоуровневых абстракций (например, Java → диаграммы UML);
- Средства, предоставляющие среду разработки программ по диаграммам UML;
- Средства сборочного программирования (из готовых программных модулей, официально распространены достаточно слабо в связи с различными правовыми проблемами копирования модулей, низким качеством модулей и их плохой документацией).

Статистика отмечает, что около 80% программного обеспечения создаётся по уже имеющемуся.

Следовательно, большой интерес представляют собой репозитории, поддерживающие архивы, документацию и интеллектуальный поиск нужных прототипов и фрагментов проектов программ для реализации эффективного сборочного программирования.



Лекция 11. Трансляторы

Транслятор – это программа, преобразующая программный код с того или иного языка программирования в машинный код.

Существует два типа трансляторов:
- **компиляторы** (преобразуют всю программу в модуль на машинном языке, после чего программа записывается в память в память компьютера и лишь потом исполняется) и --
- **интерпретаторы** (производят пооператорную обработку и выполнение исходного кода программы).

Компилятор похож на письменный перевод. Интерпретатор похож на устного переводчика.

Компиляторы и интерпретаторы

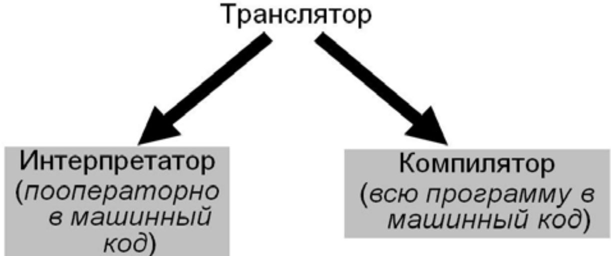
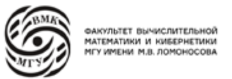
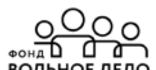


Рис.11.1 Компиляторы и интерпретаторы

Компилятор анализирует программу на входном языке и создаёт эквивалентную объектную программу. гибридная

Система программирования компилирующего типа

Рис.11.2 Система программирования компилирующего типа

Компилятор переводит программы с одного языка на другой. Выходом компилятора служит цепочка символов, составляющая исходную программу на языке программирования L1. Выход компилятора (объектная программа) также представляет собой цепочку символов, но принадлежащую другому языку L2, например, языку некоторого компьютера. Примеры: C/C++, Pascal, Delphi, Go.

В отличие от компилятора **интерпретатор** не создаёт никакой новой программы. Входными данными *интерпретатора* является не только исходная программа, но и входные данные самой исходной программы.

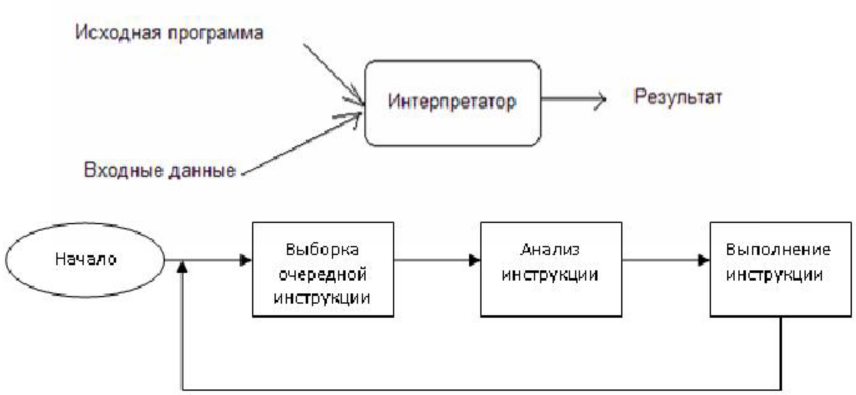


Рис.11.3 Система программирования интерпретирующего типа

Интерпретация позволяет выполнить более гибкую и лучшую диагностику ошибок, чем компиляция. В современных трансляторах часто используются как элементы компиляции, так и интерпретации. Примеры: Python, PHP, JavaScript и др.

Смешанная стратегия

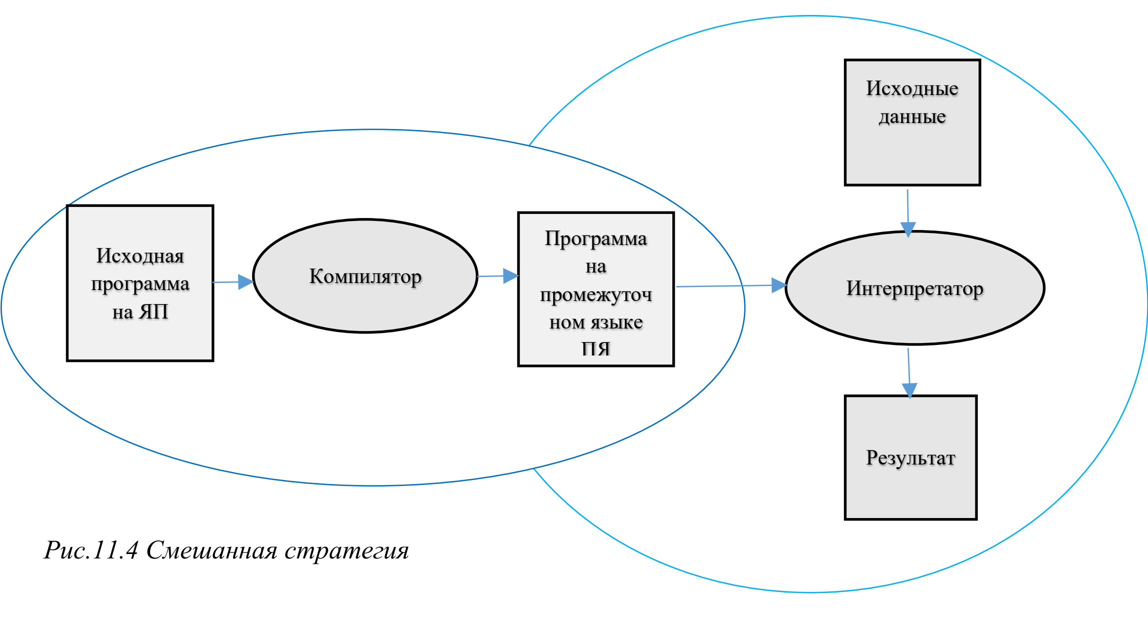
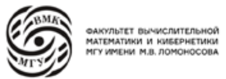


Рис.11.4 Смешанная стратегия



Примеры:

1. Язык Java JVM -- > байт-код -- > (Java Virtual Machine)
2. Языки на технологии NET (Basic, C#, C++) -- > промежуточный язык: CIL -- > Common Intermediate Language -- > JIT-компилятор (just-in-time)

Схема функционирования компилятора

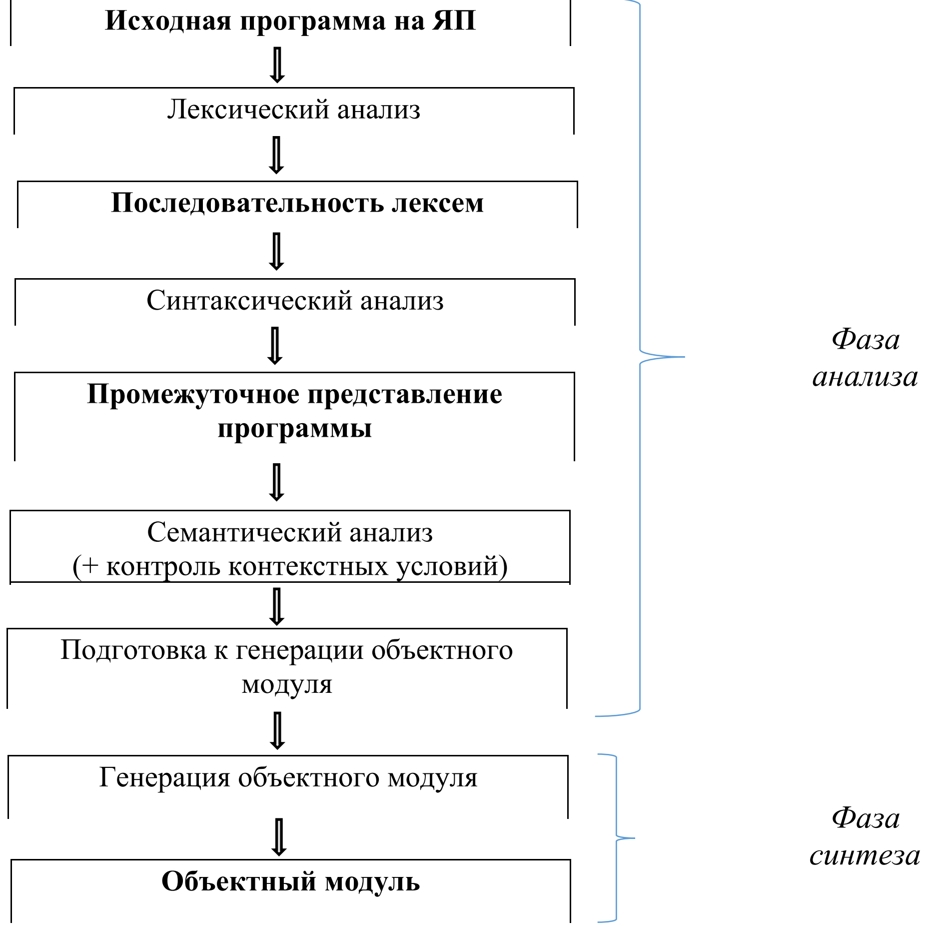
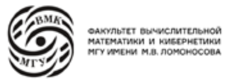


Рис.11.5 Схема функционирования компилятора

Компилятор всегда выделяет слева направо. В результате синтаксического анализа получается промежуточное решение программы.
Далее происходит анализ семантический, т.к. не все свойства программы задаются синтаксисом (например, a: =b/0).
Далее происходит подготовка (может входить оптимизация) к генерации объектного модуля и генерация объектного модуля.



Основные понятия теории формальных языков

Познакомимся с теорией формальных языков для того, чтобы изучить те понятия в этой теории, которые пригодятся нам в трансляции.

Алфавит:

это конечное множество символов (обязательно не пустое).

Пример: V = {a, b, c}

Цепочка символов в алфавите V:

любая конечная последовательность символов этого алфавита.

Пример: V = {a, b, c} Цепочка: abbba

Пустая цепочка:

цепочка, которая не содержит ни одного символа.

Обозначение: ε

(иногда для этой цели используется символ Λ).

Конкатенация (сцепление) цепочек α и β:

цепочка, полученная приписыванием последовательности β справа к α

Обозначение: $\alpha \cdot \beta$ (или $\alpha\beta$)

Пример: если α =ab и β = cd, то $\alpha\beta$ = abcd

1. Для любой цепочки α верно $\alpha\varepsilon = \varepsilon\alpha = \alpha$
   (Аналог для чисел: $\forall$ x верно $1 \cdot x = x$)

2. Операция конкатенации ассоциативна: $(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$
   для любых α, β, γ.

3. Операция конкатенации некоммутативна:
   например, для α =ab и β = cd $\alpha \cdot \beta \neq \beta \cdot \alpha$

   Обращение (или реверс) цепочка

   цепочка, символы которой записаны в обратном порядке

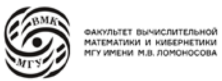
   Обозначение: α^R

   Пример: если α = abcdef, то α^R = fedcba

   Для пустой цепочки: $\varepsilon = \varepsilon^R$.

   Рекурсивное определение:

$$\alpha^R = \begin{cases} \varepsilon, \text{ если } \alpha = \varepsilon; \\ \beta^R s, \text{ если } \alpha = s\beta, \text{ где } s \text{ – символ алфавита} \end{cases}$$



n-я степень цепочки α (обозначается α^n)
конкатенация n цепочек α

$$\underbrace{\alpha\,\alpha\,\alpha\ldots\,\alpha\,\alpha}_{n\text{ раз}} = \alpha^n$$

Рекурсивное определение:

$$\alpha^N = \begin{cases} \varepsilon, \text{ если } n = 0; \\ \text{то } \alpha^{n-1}\,\alpha, \text{ если } n > 0 \end{cases}$$

Длина цепочки:

это число составляющих её символов

Пример: если α = abcdefg, то и α равна 7
Обозначение: | α |

| Ɛ | = 0

Рекурсивное определение:

$$|\alpha| = \begin{cases} 0, \text{ если } \alpha = \varepsilon; \\ |\beta|^{n+1}, \text{ если } \alpha = \beta s, \text{ где } s \text{ – символ алфавита} \end{cases}$$

Обозначение | α |, используется для числа вхождений символа s в цепочку α.

Язык в алфавите β – это множество цепочек конечной длины в этом алфавите. Каждая цепочка может очень длинной, но она всегда конечна, но при этом множество цепочек может быть бесконечным.

β* означает то множество, которое содержит все цепочки конечной длины в алфавите β, включая пустую цепочку.

Способы описания языков

Явное перечисление:

L = {ab, abb, ba, baa}
Язык L конечный.

Формулы:
$L = \{a^n\, b^n \mid n \geq 0\}$
Цепочки языка L: Ɛ, ab, aabb, aaabbb…

Порождающие грамматики Хомского (предложил формализм для описания языков).

Распознаватели (МТ, ЛОА, конечные автоматы, МП-автоматы).

Декартово произведение множеств A и B:
множество {(a, b) | a ε A, b ε B}
Обозначение: A x B

Порождающая грамматика G:
это четвёрка {T, N, P, S}, где

- T – алфавит *терминальных символов (терминалов),*

- N – алфавит *нетерминальных символов (нетерминалов),* непересекающихся с T

- P – конечное подмножества (T ᴗ N) * x (T ᴗ N) *;
элемент (α, β) множества P называется *правилом вывода* и записывается в виде α –> β, α содержит хотя бы один нетерминал.

- S – *начальный символ* (цель) грамматики, S ε N
правила
$\alpha \to \beta_1 \quad \alpha \to \beta_2 \dots \alpha \to \beta_n$
записываются сокращённо
$\alpha \to \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

β_1 для i = 1, 2, …..n – ***альтернатива*** правила вывода из цепочки α.

Порождающая грамматика G:
это четвёрка {T, N, P, S}, где

- T – алфавит *терминальных символов (терминалов),*
- N – алфавит *нетерминальных символов (нетерминалов),*
$T \cap N = \varnothing$,

- P – конечное подмножества (T ᴗ N) * x (T ᴗ N) *;
элемент (α, β) множества P называется *правилом вывода* и записывается в виде α –> β, α содержит хотя бы один нетерминал.

- S – *начальный символ* (цель) грамматики, S ε N

Пример грамматики: G_1 = ({0, 1}, {A, S}, P, S)
P:

S –> 0A1
0A –> 00A1

A –> ε

Пусть β ε (T ᴗ N) * α ε (T ᴗ N) *

β ***непосредственно выводима*** из α в грамматике G = {T, N, P, S},

Если α = $Ɛ_1 γ Ɛ_2$

β = $Ɛ_1$ δ $Ɛ_2$

где $Ɛ_1$, $Ɛ_2$, δ ε (T ᴗ N) *, γ ε (T ᴗ N) * и γ –> δ ε P.

Обозначение: α –> β

Пример: G_1 = ({0, 1}, {A, S}, P, S)
P:

S –> 0A1
0A –> 00A1

A –> ε

Цепочка 00A11 непосредственно выводима из <u>0A</u>1 в грамматике G_1: 0A1–> <u>00A</u>11 (по правилу 0A –> 00A1).

Цепочка β ε (T ᴗ N) * ***выводима*** из цепочки α ε (T ᴗ N) *
в грамматике G = {T, N, P, S},

если существуют цепочки $γ_0$, $γ_1$, …., $γ_n$ (n ≥ 0), такие, что

$α = γ_0$ –> $γ_1$ –> ….–> $γ_{n1} = β$

Обозначение: α => $β_1$

Вывод длины n:

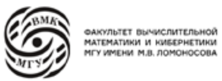
последовательность $γ_0$, $γ_1$, …., $γ_n$

Любая цепочка выводится сама из себя за 0 шагов: α = $γ_0$ = α

G_1 = ({0, 1}, {A, S}, P, S)
P:

S –> 0A1
0A –> 00A1

A –> ε



S => 000A111 в G_1, т.к. Ǝ вывод S –> 0A1–> 00A1–> 000A111
длина вывода равна 3.

Сентенциальная форма в грамматике G = (T, N, P, S):
α ε (T ᴗ N) *, для которой S => α

Язык, порождаемый грамматикой G = (T, N, P, S):
множество L (G) {α ε T *| S => α}

Грамматика – это не алгоритм, а система правил подстановки, позволяющих строить выводы.

Рассмотрим пример:
Где маленькие латинские буквы (а) означают символы терминального алфавита,
а большие (S) латинские буквы означают нетерминалы,
первый символ, который стоит в строчке слева будет считаться начальным.

S –> A | B // можно заменить S на A или на B
A –> a A | Ɛ // можно строить различные выводы: заменить A на Ɛ или a A
B –> a

Итак, ***грамматика*** G — это свод правил/ законов, по которым можно действовать.

Язык, порождаемый грамматикой G = (T, N, P, S):
множество L (G) {α ε T *| S => α}

Пример:

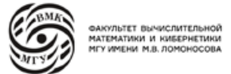
G_1 = ({0, 1}, {A, S}, P, S)
P:

(1) S –> 0A1
(2) 0A –> 00A1
(3) A –> ε

L (G) =?

S $\to_{(1)}$ 0A1 $\to_{(3)}$ 01
S $\to_{(1)}$ 0A1 $\to_{(2)}$ 00A11 $\to_{(3)}$ 0011
S $\to_{(1)}$ 0A1 $\to_{(2)}$ 00A11 $\to_{(2)}$ 000A111 $\to_{(3)}$ 000111

Предположительно: L = $\{0^n 1^n \mid n > 0\}$



Грамматики эквивалентны если их языки равны.
Эквивалентность грамматик G_1 и G_2
$L(G_1) = L(G_2)$
Пример:

$G_1 = (\{0, 1\}, \{A, S\}, P, S)$ и $G_1 = (\{0, 1\}, \{A, S\}, P, S)$

P_1: S –> 0A1 P_2: S –> 0S1 | 01
0A –> 00A1
A –> ε

$L(G_1) = L(G_2) = \{0^n 1^n \mid n > 0\}$

Грамматики G_1 и G_2 ***почти эквивалентны,***
если $L(G_1) \cup \{\varepsilon\} = L(G_2) \cup \{\varepsilon\}$.

Пример:

$G_1 = (\{0, 1\}, \{A, S\}, P_1, S)$ и $G_2' = (\{0, 1\}, \{A, S\}, P_2', S)$

P_1: S –> 0A1 P_2': S –> 0S1 | ε // входит пустая цепочка
0A –> 00A1
A –> ε

$L(G_1) = \{0^n 1^n \mid n > 0\}$, а $L(G_2') = \{0^n 1^n \mid n \geq 0\}$,
т.е. $L(G_1) = L(G_2') \cup \{\varepsilon\}$

Классификация грамматик и языков по Хомскому

Грамматики классифицируются по виду их правил вывода.
Четыре типа грамматик:

тип 0 (рекурсивно перечислимые),
тип 1 (контекстно-зависимые),
тип 2 (контекстно-свободные),
тип 3 (регулярные)

Язык, порождаемый грамматикой k (k = 0, 1, 2, 3), является языком *типа k.*

G = (T, N, P, S)
Тип 0

Любая порождающая грамматика является грамматикой *типа 0.*

На вид правил грамматик этого типа не накладывается никаких дополнительных ограничений.

Класс языков типа 0 совпадает с классом рекурсивно перечислимых языков (распознаваемых МТ- машинами Тьюринга).

Грамматики с ограничениями на вид правил ввода

Тип 1

Грамматика G = (T, N, P, S) называется *неукорачивающей*, если правая часть каждого правила из P не короче левой части (т.е. для любого правила α –> β є P выполняется неравенство |α | ≤ | β |).

В виде исключения в неукорачивающей грамматике допускается наличие правила S –> ε, при условии, что S (начальный символ) не встречается в правых частях правил.

Грамматикой типа 1 будем называть неукорачивающую грамматику.

Любую КС-грамматику можно преобразовать в эквивалентную неукорачивающую КС-грамматику (т.е. КС (контекстно-свободная), удовлетворяющую также и определению неукорачивающей).

Тип 3

Грамматика G = (T, N, P, S) называется *праволинейной*, если каждое правило из P имеет вид:

A –> wB либо A –> w, где A є N, B є N, w є T*.

Грамматика G = (T, N, P, S) называется *леволинейной*, если каждое правило из P имеет вид:

A –> Bw либо A –> w, где A є N, B є N, w є T*.

Праволинейные и леволинейные грамматики определяют один и тот же класс языков. Такие языки называются *регулярными.* Право- и леволинейные грамматики тоже называют регулярными.

Регулярная грамматика является грамматикой *типа 3.*

Автоматной грамматикой называется праволинейная (леволинейная) грамматика, такая, что каждое правило с непустой правой частью имеет вид:

A –> a либо A –> aB (для леволинейной, соответственно, A –> a либо A –> Ba), где A є N, B є N, a є T.

Для любой регулярной (автоматной) грамматики G существует неукорачивающая регулярная (автоматная) грамматика G′, такая, что L (G) = L (G′).

Иерархия Хомского

Справедливы следующие соотношения:
1. Любая регулярная грамматика является КС-грамматикой,
2. Любая неукорачивающая КС-грамматика является КЗ-грамматикой,
3. Любая неукорачивающая КС-грамматика является грамматикой типа 0.

Неукорачивающие Регулярные ⊂ Неукорачивающая КС ⊂ КЗ ⊂ Тип 0
(Запись A ⊂ B означает, что A является собственным подклассом класса B).

Иерархия классов языков

Рис.11.6 Иерархия классов языков

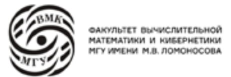
Тип 3 (Регулярные) ⊂ Тип 2 (КС) ⊂ Тип 1 (КЗ) ⊂ Тип 0

Проблема «Можно ли язык, описанный грамматикой типа k (k = 0, 1, 2, 3), описать грамматикой типа k +1?» является алгоритмически неразрешимой.
Язык L_{as} = {a, b}. Какого он типа? Обычно требуется указать максимально возможный тип.
Ответ: типа 3.

S –> a | b – грамматика типа 3, порождающая данный язык.
(L_{as} является также языком типа 2, 1, 0 в силу иерархии Хомского)

Примеры грамматик и языков

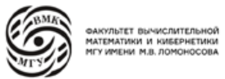
(1) S –> ABCS | ABc
BA –> AB
CA –> AC
CB –> BC
Cc –> cc



Bc –> bc
Bb –> bb
Ab –> ab
Aa –> aa

Тип 1. Неукорачивающая, но не КЗ
Язык: $\{a^n b^n c^n \mid n > 0\}$

(2) S –> ABCS | ab
Язык: $\{a^n b^n \mid n > 0\}$
(3) S –> aS | a
Язык: $\{a^n \mid n > 0\}$



Лекция 12. Трансляторы (2)

КС-грамматики позволяют выразить такие свойства языков программирования, как скобочные структуры, последовательность описаний и операторов и др. Но не могут задавать контекстно-зависимые свойства, например, соответствие числа формальных и фактических параметров при вызове функций.
Для КС-грамматик существуют эффективные алгоритмы анализа, поэтому они применяются в трансляции, контекстные условия проверяются на этапе семантического анализа.

Рис.12.1 Виды грамматик

Контекстно-свободные грамматики: основные черты - соответствие скобок, операторные скобки и пр. должны соответствовать друг другу.
Регулярные языки используются для этапа лексического анализа.

Левый (левосторонний) вывод цепочки β є T* из S є N в КС-грамматике G = (T, N, P, S):

в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого левого нетерминала.

Правый (правосторонний) вывод цепочки β є T* из S є N в КС-грамматике G = (T, N, P, S):

в этом выводе каждая очередная сентенциальная форма получается из предыдущей заменой самого правого нетерминала.

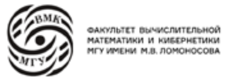
Рассмотрим пример грамматики:
G = ({a, b,+}), {S, T}, {S –> T | T = S; T –> a | b}; S)

Можно построить выводы для цепочки a+b+a:

(1) S –> T+S –>T+T+S –> T+T+T –> a+T+T –> a+b+T –> a+b+a

(2) S –> T+S –>a+S –> a+T+S –> a+b+S –> a+b+T –> a+b+a

(3) S –> T+S –>T+T+S –> T+T+T –> T+T+a –> T+b+a –> a+b+a



Здесь (2) – левосторонний вывод, (3) – правосторонний, в (1) не является ни левосторонним, ни правосторонним.

Определение: упорядоченное ориентированное дерево называется **деревом вывода** (или **деревом разбора**) в КС-грамматике G = (T, N, P, S), если выполнены следующие условия:

1. каждая вершина дерева помечена символом из множества N ᴗ T {ε}, при этом корень дерева помечен символом S, листья – символами из T {ε};

2. если вершина дерева помечена символом A, а её непосредственные потомки – символами a_1, a_2, …a_n, где каждое a_i є T ᴗ N, то A–> a_1 a_2…a_n – правило вывода в этой грамматике;

3. если вершина дерева помечена символом A, а её единственный непосредственный потомок помечен символом ε, то A –> ε – правило вывода в этой грамматике.

Пример:
Дерево вывода для цепочки a+b+a в грамматике
G = ({a, b,+}), {S, T}, {S –> T | T = S; T –> a | b}; S)

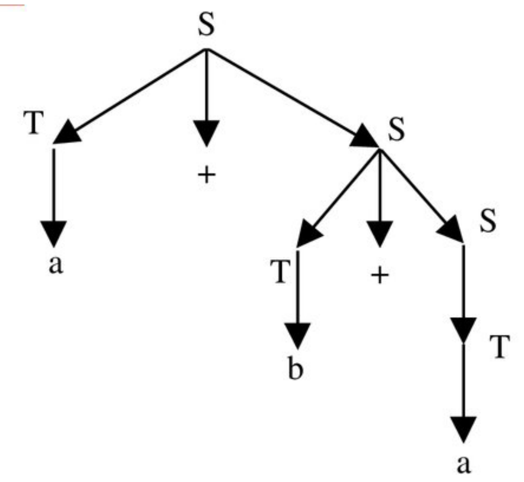


Рис.12.2 Пример

(1) S –> T+S –>T+T+S –> T+T+T –> a+T+T –> a+b+T –> a+b+a

(2) S –> T+S –>a+S –> a+T+S –> a+b+S –> a+b+T –> a+b+a

(3) S –> T+S –>T+T+S –> T+T+T –> T+T+a –> T+b+a –> a+b+a

Существуют грамматики цепочек, для которых есть два различных дерева вывода. Такие грамматики называются *неоднозначными.*
Неоднозначность естественна для естественных языков.

Утверждение: Проблема определения, является ли заданная КС-грамматика однозначной, является **алгоритмически неразрешимой**.
Язык, порождаемый грамматикой, называется *неоднозначным*, если он не может быть порождён никакой однозначной грамматикой.

Утверждение: Проблема определения, порождает ли данная КС-грамматика однозначный язык (т.е. существует ли эквивалентная ей однозначная грамматика), является **алгоритмически неразрешимой**.

Пример: неоднозначного языка

$L = \{a^n b^n c^m \mid n > 0, m > 0\} \cup \{a^n b^m c^m \mid n > 0, m > 0\}$

Контекстно-свободный класс грамматик

Рассмотрим *пример*, где маленькими буквами - терминалы, большими буквами указаны нетерминалы, S – начало строки.

Вопрос: какой язык порождается данной грамматикой?

S –> A | B
B –> b | BC
C –> CC
CB –> BC
D –> a // D не участвует ни в одном выводе

Видно, что это грамматика. Можно отказаться от D –> a, отказаться от *A* в A | B,
Также отказаться от C –> CC и *BC* в b | BC.

Приведённые КС-грамматики

Символ x є (T ᴗ N) называется *недостижимым* в грамматике G = (T, N, P, S), если он не появляется ни в одной сентенциальной форме этой грамматики.

Символ A є N называется *непорождающим* (или *бесплодным*) в грамматике G = (T, N, P, S), если множество выводимых из этого символа терминальных цепочек пусто.

КС-грамматика называется приведённой, если в ней нет недостижимых и бесплодных символов.

Алгоритм привидения грамматики:

1. Найти и удалить все бесплодные символы и правила их содержащие.
2. Найти и удалить все недостижимые символы и правила их содержащие.

Примечание: если начальный символ грамматики окажется бесплодным, то следует удалить содержащие его правила, а сам символ оставить в алфавите нетерминалов N, так как по определению грамматики N обязан содержать начальный символ.

Для нахождения бесплодных и достижимых символов полезен граф КС-грамматики:

- Каждому символу T c N соответствует единственная вершина, помеченная этим символом: если в P есть правило с пустой правой частью ε, то граф имеет вершину, помеченную ε;
- Вершина X соединяется с вершиной Y стрелкой (дугой), если в грамматике есть правило X –> αYβ, α, β є (T ᴗ N) *;
- X соединяется с вершиной ε, если в грамматике есть правилоX –> ε.

Алгоритм удаления бесплодных символов

1. Отметить терминальные вершины (вершины, помеченные терминальными символами), а также вершину ε, если таковая имеется.
2. Если в P есть правило A –> α, где α состоит из уже отмеченных в графе символов, а вершина A не отмечена, то отменить эту вершину. Повторять шаг 2 пока возможно.
3. Из грамматики удалить неотмеченные символы и правила их содержащие.

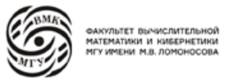
Алгоритм удаления недостижимых символов

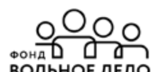
1. Отметить вершины, в которые есть путь из вершины S (достижимы из вершины S).
2. Удалить из грамматики неотмеченные символы и правила их содержащие.

Пример: Дана грамматика

G = ({a, b, c, d, e}, {S, A, B, C, D} P, S)
P:

S –> aAB | C
D –> cDc | d
C –> aCD
A –> aA | a | ε





B –> b

Удалив из G бесплодные символы, получим эквивалентную грамматику

G_1 = ({a, b, c, d, e}, {S, A, B, D} P_1, S)
P_1:

S –> aAB
D –> cDc | d
A –> aA | a | ε
B –> b

G_1 не содержит бесплодных символов.

Далее находим недостижимые символы.

G_1 = ({a, b}, {S, A, B} P_1, S)
P_1:

S –> aAB
A –> aA | a | ε
B –> b

Неотмеченные символы являются недостижимыми.
Удалив из G_1 недостижимые символы, получим эквивалентную грамматику:

G_2 = ({a, b}, {A, B} P_2, S)
P_2:

S –> aAB
A –> aA | a | ε
B –> b

G_2 **- приведённая грамматика**

L (G) = L (G_1) = L (G_2) = {a^n b | n ≥1}

Задача.
Убедиться, что если в рассмотренном выше примере поменять местами шаги (1) и (2) алгоритма приведения грамматики, то результатом будет непереведённая грамматика.

Устранение правил с пустой правой частью из КС-грамматики

1. Построить множество X = {A є N | A=> ε}.
2. Удалить правила с пустой правой частью.
3. Если S є X, то S'– новый начальный символ, S'–> S| ε є P.
4. ∀ A_1 є X правило вида B –> α_1 A_1 α_2 A_2…$\alpha_n A_n \alpha_{n+1}$, где α_1 є ((N-X) ᴗ T) * заменить 2^n правилами, соответствующими всем возможным комбинациям вхождения A между α_i;

$B \to \alpha_1\alpha_2\ldots\alpha_n\alpha_{n+1}$

$B \to \alpha_1\alpha_2\ldots\alpha_n A_n\alpha c$

…..

$B \to \alpha_1\alpha_2A_2\ldots\alpha_n A_n\alpha_{n+1}$

$B \to \alpha_1 A_1\alpha_2A_2\ldots\alpha_n A_n\alpha_{n+1}$

Замечание: Если все $a_i = \varepsilon\ \forall\ i = 1, \ldots, n+1$, то правило $B \to \varepsilon$ не включать в новую грамматику.

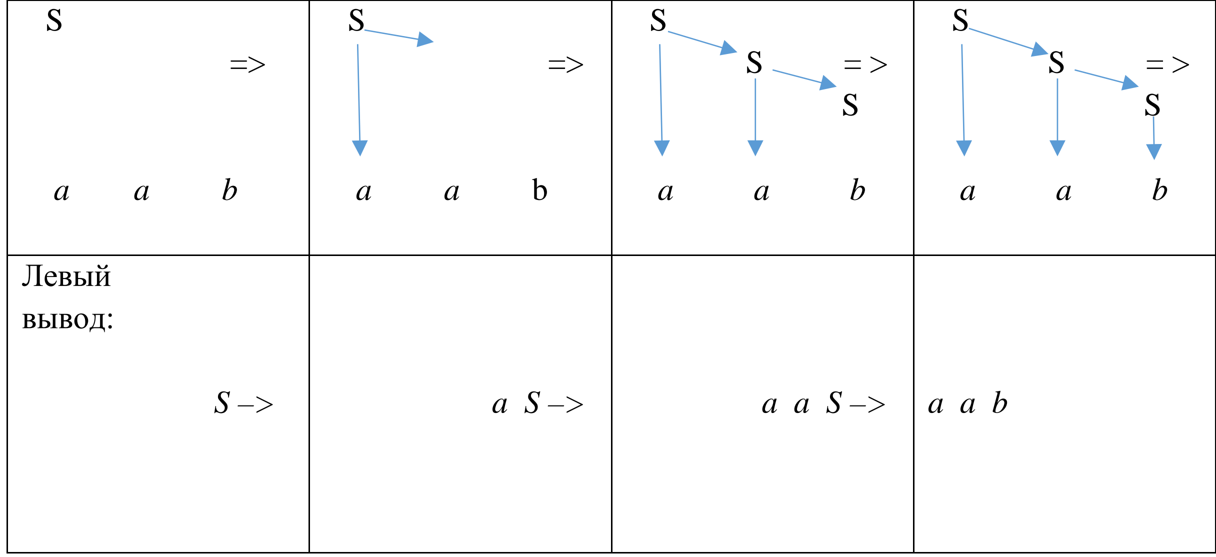
5. Удалить бесполезные символы и правила их содержащие.

Пример:

исходная грамматика		эквивалентная грамматика	
	$S \to C \mid b \mid Ab$		$S \to C \mid b \mid Ab$
	$B \to \varepsilon$		$C \to c$
	$C \to c$		$A \to Aa \mid a$
	$A \to Aa \mid \varepsilon$		

Мы рассмотрели основные понятия теории формальных языков, что даёт математическую базу (ТФЯ) для изучения основ трансляции.

ТФЯ является одной из старейших и наиболее фундаментальных областей информатики.



S => *a a b*	S => *a a* b	S S => S *a a b*	S S S *a a b*
Левый вывод: *S* –>	*a S* –>	*a a S* –>	*a a b*

Таблица 12.3 Правило вывода (свёртка)

Свёртка – это применение правила вывода «в обратную сторону», замена правой части на нетерминал из левой части:

aab <– *aaS* – свёртка по правилу S –>*b*. Обозначаем свёртку с помощью обратной стрелки <–.

С помощью свёрток можно построить вывод «задом наперёд» (обращение вывода): от цепочки к цели грамматики S. Например, сентенциальную форму *aaS* можно свернуть к *aS*, а затем к *S:*

aab <– aaS <– aS<– S

<u>Построение дерева вывода (2)</u>

(2) G: S –> a S | b Цепочка: aab

Снизу вверх:

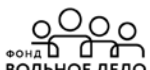
S => *a* *a* *b*	S => S *a* *a* b	S S => S *a* *a* *b*	S S => S *a* *a* *b*
Обратный правый вывод: *a a b* <–	*a a S* <–	*a S* <–	*S*

Таблица 12.4 Дерево вывода снизу вверх

Регулярные языки

Способы описания:

- Регулярные грамматики
  (леволинейные либо праволинейные)
- Конечные автоматы
  (недетерминированные или детерминированные)
- Регулярные выражения
  (см. материал «О регулярных языках» на сайте cmcmsu.info)



Недетерминированный конечный автомат (НКА) – это пятёрка A = (K, Σ, δ, I, F), где:

K— конечное множество состояний или вершин;
Σ— входной алфавит (также конечный);
$\delta \subseteq K$ xΣ xK — множество команд или дуг;
$I \subseteq K$ — множество начальных состояний;
$F \subseteq K$ — множество заключительных состояний.

Множество δ можно также интерпретировать как отображение K x Σ в множестве K.

Если A = (K, Σ, δ, I, F) – это НКА, то каждая дуга НКА A имеет пометку из Σ.
Путь в ориентированном графе может быть представлен последовательностью дуг.

Пустой путь можно представить одной вершиной, которая считается одновременно и началом, и концом дуги.

Пометка пути – это сцепление (конкатенация) пометок его дуг. Пустой путь имеет пустую пометку. Путь из начальной вершины в заключительную называется *успешным.*

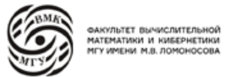
Язык, допускаемый автоматом *A* (обозначается L (A)), – это множество пометок всех успешных путей автомата.

Алгоритм построения НКА по праволинейной автоматной грамматике

1. Множество вершин НКА состоит из нетерминалов грамматики и, возможно, ещё одной вершины F, которая объявляется заключительной.

2. Каждому правилу вида *A* –> aB в автомате соответствует дуга из вершины *A* в вершину B, помеченная символом α:
$A \overset{\alpha}{\longrightarrow} B$. Каждому *A* –> a соответствует дуга $A \overset{\alpha}{\longrightarrow} F$. Других дуг нет.

3. Начальной вершиной автомата является вершина, соответствующая начальному символу грамматики. Заключительными являются новая вершина F, если она использовалась на шаге 2, и каждая вершина *A* такая, что для нетерминала *A* в грамматике есть правило *A* –> ε.



Пример 1: (НКА по праволинейной грамматике)

$S \longrightarrow aS$
$S \longrightarrow bA$
$A \longrightarrow bA$
$A \longrightarrow \varepsilon$

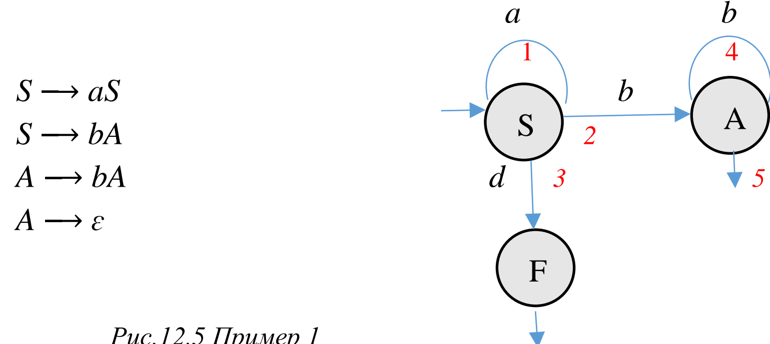


Рис.12.5 Пример 1

$S \longrightarrow_1 aS \longrightarrow_2 abA \longrightarrow_4 abbA \longrightarrow_5 abbA$ - вывод в грамматике *abb*

$\longrightarrow S \xrightarrow{\alpha}{}_1 S \xrightarrow{b}{}_2 A \xrightarrow{b}{}_4 A \longrightarrow_5$ - соответствующий путь в автомате для *abb*

Алгоритм построения праволинейной автоматной грамматике по НКА с единственной начальной вершиной

1. Нетерминалами грамматики будут вершины автомата, терминалами – пометки дуг.
2. Для каждой дуги $A \xrightarrow{\alpha} B$ в грамматику добавляется правило *A* –> aB. Для каждой заключительной вершины *B* в грамматику добавляется *B* –> ε.
3. Начальным символом будет нетерминал, соответствующий начальной вершине.
4. К построенной по пунктам 1-3 грамматике можно применить алгоритм устранения ε-правил.

Алгоритм построения НКА по леволинейной автоматной грамматике

1. Множество вершин НКА состоит из нетерминалов грамматики и, возможно, еще одной новой вершины *H*, которая объявляется начальной.
2. Каждому правилу вида *A* –> *B*a в автомате соответствует дуга из вершины B в вершину *A*, помеченная символом α:
   $B \xrightarrow{\alpha} A$. Каждому *A* –> a соответствует дуга $H \xrightarrow{\alpha} A$. Других дуг нет.
3. Заключительной вершиной автомата является вершина, соответствующая начальному символу грамматики. Началльными являются новая вершина *H*, если она использовалась на шаге 2, и каждая вершина *A* такая, что для нетерминала *A* в грамматике есть правило *A* –> ε.

Пример 2: (НКА по леволинейной грамматике)

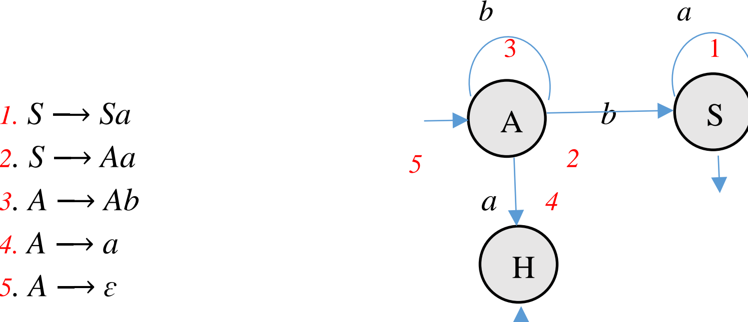


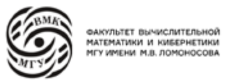
Рис.12.6 Пример 2

$aba \leftarrow_4 Aba \leftarrow_3 Aa \leftarrow_2 S$ - обращение вывода в грамматике *abb (свёртки)*

$\longrightarrow H \xrightarrow{\alpha}_4 A \xrightarrow{b}_3 \mathrm{A} \xrightarrow{\alpha}_2 \mathrm{S} \longrightarrow$ - соответствующий путь в автомате для *aba* *(моделирует свёртки)*

Алгоритм построения леволинейной автоматной грамматике по НКА с единственной заключительной вершиной

1. Нетерминалами грамматики будут вершины автомата, терминалами – пометки дуг.
2. Для каждой дуги $A \xrightarrow{\alpha} B$ в грамматику добавляется правило *B* –> *A*a. Для каждой начальной вершины *B* в грамматику добавляется *B* –> ε.
3. Начальным символом будет нетерминал, соответствующий заключительной вершине.
4. К построенной по пунктам 1-3 грамматике можно применить алгоритм устранения ε-правил.



Лекция 13. Регулярные языки

Диаграмма состояний для грамматики G – это граф, представляющий конечный автомат, построенный нашим алгоритмом по грамматике.

Представление в виде набора команд или функции переходов

(C,a) –> A или $\delta(C,a)=\{A\}$
(C,b) –>B или $\delta(C,b) = \{B\}$
(C, ⊥) –> S или $\delta(C, \perp) =\{S\}$
(A,b) –> C или $\delta(A, B)=\{C\}$

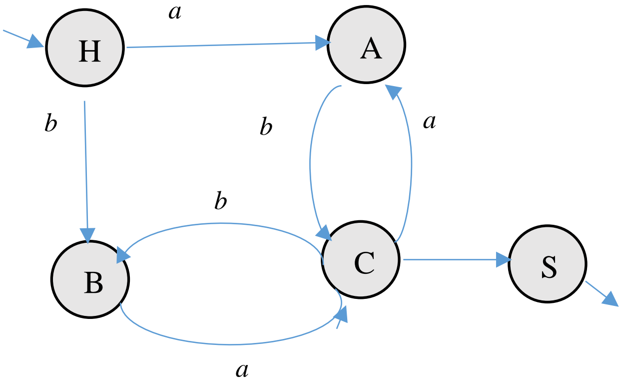


Рис.13.1 Диаграмма состояний

G = ({a, b,⊥}, {S, A, B, C}, P, S), где
P:

S –> C⊥
C –> Ab | Ba
A –> a | Ca
B –> b | Cb

Представление в виде таблицы

	a	**b**	**⊥**
H	A	B	-
C	A	B_1	S
A	-	C	-
B	C	-	-
S	-	-	-

Конечный автомат называется **детерминированным** конечным автоматом (ДКА), если он имеет единственное начальное состояние, и любые две дуги, исходящие из одной и той же вершины имеют различные пометки.

Множество δ в ДКА можно интерпретировать как отображение K xΣ в множестве K.

Тогда конечный автомат допускает цепочку $a_1\ a_2\ \ldots.\ a_n$, если δ (H, a_1) = A_1; δ (A_1, a_2) = A_2; ….; δ (A_{n-2}, a_{n-1}) = A_{n-1}; δ (A_{n-1}, a_n) = S, где $a_1 \in \Sigma$, $A_1 \in K$, j 1, 2 ….,n-1: i = 1, 2 , …., *n: H – начальное состояние* S – одно из заключительных состояний.

Язык, допускаемый ДКА – это множество всех допускаемых им цепочек.

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

Пример:

A’:

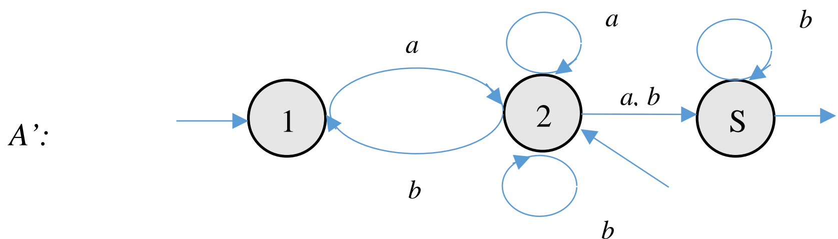


Рис.13.2 Пример построения ДКА

Процесс построения ДКА по заданному НКА удобно изобразить в виде таблицы, начав с состояния {1, 2}. Затем заполняем строки для вновь появляющихся состояний.

символ / **состояние**	*a*	*b*
{1, 2}	{2, 3}	{1, 2, 3}
{2, 3}	{2, 3}	{1, 2, 3}
{1, 2, 3}	{2, 3}	{1, 2, 3}

Вход - {1, 2}; выход - **{2, 3}** и **{1, 2, 3}.**

Обозначим состояние {1, 2} через A, {2, 3} – B, {1, 2, 3}– C.

символ / **состояние**	*a*	*b*
A	B	C
B	B	C
C	B	C

С учётом переобозначений построим по таблице ДКА A*:*

L (A) = {a, b}

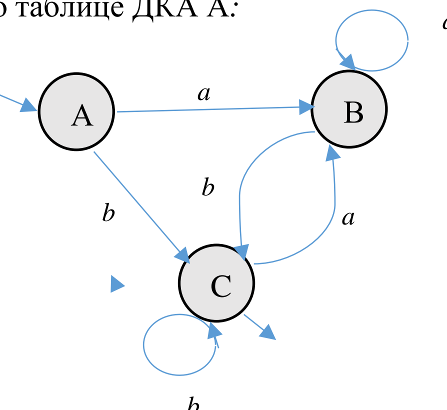


Рис.13.3 Пример построения ДКА A

Можно заметить, что язык *L = {a, b}*, допускаемый автоматом *A*, допускается также ДКА *A''*, имеющим только два состояния:

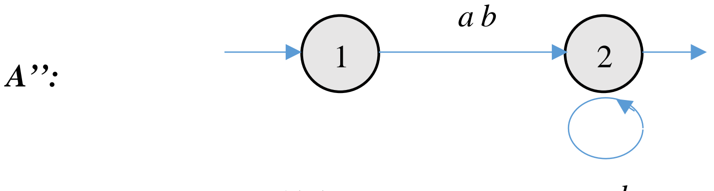


Рис.13.4 Пример построения ДКА A"

Существует алгоритм, позволяющий по любому ДКА построить эквивалентный ДКА с минимальным числом состояний.

Алгоритм минимизации конечного автомата

Вход: КА A_1

Выход: A_{mi}: $L(A_{min}) =: L(A_1)$ с минимальным числом состояний

Метод: $A_{min} = det(rev(det(rev(A_1))))$

det ( ) – детерминизация
rev ( ) – обращение автомата (выходы делаем входами, а входы – выходами)

Алгоритм построения ДКА по НКА

Вход: *A= (K', Σ, δ', I, F)* – НКА.
Выход: *A= (K', Σ, δ, InnitState, FinalStates)* – ДКА. Автомата
Метод: Вершинами (состояния) автомата *A* будут подмножества множества *K'* автомата *A'*. *CurState и NewState* – вспомогательные переменные для хранения таких подмножеств. Сам алгоритм запишем в паскалеподобном стиле. Фигурные скобки означают конструкторы множеств.

```
begin InnitState: = {s | s є 1}: K : = {InnitState}: δ : = ∅;
    while (в K есть нерассмотренный элемент)
    begin
        CurState – нерассмотренный элемент из K;
        for (каждого a є Σ)
        begin
```

$NewState := \{q \mid (p \xrightarrow{a} q\} \in \delta, p \in CurState;$
$K := K \cup \{ NewState \}$:
$\delta := \delta \cup \{(CurState \xrightarrow{a} NewState)\}$;
end
end:
FinalStates :={P є K существует q є P: q є F}
end.

Примечание: Получаемый ДКА будет всюду определённым, он умеет дочитать до конца любую цепочку. Тупиковое состояние ∅ (состояние ошибки) и связанные с ним дуги при необходимости можно удалить.

Для более удобной работы с диаграммами состояний пользуемся следующими соглашениями:

1. Если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то будем изображать одну дугу, помеченную всеми этими символами;

2. Непомеченная дуга будет соответствовать переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния;

3. Для любого автомата существует состояние ошибки (ERR); переход в это состояние означает, что исходная цепочка языку не принадлежит.

Алгоритм моделирования работы ДКА

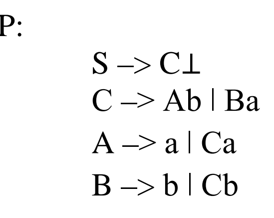
Вход: ДКА $A = (K', \Sigma', \delta', I, F)$ и цепочки x ⊥, где $x \in \Sigma^*$, $\perp \notin \Sigma$ – маркер конца цепочки.
Выход: *«Да»,* если x є L *(A),* иначе – «Нет»
Метод: Введем переменные *St* для хранения текущего состояния автомата и *c* для хранения очередного считанного символавходной цепочки x.

begin
c: = первый символ цепочки x;
St: = I {начальное состояние}
while ($St \neq$ ERR $and\ c \neq {'}\perp{'}$)
begin
St: =δ(St, c);
c: = очередной символ
end:
if St є F ***then***

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

write ('Да')
else
write (' Нет ')
end:

Пример **анализатора** для грамматики G = ({a, b, ⊥}, {S, A, B, C}, P, S), где


P:
S –> C⊥
C –> Ab | Ba
A –> a | Ca
B –> b | Cb

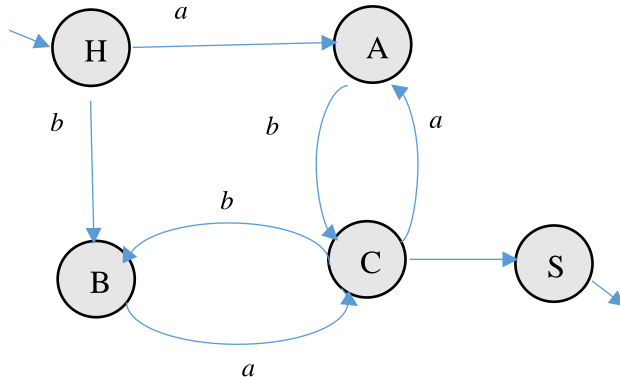


Рис.13.5 Пример анализатора

Программа-анализатор на С++

```
#include <iostream>
char c;                              //      текущий символ
void gc ( ) {std: : cin >> c;}       //      считать очередной символ со входа
bool scan_G ( ) {
enum state {H, A, B. C. S, ERR};     //      множество состояний
state CS;                            // CS – текущее состояние
CS= H;                               // начинаем с начального состояния H
gc ( );                              // считать первый символ
do {switch (CS)  {
        case H:   if (c == 'a') {gc (); CS = A;}
                  else if (c == 'b') {gc (); CS = B;}
                          else CS = ERR;
                  break;
        case A:   if (c == 'b') {gc (); CS = C;}
                          else CS = ERR;
                  break;
        case B:   if (c == 'a') {gc (); CS = C;}
                          else CS = ERR;
                  break;
        case C:   if (c == 'a') {gc (); CS = A;}
                  else if (c == 'b') {gc (); CS = B;}
```


ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

```
                        else if (c = = '⊥') CS = C;
                             else CS = ERR;
                   break;
                        }
        } while (CS! = S δδ CS! = ERR);
if (CS == ERR)
        return false;
else
                   return true;
```

Пример разбора цепочки *abba* ⊥

$H \xrightarrow{\alpha} A \xrightarrow{b} C \xrightarrow{b} B \xrightarrow{\alpha} C \xrightarrow{\perp} S$

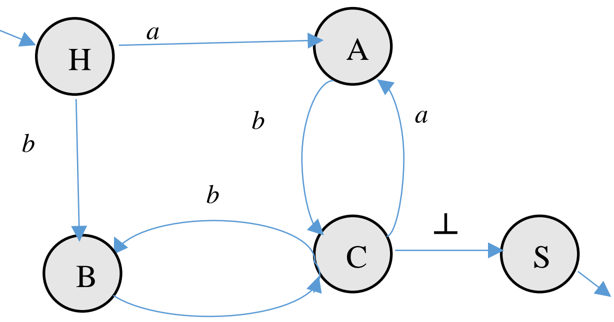


Рис.13.6 Пример разбора цепочки abba ⊥

$abba \perp \leftarrow Abba \perp \leftarrow Cba \perp \leftarrow Ba \perp \leftarrow C \perp \leftarrow \boldsymbol{S}$

S => a b b a ⊥	S => A a b b a ⊥	S = > C A a b b a ⊥
S B C => A => a b b a ⊥	S C B C => A => a b b a ⊥	S C B C => A => a b b a ⊥

Таблица 13.7

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

ФОНД ВОЛЬНОЕ ДЕЛО

teach-in ЛЕКЦИИ УЧЕНЫХ МГУ

Недетерминированный разбор

Если конечный автомат, построенный по грамматике, не является детерминированным, то нужно перебирать все возможные варианты переходов. Можно также преобразовать его в эквивалентный ДКА и проводить детерминированный разбор.

Пример **использования автоматов в решении теоретических задач**

Утверждение: *Контекстно-свободный язык*

$$L = \{a^n b^n \mid n \geq 1\}$$

нерегулярен

(доказательство см .на сайте cmcmsu.info.download.formal grammars and languages.2009.pdf)

Лексический анализ (ЛА) – это первый этап процесса компиляции. На этом этапе литеры, составляющие исходную программу, группируются в отдельные элементы, называемые лексемами.

Задачи лексического анализатора:

- выделить в исходном тексте цепочку символов, представляющую лексему, и проверить правильность её записи;

- зафиксировать в специальных таблицах для хранения разных типов лексем факт появления соответствующих лексем в анализируемом тексте;

- преобразовать цепочку символов, представляющих лексему, в пару:
(тип_лексемы указатель_на_информацию_о_ней);

- удалить проблемные литеры и комментарии.

Лексический анализатор для М-языка
Описание модельного языка

P → **program** DI; B ⊥
DI → **var** D {, D}
D → I {, I}: [**int** | **bool**]
B → **begin** S {; S} **end**
S → I: = E | **if** E **then** S **else** E | **while** E **do** S | B | **read** (I) | **write** (E)
E → EI [= | < | > |<= | =>|! =] EI | EI
EI → T {[+ | - | **or**] T}
T → F {[* | / | **or**] T}

F → I | N | L | ***not*** F | (E)
L → **true | false**
F → a | b | … | z | Ia | Ib | … | Iz | 10 | 11 | … | I9
N → 0| 1 | … | 9 | N0 | N | … | N9 |

Контекстные условия:

1. Любое имя, используемое в программе, должно быть описано и только один раз.
2. В операторе присваивания типы переменной и выражения должны совпадать.
3. В условном операторе и в операторе цикла в качестве условия возможно только логическое выражение.
4. Операнды операции отношения должны быть целочисленными.
5. Тип выражения и совместимость типов операндов в выражении определяются по обычным (паскалевским) правилам; старшинство операций задано синтаксисом.

Проектирование структуры классов лексического анализатора М-языка

Представление лексем: выделим следующие типы лексем:

```
enum type_of_lex {LEX_NULL, /*0*/
        LEX_AND, LEX_BEGIN, …. LEX_WRITE, /*18*/
        LEX_FIN, /*19*/
        LEX_SEMICOLON, LEX_COMMA, … LEX_GEQ, /*35*/
        LEX_NUM, /*37*/
        POLIZ_LABEL, /*38*/
        POLIZ_ADDRESS, /*39*/
        POLIZ_GO, /*40*/
        POLIZ_FGO}; /*41*/
```

Соглашение об используемых таблицах лексем:

TW – таблица служебных слов М-языка

TD – таблица ограничителей М-языка

TID – таблица идентификаторов анализируемой программы.

Таблицы TW и TD заполняются заранее, т.к. их содержимое не зависит от исходной программы.

Таблица TID формируется в процессе анализа.

Ident	
-	***char*** * name;
-	***bool*** declare;
-	*type_of_lex_type;*
-	***bool*** assing;
-	***int value****;*

N (0....N)

Table_ident	
-	*Ident * p;*
-	***int*** ***size****;*
-	***int*** ***top****;*
+	*Ident &* ***operator*** *[] (int k);*
+	***int*** *put (****const char*** **buf);*

Lex	
-	*type_of_lex_type t_lex;*
-	***int*** v_lex;
+	*type_of_lex_type get_type ();*
+	***friend*** *ostream &* ***operator*** *<< (ostream* ***&****s, Lex l);*

Scanner	
-$	***char*** * *TW [];*
-$	*type_of_lex words [];*
-$	***char*** * *TW [];*
-$	*type_of_lex dlms [];*
+	*Scanner (****const char*** ** program);*
+	*Lex_get_lex ();*
-	*Look (****const char*** **buf,* ***char*** ** * list);*

Таблица 13.8 Проектирование структуры классов

Класс Lex

```
class Lex {
        type_of_lex_   t_lex;
        int            v_lex;
public:
Lex (type_of_lex_   t = LEX_NULL, int v = 0) {
                        t_lex = t; v_lex = v;
        }
type_of_lex_type  get_type  ( ) { return t_lex ;}
int               get _value ( ) { return v_lex ;}
friend ostream & operator << (ostream &s, Lex 1);
        {
        s << ‘ ( ‘ << 1.t_lex  ‘ , ‘  << 1.v_lex << “); “ ;
        return s;
        }
```

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

Класс Ident

```
class Ident {
        char * name;
        bool    declare;
type_of_lex_type;
        bool assing;
        int value;
public:
        Ident ( ) { declare = false; assign = false; }
        char * get_name ( )  {return name;}
        void    put_name (const char *n)
        {name = new char [strlen (n) + 1];
                                        strcpy (name, n);}
        bool get_declare ( )  {return declare;}
        void put_declare ( )  { declare = true; }
type_of_lex_type   get_type  ( ) { return type ;}
        void put_type    (type_of_lex t)  { type = t;}
        bool get_assing  ( ) { return assing ;}
        void put_assing ( ) { assing = true ;}
        int get_value  ( ) { return value ;}
        void put_ value (int v) {value = v;}
```

Класс table_ident

```
class table_ident {
        ident *p;
        int size;
        int top;
public:
        tabl_ident (int max_size)
        {p= new ident [size = max_size]; top = 1;}
        ~tabl_ident ( ) {delete [] p;}
        ident & operator [ ] int k) {return p [k];}
        int put (const char * buf);
};
int tabl_ident : : put (const char * buf) {)
        for (int j = 1; j < top; j ++)
          if (! strcmp (buf, p [j].get_name () ) ) return j;
        p [top]. put_name (buf); top ++;
        return top_1;
```

Класс Scanner

```
class Scaner {
        enum state {H, IDENT, NUMB, COM, ALE, DELIM, NEQ };
        static char * TW [ ];
        static type_of_lex words [];
        static char * TD [ ];
        static type_of_lex dlms [];
        state CS;
        FILE * fp;
        char  c;
        char buf {80};
        int  buf_top;
        void clear () {
                buf_top = 0;
                        for (int j = 0; j < 80; j++)
                                buf [j] = ‘ \ 0’;
void add ( ) {buf [buf_top ++] = c;}
int look (const char * buf, char ***list) {
        int i = 0;
        while (list [i])  {
          if (! strcmp (buf, list [i] )) return i;
                                        i++;
                                }
        return 0;
}
void gc ( ) { c = fgetc (fp); }
public:
        Scanner (const char * program) {
        fp = fopen (program, “r”); CS + H;
        clear ( ); gc ( );
}
Lex get_lex ( );
```

Таблицы М-языка

```
    char * Scanner:: TW [ ] =
  { NULL,"and","begin","bool","do","else","end",
//            1       2        3     4     5     6
    "if","false","int","not","or","program","read",
//   7      8      9    10    11      12        13
      "then","true","var","while","write“ };
//      14      15     16     17      18

char * Scanner:: TD [ ] = {NULL, ";", "@", ",", ":", ":=", "(", ")",
//                                1    2    3    4    5     6    7
                          "=","<", ">", "+", "-", "*", "/", "<=", ">="};
//                         8   9   10   11   12   13   14   15    16
```

tabl_ident TID (100);

Диаграмма состояний для лексического анализатора

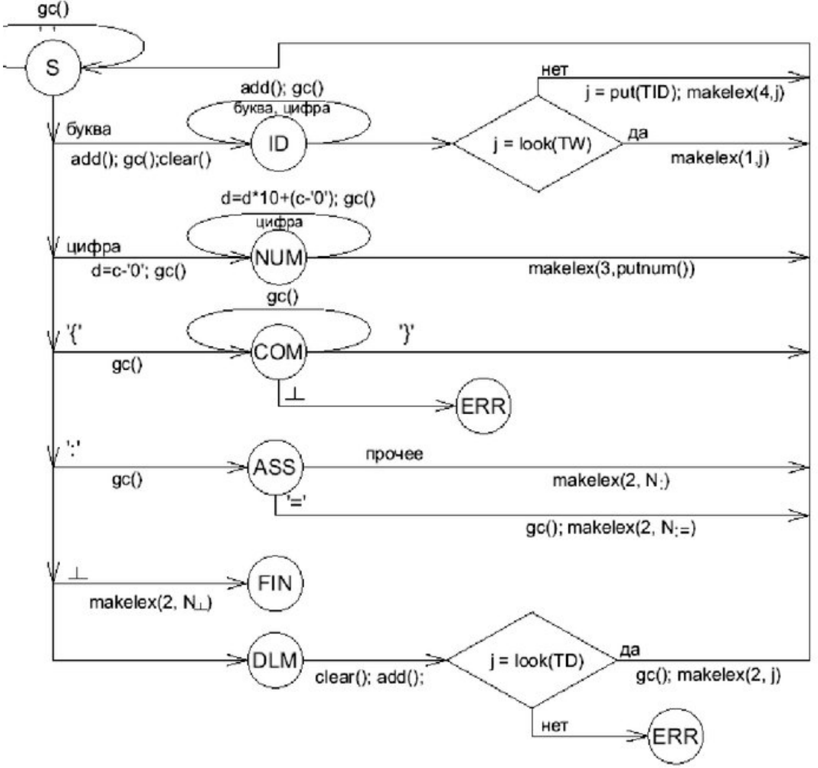
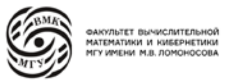


Рис.13.9 Диаграмма состояний для лексического анализатора



```
Lex Scanner : : get_lex ( ) {
        int d, j;
        CS = H;
do {
        switch (CS) {
        case H:
        if (c == ‘ ‘ / / c = = ‘\ n’ / / c = = ‘\ r’ / / c = = ‘\ t’ ) gc;
        else
        if ( isalpha  ( c) {clear ();  add ( ); gc ( ); CS = IDENT; }
        else
        if ( isdigit  ( c))  { d = c – ‘0’; gc ( ); CS = IDENT; }
        else
        if ( c == ‘:’ / / c == ‘<’ / /  c==’>’ )  {clear (); add ();gc ( ); CS =ALE; }
        else
        if ( c== ‘δ’) return Lex (LEX_FIN) // δ - символ конца программы
        else
        if ( c == ‘!’ )  {clear (); add ();gc ( ); CS =NEQ; }
        else CS = DELIM;
        break;
case NEQ:
        if ( c == ‘=’ )  {
                add (); gc ( ); j = look ( buf; TD);
                return Lex (LEX_NEQ, j) ; }
        else throw ‘!’;
        break;
case DELIM:
        clear (); add ();
        if ( j = look (buf; TD)) {
                gc ( ); return Lex ( dlms,[j], ) ; }
        else throw c;
        break;
  }        // end of switch
} while (true);
} // end ofgetlex ( )
```

Лекция 14. Регулярные языки (2)

Синтаксический анализ

Рассмотрим задачу разбора (синтаксический анализ)

Даны КС-грамматика G x
x є L (G)?

Если да, то построить дерево вывода для x (или левый вывод для x, или правый вывод для x).

Существуют различные методы синтаксического анализа для КА-грамматик:
- для некоторых подклассов есть эффективные методы, затрачивающие линейное время Cn на анализ цепочки длины n.

Каждый метод синтаксического анализа предполагает свой способ построения по грамматике программы-анализатора, которая будет осуществлять разбор цепочек.

В основе анализатора может быть автомат с магазинной памятью. Мы рассмотрим другой способ – метод рекурсивного спуска (система рекурсивных процедур).

Анализатор некорректен, если:
- не распознает хотя бы одну цепочку, принадлежащему языку;
- распознает хотя бы одну цепочку, языку не принадлежащую;
- зацикливается на какой-либо цепочке.

Метод анализа применим к данной грамматике, если анализатор, построенный в соответствии с этим методом, корректен.

Метод рекурсивного спуска (РС-метод)

Пример: пусть дана грамматика G = ({a, b, c, d}, {S, A, B}, P, S), где

P:

$S \rightarrow CBd$
$A \rightarrow a \mid Ca$
$B \rightarrow bA$

и надо определить, принадлежит ли цепочка cabad языку L(G).

Построим левый вывод этой цепочки:

$S \text{ –> } ABd \text{ –> } cABd \text{ –> } caBd \text{ –> } cabAd \text{ –> } cabad$

Следовательно, цепочка принадлежит *L(G)*.

$S \text{ –> } ABd \text{ –> } cABd \text{ –> } caBd \text{ –> } cabAd \text{ –> } cabad$

Построение левого вывода эквивалентно построению дерева вывода методом «сверху вниз» (нисходящим методом):

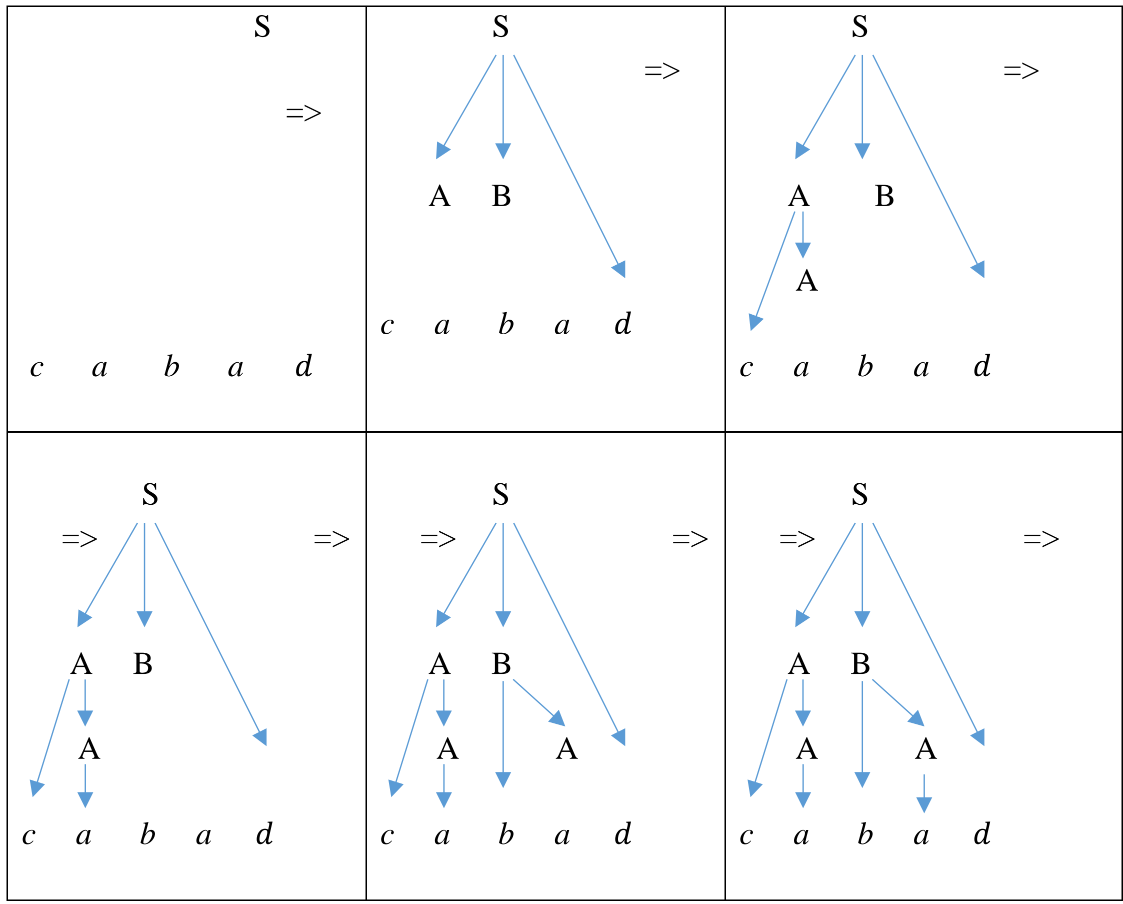


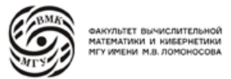
Таблица 14.1 Метод «сверху вниз»

Метод рекурсивного спуска (РС-метод)

Для **каждого нетерминала** грамматики создаётся своя процедура с **именем** этого нетерминала: её задача – начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала.

Если подцепочку удалось найти, то работа процедуры считается нормально завершённой и осуществляется возврат в точку вызова, иначе – разбор прекращается и сообщается об ошибке, цепочка не принадлежит языку.

Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: терминалы из правой части распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.



Программа анализатор для G_1

```
# include <iostream>
using namespace std;
int c;                          // текущий символ
void A ( );
void B ( );
void gc ( )
{
        cin >>  c;              // считать символ (лексему) из входного потока

}
```

G_1:
$S \to CBd$
$A \to a \mid Ca$
$B \to bA$

```
void s ( );
{
        cout  << "s - -> Abd, ";            //применяемое правило вывода
        A ( );
        B ( );
        if ( c == 'd' )
        else throw c;
}
```

G_1:
$S \to CBd$
$A \to a \mid Ca$
$B \to bA$

```
void A ( );
{

        if (c == 'a')
        }
                cout << "A -- > a, ";
                gc ( );
        {
        else if  ( c == 'c'; )
        }
                cout << "A -- > CA, ";
                gc ( );
                A ( );
        {
        else
                throw c;
        }
void B ( );
{
```

```
        if ( c == 'b' )
        {
                cout  << "B - -> ba, " ;
                gc ( );
                A ( );
        }
        else
                throw c;
        }
int main ()
{
        try
        {
                gc ( );
                S ( );
                if  ( c != '⊥' )            // проверяем, что достигнут конец цепочки
                        throw c;
                cout << "SUCCEESS !!! "<< endl ;
                return 0;
        }
        catch (int c)
        {
        cout  << "ERROR on lexeme" << c << endl ;
                return 1;
        }
}
```

G_1:
$S \to ABd$
$A \to a \mid cA$
$B \to bA$

G_1:
$S \to ABd$
$A \to a \mid cA$
$B \to bA$

Достаточное условие применимости метода рекурсивного спуска

Для применимости метода рекурсивного спуска достаточно, чтобы каждое правило в грамматике имело вид:

(а) либо $X \to a$;
где $A \in (N \cup T)^*$ и это единственное правило вывода для этого нетерминала;

(б) либо $X \to a_1 a_1 | a_2 a_2 | \dots | a_n a_n$,
где $a_1 \in T$ для всех $i = 1, 2, \dots, n$; $a_1 \neq a_j$ для $i \neq j$; $a_1 \in (N \cup T)^*$,
т.е. если для нетерминала X правил вывода несколько, то они должны начинаться с терминалов, причём все эти терминалы должны быть различными;

Это условие не является необходимым.

Найдём критерий применимости

РС-метод применим, если и только если левый вывод (или дерево нисходящим способом) можно построить, начиная с начального символа S, так, что на каждом шаге вывода решение о том, какое правило (альтернативу) применять для замены левого нетерминала, безошибочно принимается по первому символу из непрочитанной части входной цепочки (т.е. по «текущему» символу).

Рассмотрим примеры:

G_2:
S –> ABd
A –> a | cA
B –> bA

G_2 неоднозначна, РС-метод неприменим.
Нельзя дать однозначный прогноз, что делать на первом шаге при анализе цепочки, начинающейся с символа *a* (т.е. по текущему символу *a* невозможно сделать однозначный выбор: *S –> aA* или*S –> B).*

G_3 однозначна, но РС-метод неприменим
G_3:
S –> A | B
A –> aA | d
B –> aB | b

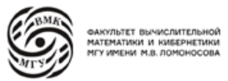
Определение: множество first (*a*) – это множество терминальных символов, которыми начинаются цепочки, выводимые из цепочки *a* в грамматике **G = (T, N, P, S),** т.е.
first (*a*) = {*a* ε T | *a* => *aa*’, где *a* є (N ᴗ T)*, *a*’ є (N ᴗ T)*}.

Например, first (*A*) = {*a*, d}, first (*B*) = { *a*, b}. Пересечение этих множеств не пусто:
first (*A*) ∩ first (*B*) = {*a* } ≠ ∅, и поэтому метод рекурсивного спуска к G_3 неприменим.

Итак, наличие в грамматике правил вида X → a | Σ, таких что first (*A*) ∩ first (*B*) = {*a* } ≠ ∅, делает метод рекурсивного спуска неприменимым.

Рассмотрим несколько *примеров:*

G_4:	first (a) = { a}, first (BDc) = { b, c};
S –> aA| BDc	first (BAa) = { a, b}, first (aB) = { a};
A –> BAa | ab| b	first (BAa) first (b) = { b};
B–> ε	first (ε) ≠ ∅;
D –> B | b	first (B) ≠ ∅, first (b) = {b}.



Метод рекурсивного спуска неприменим к грамматике G_4, так как first (B*Aa*) ∩ first (*aB*) = {*a*} ≠ ∅.

G_5:
S –> aA
A –> BC | B
C –> b | ε
B–> ε

Пересечение множеств *first* пусто, но РС-метод неприменим.
Действительно, BC => ε и B => ε. Цепочка *a* имеет два различных дерева *вывода*

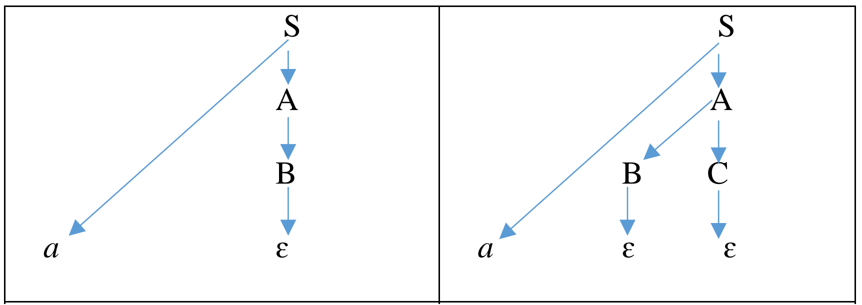


Рис.14.2 Два различных дерева вывода для цепочки a

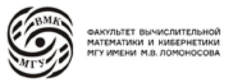
Таким образом, если **в грамматике для двух различных правил X → *a* /β выполняются соотношения** *a* => b *и* β => ε*,* то метод рекурсивного спуска **неприменим**.

Рассмотрим примеры с единственной альтернативой, из которой выводится ε.

G_6:
S –> Cad | d
A –> aA | ε
Метод применим: если текущий символ *a,* то выбираем альтернативу A –> aA, иначе A –> ε

G_7:
S –> Bd
B–> cAa | a
A –> aA | ε

Неприменим, т.к. Ф невозможно правильно выбрать альтернативу без «заглядывания» на символ вперёд.



Определение: множество follow (A) – это множество терминальных символов, которые могут появляться в сентенциальных формах грамматики непосредственно справа от A, т.е.
follow (A) = { a є T | aS => aAβ, β => ay, A є N, a, β, y є (T ᴗ N)*}

Тогда, если в грамматике есть пара правил **X → a /β, таких что** β => ε,
first (X) ∩ follow (X) ≠ ∅, то метод рекурсивного спуска неприменим к данной грамматике.

Утверждение. Пусть G – КС-грамматика. Метод рекурсивного спуска к G, если и только если для любой пары альтернатив вида X → a /β выполняются следующие условия:

(1) first (a) ∩ first (β) = ∅
(2) справедливо не более чем одно из двух соотношений:
 a => ε, β => ε;
(3) если β => ε, то first (X) ∩ follow (X) = ∅

Канонический вид для РС-метода

(1) либо X → a,
 где a є (T ᴗ N)* и это единственное правило вывода для этого терминала;

(2) либо X → a_1a_1| a_2a_2| …| a_n a_n,
 где a_1 *є T* для всех $i = 1, 2, …, n$; $a_1 \neq a_j$ для $i \neq j$; a_1 є (T ᴗ N) *, т.е. если для нетерминала X правил вывода несколько, то они должны начинаться с терминалов, причем, все эти терминалы должны быть *попарно* различными;

(3) либо X → a_1a_1| a_2a_2| …| ε,
 где a_1 *є T* для всех $i = 1, 2, …, n$; $a_1 \neq a_j$ для $i \neq j$;
 a_1 є (T ᴗ N) *, и first (X) ∩ follow (X) = ∅.

Этот вид удобен для построения рекурсивных процедур, но лн даёт только достаточное условие применимости.

Вопрос: если грамматика не удовлетворяет критерию применимости РС-метода, то существует ли эквивалентная КС-грамматика, для которой метод рекурсивного спуска применим?

К сожалению, нет алгоритма, отвечающего на этот вопрос для произвольной КС-грамматики, т.е. это **алгоритмически неразрешимая проблема.**

Модификация метода для грамматик с итерациями:

L → *a* {, *a*}
L → *a* | *a*, L

```
void L ()
{if ( c! = 'a') throw c;
gc ( ) ;
while ( c == ',')
{gc ( ) ;
if (c! = 'a') throw c;
else gc ( ) ;}
}
```

Важно, чтобы в любой сентенциальной форме после L не было запятой, иначе L прочитает «не свою» запятую.
(Вместо запятой в данном примере может быть любой другой символ.)

Пример, когда анализатор по выше приведённой схеме не даёт корректный ответ:

G:
S –> LB⊥
L –> *a* {, *a*}
C –> b | ε
B–> , b

Если для этой грамматики написать анализатор, действующий РС-методом, то цепочка *a, a, a, b* будет признана им ошибочной, хотя *a, a, a, b* є L(G).

В языках программирования после повторяющихся конструкций обычно идёт какой-нибудь новый символ, так что подобных проблем не возникает:

var a, b, c, d: integer;   или **int** a, b, c, d;

Если грамматику переписать без итерации { }:

S –> LB⊥
L –> *a* M
M –> , *a* M | ε
B–> , b

то нетрудно видеть, что a_1 є (T ∪ N) *, и first {, *a*} ∩ follow (M) =
и поэтому метод рекурсивного спуска неприменим.

Эквивалентные преобразования для КС-грамматик, которые могут помочь перестроить исходную грамматику так, чтобы РС-метод был применим.

1) Если в грамматике есть нетерминалы, правила вывода которых **леворекурсивны**, т.е. имеют вид:

   $A \to A a_1 \mid \dots \mid A a_n \mid \beta_1 \mid \dots \mid \beta_m$
   где $a_1 \in (V_t \cup V_n)$, $\beta_j \in (V_t \cup V_n)^*$, $i = 1, 2, \dots, n$; $j = 1, 2, m$;

   то левую рекурсию всегда можно заменить правой:
   $A \to \beta_1 A' \mid \dots \mid \beta_m A'$
   $A' \to a_1 A' \mid \dots \mid a_n A' \mid \varepsilon$

Будет получена грамматика, эквивалентная исходной.

2) Если в грамматике есть нетерминал, у которого несколько правил вывода начинаются **одинаковыми терминальными символами,** т.е. имеют вид

   $A \to a a_1 \mid a a_2 \mid \dots \mid a a_n \mid \beta_1 \mid \dots \mid \beta_m$
   где $a_1 \in V_t$;a_1, $\beta_j \in (V_t \cup V_n)^*$;

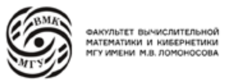
то можно преобразовать правила вывода данного нетерминала, объединив правила с общими началами в одно правило:
   $A \to aA' \mid \beta_1 \mid \dots \mid \beta_m$
   $A' \to a_1 \mid a_2 \mid \dots \mid a_n$
Будет получена грамматика, эквивалентная исходной.

3) Если в грамматике есть нетерминал, у которого **несколько** правил вывода, и среди них есть правила, **начинающиеся нетерминальными символами**, т.е. имеют вид

   $A \to B_1 a_1 \mid \dots \mid B_n a_n \mid a_1 \beta_1 \mid \dots \mid a_m \beta_m$
   $B_1 \to \gamma_{11} \mid \dots \mid \gamma_{1k}$
   ….
   $B_n \to \gamma_{n1} \mid \dots \mid \gamma_{np}$, где $B_i \in N$; $a_j \in T$; $a_i \beta_j$
   $\gamma_{ij} \in (T \cup N)^*$,

   то можно заменить вхождения нетерминалов B_i их правыми частями из правил вывода:

   $A \to \gamma_{11} a_1 \mid \dots \mid \gamma_{1k} a_1 \mid \dots \mid \gamma_{1n} a_n \mid \dots \mid \gamma_{np} a_n \mid a_1 \beta_1 \mid \dots \mid a_m \beta_m$
   Будет получена грамматика, эквивалентная исходной.



4) Если в грамматике есть правила
$A \to a_1 A \mid \ldots \mid A' a_n A \mid \beta_1 \mid \ldots \mid \beta_m \mid \varepsilon$
$B \to aA\beta$

и first $(X) \cap$ follow $(X) = \varnothing$ (из-за вхождения A в правило вывода для B), то можно преобразовать их в такие:
$B \to aA'$
$A' \to a_1 A' \mid \ldots \mid a_n A' \mid \beta_1 \beta \mid \ldots \mid \beta_m \beta \mid \beta$

Полученная грамматика будет эквивалентна исходной.

Задача разбора (синтаксический анализ) для неоднозначных грамматик

Две постановки задачи:

(1) *Даны КС-грамматика G и цепочка x. Требуется проверить: x є L(G)? Если да, то построить все деревья вывода для x (или все левые выводы для x, или все правые выводы для x)*

Для решения этой задачи можно обобщить метод рекурсивного спуска, чтобы он работал с возвратами, пробуя различные подходящие альтернативы.

(2) *Даны КС-грамматика G и цепочка x. Требуется проверить: x є L(G)? Если да, то построить одно дерево вывода для x (возможно, «наиболее подходящее» в некотором смысле).*

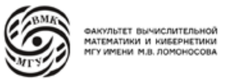
Рассмотрим *пример.* Грамматика неоднозначна. РС-метод неприменим.

G = ({if**, then, else,** a, b}, {S}, P, S, S'),

где P = {S –> **if** b **then** SS' | a;
S' –> **else** S | ε}

В этой грамматике для цепочки **if** b **then** if **else** b **then** a **else** a можно построить два различных дерева вывода:

Одно из них соответствует семантике Паскаля **else** относится к ближайшему **if.** Такое дерево можно получить, написав РС-процедуры, где S' по возможности отдаёт предпочтение непустой альтернативе.



(не соответствует семантике Паскаля)

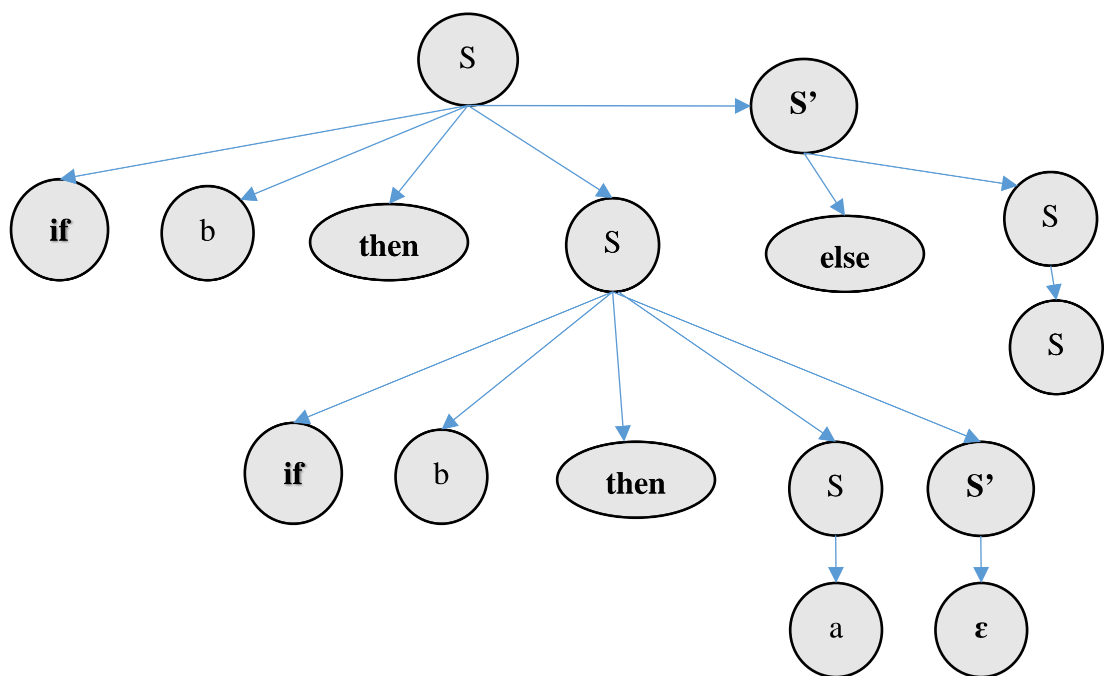


Рис.14.3 Пример: не соответствует семантике Паскаля

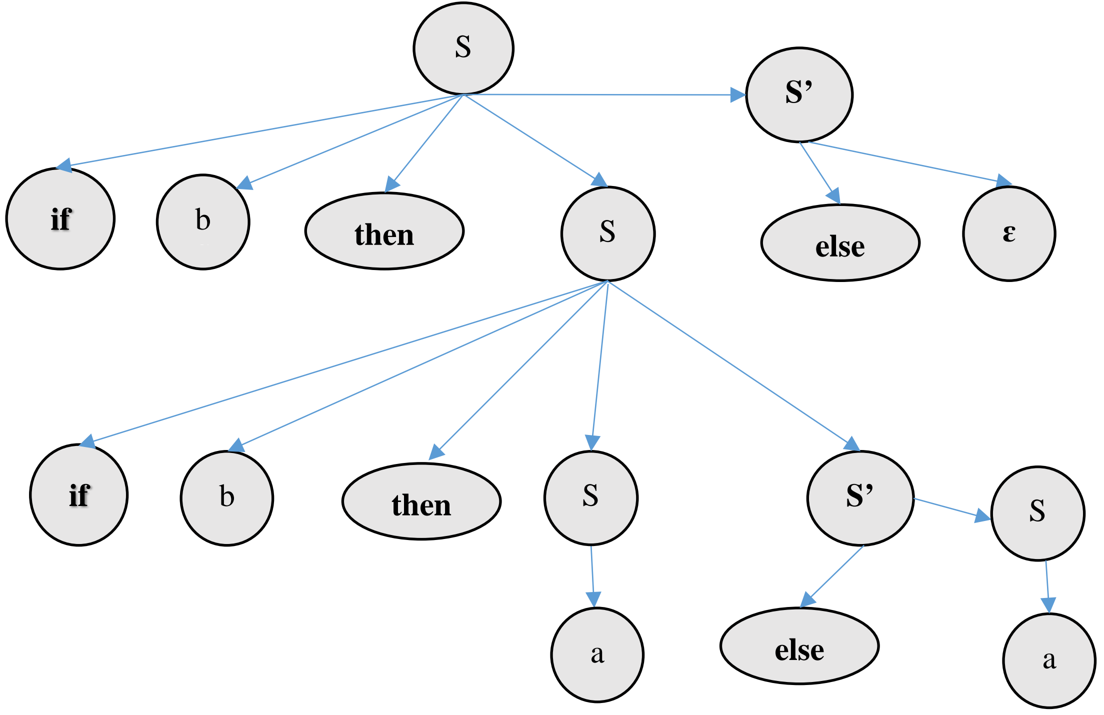
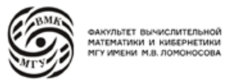


Рис.14.4 Пример: соответствует семантике Паскаля

Рассмотрим пример. Грамматика неоднозначна. РС-метод неприменим.

G = ({ **if, then, else,** a, b}, {S}, P, S, S’),



где P = {S –> **if** b **then** SS' | a ;
S' –> **else** S | ε }

В этой грамматике для цепочки **if** b **then** if **else** b **then** a **else** a можно построить два различных дерева вывода:

Одно из них соответствует семантике Паскаля: **else** относится к ближайшему **if.** Такое дерево можно получить, написав РС-процедуры, где S' по возможности отдаёт предпочтение непустой альтернативе.

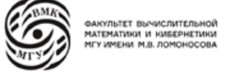
Синтаксический анализатор для М-языка

Будем считать, что синтаксический и лексический анализаторы взаимодействуют следующим образом: анализ исходной программы идёт под управлением синтаксического анализатора; если для продолжения анализа ему нужна очередная лексема, то он запрашивает ее у лексического анализатора; тот выдаётодну лексему и «замирает» до тех пор, пока синтаксический анализатор не запросит следующую лексему.

Соглашение:

- Лексический анализатор – это функция-член класса Scanner – get_lex ( ), которая качестве результата выдаёт лексемы типа (class) Lex;

- В переменной Lex curr_lex будем хранить текущую лексему, выданную лексическим анализатором, (а в переменной c_val – её значение, в c_type – её тип, это пригодится на этапе семантического анализа.)



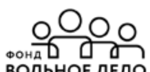
Лекция 15. Анализаторы языка

Грамматика модельного языка

P	→	**program** D1; B⊥
DI	→	**var** D {, D}
D	→	I {, I}**: [int** ⁞ **bool]**
B	→	**begin** S {; S} **end**
S	→	**I: = E** ⁞ **if** E **then** S **else** S ⁞ **while** E **do** S ⁞ B ⁞ **read** (I) ⁞ **write** €
E	→	EI [= ⁞ < ⁞ > ⁞ <=⁞ => ⁞!= ⁞] EI ⁞ EI
EI	→	T {[+ ⁞ - ⁞ or] T}
T	→	F {[* ⁞ / ⁞ and] F}
F	→	I ⁞ N ⁞ L⁞ not F ⁞ (E)
L	→	**true ⁞ false**
I	→	a ⁞ b ⁞ … ⁞ z ⁞ Ia ⁞ Ib ⁞ …⁞ Iz ⁞ I0 ⁞ I1 ⁞ … ⁞ I9
N	→	0 ⁞ 1 ⁞ … ⁞ 9 ⁞ N0 ⁞ N1 ⁞ …⁞ N9

→

```
class Parser {
        Lex curr_lex;
        type_of_lex c_type;
        int c_val;
        Scanner scan;
        // Stack < int, 100 > st_int;
        // Stack < type_of_lex, 100 > st_lex;
        void P ( ); void D1 (); void D (); void B (); void S ();
        void E (); void E1 (); void T (); void F ();

        void gl () {
                curr_lex = scan.get_lex ( );
                c_type = curr_lex.get_type ( );
        //      c_val = curr_lex.get_value ( );
        }
public:
```



```
        // Parser (const char *program) : scan (program) // , prog (1000)
        { }
        void analyze ( );
};
void Parser: : analyze ( ) {
        gl ( );
        P ( );
//      prog.print ( );
        Cout << endl << "OK" << endl;
}

 void Parser : : P ( ) {
if (c_type == LEX_PROGRAM) gl ( );
else throw curr_lex;
D1 ( );
if (c_type == LEX_SEMICOLON) gl ( );
else throw curr_lex;
B ( );
if (c_type != LEX_FIN) throw curr_lex;
}
void Parser : : D ( ) {
        if (c_type == LEX_VAR) {
                gl ( );
                D ( );
                while (c_type == LEX_COMMA) { gl ( ); D ( );}
else throw curr_lex;
}
…
```

Семантический анализ

Контекстно-свободные грамматики, с помощью которых описывают синтаксис языков программирования, не позволяют задавать контекстные условия, имеющиеся в любом языке.

Примеры наиболее часто встречающихся контекстных условий:

1) Каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;

2) При вызове число фактических параметров и их типы должны соответствовать числу и типам формальных параметров;

3) Обычно в языке накладываются ограничения на типы операндов любой операции, определённой в этом языке, на типы левой и правой частей в операторе присваивания, на тип параметра цикла, на тип условия в операторах цикла и условном операторе и т.п.

Семантический анализ для М-языка

Контекстные условия, выполнение которых надо контролировать в программах на М-языке:

1. Любое имя, используемое в программе, должно быть описано и только один раз.
2. В операторе присваивания типы переменной и выражения должны совпадать.
3. В условном операторе и в операторе цикла в качестве условия возможно только логическое выражения.
4. Операнды операции отношения должны быть целочисленными.
5. Тип выражения и совместимость типов операндов в выражении определяются по обычным правилам (как в Паскале).

Проверку контекстных условий совместим с синтаксическим анализом. Для этого в синтаксические правила вставим вызовы процедур, осуществляющих необходимый контроль, а затем перенесём их в процедуры рекурсивного спуска.

Обработка описаний

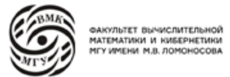
Лексический анализатор запомнил в таблице идентификаторов *TID* все идентификаторы-лексемы, которые были обнаружены в тексте исходной программы. Информация о типе переменных и о наличии их описания заносится в ту же таблицу.

i-ая строка таблицы TID соответствует идентификатору-лексеме вида (LEX_ID, i).

Раздел описаний имеет вид:

D –> I {J} : [int | bool],

т.е. имени типа (int или bool) предшествует список идентификаторов. Эти идентификаторы (вернее, номера соответствующих им строк таблицы *TID)* надо запоминать (мы будем их запоминать в стеке целых чисел ***Stack <int, 100> st_int***), а когда будет проанализировано имя типа, надо заполнить поля declare и type в этих строках.



Функция ***void Parser : : dec (type_of_lex type)*** считывает из стека номера строк таблицы TID, заносит в них информацию о типе соответствующих переменных, о наличии их описаний и контролирует повторное описание переменных.

```
void Parser : : dec (type_of_lex type {
        int i;
        while (!st_int.is_empty ( ) ) {
        i = st_int.pop ( );
        if (TID [i]. get_declare ( ) ) throw “twice”;
        else {
                TID [ I ]. get_declare ( ) ;
                TID [ I ]. put_type (type);
                }
        }
}
```

С учётом имеющихся функций правило вывода с действиями для обработки описаний будет таким:

```
D –> < st_int.reset ( ) > I < st_int.push (c_val)>
        {, I < st_int.push (c_val) >}:
        [int < dec (LEX_INT) > | bool < dec (LEX_BOOL) >]
```

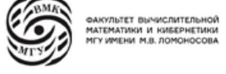
Контроль контекстных условий в выражении

Типы операндов и обозначение операций будем хранить в стеке
Stack < type_of_lex, 100> st_lex.

Если в выражении встречается лексема-целое число или логические константы **true** или **false,** то соответствующий тип сразу заносится в стек.

Если операнд – лексема-переменная, то необходимо проверить, описана ли она; если описана, то ее тип надо занести в стек. Эти действия можно выполнить с помощью функции check_id:

```
void Parser : : check_id ( ) {
        if (TID [c_val]. get_declare ( ) )
                st_lex.push (TID [ c_val]. get_type ( ) );
        else throw “not declared”;
}
```

Для контроля контекстных условий каждой тройки – «операнд – операция - операнд» (для проверки соответствия типов операндов данной двуместной операции) буем использовать функцию *check_op:*

```
void Parser: : check_op () {
        type_of_lex c t1, t2, jh, t = LEX_INT, r = LEX_BOOL;
        t2 = st_lex.pop ( ) ;
        op = st_lex.pop ( ) ;
        tl = st_lex.pop ( ) ;
        if ( op == LEX_PLUS | | op  ==LEX_MINUS | | op  ==LEX_TIMES | |
        op ==LEX_SLASH)
                r == LEX_INT;
         if ( op == LEX_OR | | op  ==LEX_AND)
                t = LEX_BOOL;
        if ( t1 = t2 && t1 == t ) st_lex.push ( r );
        else throw "wrong types are in operation";
        prog.put_lex (Lex (op));
```

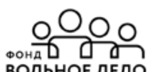
Для контроля за типом операнда одноместной операции *not* будем использовать функцию *check_not:*

```
void Parser : : check_not() {
        if ( st_lex.pop ( ) ! = LEX_BOOL)
        throw "wrong type is in not";
        else {
        st_lex.push (LEX_BOOL);
        prog.put_lex (Lex (LEX_NOT));
        }
}
```

В грамматике модельного языка задано старшинство операций: наивысший приоритет имеет операция отрицания, затем в порядке убывания приоритета – группа операций умножения (*, /, and), группа операций сложения (+, -, or), операции отношения.

E → EI \ EI [= | < | > | >= |! =] EI
E → T { [+ | - | or] T }
T → F { [* | / | and] F }
F → I | N | [true | false] | not F] | (E)

Именно это свойство грамматики позволит провести синтаксически-управляемый контроль контекстных условий.



Правила вывода выражений модельного языка с действиями для контроля контекстных условий:

E → EI | EI [= | < | > | >= |! =] st_lex.push (c_type) > EI <check_op ()>
E → T { [+ | - | or] T } st_lex.push (c_type) > T <check_op ()>
T → F { [* | / | and] F } st_lex.push (c_type) > F <check_op ()>
F → I <check id ()> | N <st_lex.push (LEX_INT) >| [true | false] <> st_lex.push (LEX_BOOL) > | not F <check_not ( ) >| (E)

Контроль контекстных условий в операторах

S –> I: = E | if E then S else S | while E do S | B | read (1) | write (E)

1. **Оператор присваивания** I: = E

Контекстное условие: в операторе присваивания типы переменной I и выражение E должны совпадать.

В результате контроля контекстных условий выражение E в стеке омтанется тип этого выражения (как тип результата последней операции); если при анализе идентификатора I проверить описан ли он, и занести егт тип в тот же стек (для этого можно использовать функцию check_id ()), то достаточно будет в нужный момент считать из стека два элемента и сравнить их:

```
void Parser : : eq_type () {
        if ( st_lex.pop ( ) ! = st_lex_pop ())
        throw "wrong types are in: = ";
}
```

Правило для операторов присваивания:
I <check_id ( ) >: = E < eq_type () >

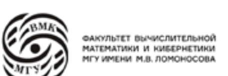
2. **Условный оператор и оператор цикла**

If E then S else S | while E do S

Контекстные условия: в условном операторе и в операторе цикла в качестве условия возможны только логические выражения.

В результате контроля контекстных выражений E в стеке останется тип этого выражения (как тип результата последней операции; следовательно, достаточно извлечь его из стека и проверить:

```
void Parser : : eq_bool () {
        if ( st_lex.pop ( ) ! = LEX_BOOL ())
```

```
        throw "expression is not boolean";
}
```

Правила для условного оператора и оператора цикла будут такими:
if E <eg_bool ( ) >then S else S while E <| eq_bool () > do S

3. Для проверки **оператора ввода** *read (I)* можно использовать следующую:

```
void Parser : : check_id_in_read () {
        if ( !  TID [ c_val].get_declare ( )) throw "not declared";
}
```

Правило оператора ввода будет таким:

Read (I <check_id_in_read ( )>.

В итоге получаем процедуры для синтаксического анализа методом рекурсивного спуска с синтаксически управляемым контролем контекстных условий, которые легко написать по правилам грамматики с действиями.

В качестве примера приведём функцию для нетерминала (раздел описаний):

```
void Parser : : D () {
        st_int.resert ();
        if (c_type != LEX_ID) throw curr_lex;
        else {
                st_push (c_val);
                gl ();
                while (c_type ==LEX_COMMA) {
                        gl ();
                        if (c_type != LEX_ID) throw curr_lex;
                        else {
                                st_int.push (c_val); gl ();
                        }
                }
                if (c_type != LEX_COLON) throw curr_lex;
                else { gl ();
                        if (c_type == LEX_INT) (dec LEX_INT); gl ();
                                else
                                     if (c_type == LEX_BOOL) (dec LEX_BOOL); gl ();
                                             else throw curr_lex;
                }}}
```

Внутреннее представление программы

Основные свойства языка внутреннего представления программ:

1. Внутреннее представление фиксирует синтаксическую структуру исходной программы;
2. Генерация внутреннего представления происходит в процессе синтаксического анализа;
3. Конструкции языка внутреннего представления должны относительно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

Некоторые способы внутреннего представления программ:

(1) Постфиксная запись
(2) Префиксная запись
(3) Многоадресный код с явно именуемыми результатами
(4) Многоадресный код с неявно именуемыми результатами
(5) Связные списочные структуры, представляющие синтаксическое дерево.

В основе каждого из этих способов лежит некоторый метод представления синтаксического дерева.

ПОЛИЗ – польская инверсная запись (постфиксная запись)

Пример: обычной (инфиксной)записи выражения

a* (b+c)-(d-e)/f

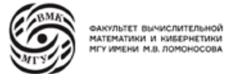
соответствует такая постфиксная запись:

abc+*de-f/-

- порядок операндов остался таким же как ив инфиксной записи,
- учтено старшинство операций
- нет скобок.

Записи:

a+в - инфиксная
ав+ - постфиксная
+ав - префиксная
+(ав) - функциональная



Рассмотрим следующие определения:
Простым будет называться выражение, состоящее из одной константы или имени переменной.

Приоритет и ассоциативность операций в инфиксных выражениях позволяет четко установить операндов:

a - простое выражение
a+в*c ~ a+(в*c) ;
a-b+c-d ~ ((a-b)+c) -d.

ПОЛИЗ выражений

(1) Если *E* является простым выражением, то ПОЛИЗ выражение *E* – это само выражение *E;*

(2) ПОЛИЗом выражения $E_1 \boldsymbol{\theta} E_2$,

где θ – знак бинарной операции, E_1 *и* E_2 операнды для $\boldsymbol{\theta}$,
является запись $E_1' \ E_2' \ \boldsymbol{\theta}$,
где E_1' *и* E_2' – ПОЛИЗ выражений E_1 *и* E_2 соответственно;

(3) ПОЛИЗом выражения $\boldsymbol{\theta} E$, где $\boldsymbol{\theta}$ – знак унарной операции, а *E – операнд* $\boldsymbol{\theta}$,

является запись $E' \boldsymbol{\theta}$,
где E′– ПОЛИЗ выражения E;

(4) ПОЛИЗом выражения *(E)* является ПОЛИЗ выражения *E.*

Алгоритм интерпретации с помощью стека

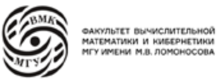
ПОЛИЗ просматривается поэлементно слева направо. В стеке хранятся значения промежуточных вычислений и результат.

1) Если очередной элемент – операнд, то его значение заносится в стек;

2) Если очередной элемент – операция, то на «вершине» стека сейчас находятся её операнды (это следует из определения ПОЛИЗа и предшествующих действий алгоритма); они извлекаются из стека, над ними выполняется операция, результат снова заносится в стек;

3) Когда выражение, записанное в ПОЛИЗе прочитано, в стеке останется один элемент – это значение всего выражения.

Замечание: для интерпретации, кроме ПОЛИЗа выражения, необходима дополнительная информация об операндах, хранящаяся в таблицах.



Лекция 16. ПОЛИЗ

Алгоритм Дейкстры перехода в ПОЛИЗ выражений

Будем считать, что ПОЛИЗ выражение будет формироваться в массиве, содержащем лексемы — элементы ПОЛИЗа и при переводе в ПОЛИЗ будет использоваться вспомогательный стек, также содержащий элементы ПОЛИЗа – Операции, имена функций и круглые скобки.

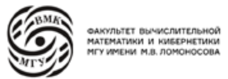
1. Выражение просматривается один раз слева направо.

2. Пока есть непрочитанные лексемы входного выражения, выполняем действия:

   а) Читаем очередную лексему,
   б) Если лексема является числом или переменной, добавляем её в ПОЛИЗ-массив,
   в) Если лексема является символом функции, помещаем ей в стек,
   г) Если лексема является разделителем аргументов функции (например, запятая):
   - до тех пор, пока верхним элементом стека не станет открывающаяся скобка, выталкиваем элементы из стека в ПОЛИЗ-массив. Если открывающаяся скобка не встретилась, это означает, что в выражении либо неверно поставлен разделитель, либо не согласованы скобки.
   д) Если лексема является операцией θ, тогда:

   - Пока приоритет θ меньше либо равен приоритету операции, находящейся на вершине стека (для лево-ассоциативных операций), или приоритет $\boldsymbol{\theta}$ сторого меньше приоритета операции, находящейся на вершине стека (для право-ассоциативных операций), выталкиваем верхние элементы стека в ПОЛИЗ-массив;
   - Помещаем операцию θ в стек.

   е) Если лексема является открывающей скобкой, помещаем её в стек;
   ж) Если лексема является закрывающей скобкой, выталкиваем элементы из стека в ПОЛИЗ-массив до тех пор, пока на вершине стека не окажется открывающая скобка. При этом открывающая скобка удаляется из стека, но в ПОЛИЗ-массив не добавляется.
   Если после этого шага на вершине стека оказывается символ функции, выталкиваем его в ПОЛИЗ-массив.
   Если в процессе выталкивания открывающей скобки не нашлось и стек пуст, это означает, что в выражении не согласованы скобки.



3. Когда просмотр входного выражения завершен, выталкиваем все оставшиеся в стеке символы в ПОЛИЗ-массив. (В стеке должны были оставаться только символы операций; если это не так, значит в выражении не согласованы скобки).

Представление операторов

Оператор присваивания

I := E

В ПОЛИЗе будет записан как

I E:=

где ":=" - это двухместная операция, а I и E – её операнды; I означает, что операндом операции ":=" является адрес переменной 1, а не её значение.

Расширение набора операций ПОЛИЗа

Операция перехода (обозначается «!») в терминах ПОЛИЗа означает, что процесс интерпретации надо продолжить с того элемента ПОЛИЗа, который указан как операнд этой операции.

Чтобы можно было ссылаться на элементы ПОЛИЗа, будем считать, что все они перенумерованы, начиная с 1 (например, занесены в последовательные элементы одномерного массива).

Пусть ПОЛИЗ оператора, помеченного меткой , начинается с номера p, тогда оператор перехода

goto L в ПОЛИЗе записывается так:

p !

где ! – операция выбора элемента ПОЛИЗа, номер которого равен p.

Операция условный переход «по лжи» с семантикой if (!B) goto L

Это двухместная операция с операндами B и I. Обозначим её !F, тогда в ПОЛИЗе переход «по лжи» записывается так:

B' p !F

где p – номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L, B' – ПОЛИЗ логического выражения B.

Семантика условного оператора:

if E then S_1 **else** S_2

с использованием введённой операции может быть описана так:

if (!E) goto L_3; S_1; goto L_3; L_2; S_2; L_3;…

Тогда ПОЛИЗ условного оператора будет таким (порядок операндов – прежний):

E' p_2 !F S_1* p_3 1 S_2' …,

где p_i – номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 2, 3, E'— ПОЛИЗ логического выражения E.

Семантика оператора цикла **while E do S** может быть описана так:

L_0: if (!E) goto L_1; S; goto L_0; L_1; S_2;…

Тогда ПОЛИЗ оператора цикла **while** будет таким (порядок операндов – прежний!):

E’ p_1 !F S’ p_0 ! …,

где p_i – номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 0, 1, E’— ПОЛИЗ логического выражения E.

Операторы ввода и вывода М-языка являются одноместными операциями.
Оператор ввода **read** (I) в ПОЛИЗе будет записан как **I read.**
Оператор вывода **write** (E) в ПОЛИЗе будет записан как **E’ write,**
где E’ – ПОЛИЗ выражения E.

Синтаксически управляемый перевод

В основе синтаксически управляемого перевода лежит уже известная нам грамматика с действиями.

G_{expr} – грамматика, описывающая простейшее арифметическое выражение:

E –> T {+T}
T –> F {*F}
F–> a | b | (E)

G_{expr_polish} – грамматика с действиями по переводу выражения в ПОЛИЗ:

E –> T {+T cout <<’ + ‘;}
T –> F {*F cout <<’ * ‘;}
F–> a {cout <<’ a ‘;} | b< cout <<’ b ‘; > | (E)

В процессе анализа методом рекурсивного спуска входной цепочки *a+b*a* по грамматике G_{expr_polish} в выходной поток будет выведена цепочка *a b a* +*.

Определение: Пусть T_1 и T_2 – алфавиты. *Формальный перевод* τ – это подмножество всевозможных пар цепочек в алфавитах T_1 и T_2: $\tau \subseteq (T_1^* \times T_2^*)$.

Назовём *входным* языком перевода τ язык

$$L_{вх}(\tau) = \{\alpha \mid \exists\, \beta : (\alpha;\ \beta) \in \tau\}.$$

Назовём *целевым* (или *выходным)* языком перевода τ язык

$$L_{ц}(\tau) = \{\beta \mid \exists\, \alpha : (\alpha;\ \beta) \in \tau\}.$$

Перевод τ неоднозначен, если для некоторых $\alpha \in T_1^*$, $\beta, \gamma \in T_2^*$, $\beta \neq \gamma$ справедливы соотношения: $(\alpha, \beta) \in \tau$ и $(\alpha, \gamma) \in \tau$.

Рассмотренная выше грамматика G_{expr_polish} задаёт однозначный перевод: каждому выражению ставится в соответствие единственная польская запись. Неоднозначные переводы могут быть интересны при изучении моделей естественных языков: для трансляции языков программирования используются однозначные переводы.

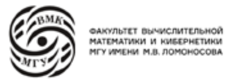
Генератор внутреннего представления программы на М-языке

Каждый элемент в ПОЛИЗе – это лексема, т.е. пара вида (тип_лексемы, значение_лексемы). При генерации ПОЛИЗа будем использовать дополнительные типы лексем:

POLIZ_GO –“!”
POLIZ_FGO –“!F”;
POLIZ_LABEL – для ссылок на номера элементов ПОЛИЗа;
POLIZ_ADDRESS – для обозначения операндов-адресов (например, в ПОЛИЗе оператора присваивания).

Будем считать, что генерируемая программа размещается в объекте
Poliz prog (1000);   класса Poliz:

```
class Poliz (
        lex *p;
        int size;
        int free;
public:
        Poliz (int max_size) (p=new Lex [size = max_size]; free = 0;);
        ~ Poliz ( ) {delete [] p;};
        void put_lex (Lex l) {p [free] = l; free ++;};
        void put_lex (Lex l, int place) {p [place] = l;};
        void blank ( ){free ++;};
        int get_free ( ) {return free;};
        lex & operator [ ] (int index) {
                if (index > size) throw “POLIZ: out of array”; else
        if (index > free) throw “POLIZ: indefinite element of array”;
                else return p [index];
        };
        void print ( ) {
        for (int i = 0; I < free ; i++) cout << p [i]; };
        };
```

Добавим действия по генерации в некоторые функции семантического анализа: *check_not ( ) и check_op ( )*

```
void Parser : :  check_not () {
        if (st_lex.pop () != LEX_BOOL);
        throw "wrong type is in not";
else {
        st_lex.push (LEX_BOOL);
        prog.put_lex (Lex (LEX_NOT)
        }
}

void Parser : :  check_op () {
        type_of_lex t1, t2, op, t = LEX_INT, r = LEX_BOOL;
        t2 = st_lex.pop ( );
        op = st_lex.pop ( );
        t1 = st_lex.pop ( );
        if (op == LEX_PLUS | | op == LEX_MINUS | | op == LEX_TIMES
        | | op == LEX_SLASH)
        if (op == LEX_OR | | op == LEX_END)
                t= LEX_BOOL;
        if (t1 == t2  &&  t1 == t) st_lex.push ( r );
        else throw "wrong types are in operation";
        prog. put_lex (Lex (op));
        }
```

Грамматика, содержащая действия по контролю контекстных условий и переводу выражений модельного языка в ПОЛИЗ

```
E –> E1 | E1 [ = |< | > | ] < st_lex.push (c_type)> E1 <check_op ( ) >
E1 –> T { [ + | - | or | ] < st_lex.push (c_type)> T <check_op ( ) >
T –> F { [ * | / | and | ] < st_lex.push (c_type)> F <check_op ( ) >
F –> I < check_id (LEX_INT);  prog.put_lex (curr_lex);> |
        N < st_lex.push (LEX_INT); prog.put_lex (curr_lex);> |
[true | false] < st_lex.push (LEX_BOOL); prog.put_lex (curr_lex);> |
not F <check_out ( ); | ( E )
```

Пример реализации анализа и перевода для нетерминала F:

```
void Parser : : F ( )
{
```

```
        if ( c_type == LEX_ID)
    {
            check_id ( );
            prog.put_lex (Lex (LEX_ID, c-val));
            gl ();
        }
        else if (c_type == LEX_NUM)
        {
            st_lex.push (LEX_INT);
            prog. put_lex (curr_lex);
            gl ();
        }
        else if (c_type == LEX_TRUE)
        {
            st_lex.push (LEX_BOOL);
            prog. put_lex (Lex ( LEX_TRUE, 1) );
            gl ();
        }
```

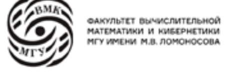
Действия для оператора присваивания

S –> I <check_id (); prog.put_lex (Lex (POLIZ_ADDRESS, c_val)); > ^ =>
E <eqtype ( ); prog. put_lex (Lex (ASSING));>

Для условного

if (!E) goto I2; S1; goto I3, I2; S2; I3;…..

S –> ***if*** E < eqbool ( ); pl2 = prog.get_free ( ); prog.blank ( );
prog. put_lex (Lex (POLIZ_FGO));>
then S1 < pl3 = prog.get_free ( ); prog.blank ( );
prog. put_lex (Lex (POLIZ_GO));>
prog. put_lex (Lex (POLIZ_LABEL, prog.get_free ( )), pl2);>
else S2 prog. put_lex (Lex (POLIZ_LABEL, prog.get_free ( )), pl3);>



Лекция 17. ПОЛИЗ (2)

Оператор цикла **while E do** S описывается так:

L_0: if (!E) goto L_1; S; goto L_0; L_1; S_2;…

а грамматика с действиями по контролю контекстных условий и переводу оператора цикла в ПОЛИЗ будет такой:

```
S –> while < pl0 = prog.get_free () ; > E < eqbool ( );
                p1 = prog.get_free () ; prog/blank ( );
                prog.put_lex (Lex (POLIZ_FGO));>
  do    S       < prog.put_lex (Lex (POLIZ_LABEL, pl0);
                prog.put_lex (Lex (POLIZ_GO));
                prog.put_lex (Lex (POLIZ_LABEL, prog.get_free ()) ; pl1);>
```

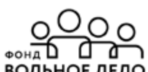
Замечание: переменные pli (i= 0, 1, 2, 3) должны быть локализованы в процедуре S, иначе возникает ошибка при обработке вложенных условных операторов.

Грамматика с действиями по контролю контекстных условий и переводу в ПОЛИЗ операторов ввода и вывода:

```
S –> read (I < check_id_in_read ();
                        prog.put_lex (Lex ( POLIZ_ADDRESS, c_val)); >)
                        < prog.put_lex (LEX_READ));>

S –> write ( E) <prog.put_lex (Lex (LEX_WRITE));>
```

Интерпретатор ПОЛИЗ для модельного языка

```
class Executer {
        Lex pc_el;
        public:
                void execute (Poliz & prog);
};

void Execute : : execute (Poliz & prog) {
        Stack < int, 100 > args;
        int  i,  j, index = 0, size = prog.get_free ( );
        while ( index < size ) {
                pc_el = prog [ index ];
                switch (pc_el.get_type ( ) ) {
                        case LEX_TRUE: case LEX_FALSE: case LEX_NUM:
```



```
case POLIZ_ADDRESS: case POLIZ_LABEL:
        args. push ( pc_el.get_value ( ) ); break;
case LEX_ID:
        i = (pc_el.get_value  ( );
        if = ( TID [i].get_assign () ) {
                args. push ( TID [i]. get_value () ); break;}
        else throw "POLIZ": indefinite identifier";
case LEX_NOT:
        args. push ( !args. pop  () ); break;
case LEX_OR:
        i = args. pop  ();
         args. push ( args. pop  ()  | | i); break;
case LEX_AND:
        i = args. pop  ();
         args. push ( args. pop  ()  |& i); break;
case POLIZ_GO:
        index = args. pop  () - 1 ; break;
case POLIZ_FGO:
        i = args. pop  () ;
        if (! args. pop  () ) index = I – 1; break;
case LEX_WRITE:
        cout  << args. pop  () << endl; break;
case LEX_READ:
        {int k;
        i = args.pop ( );
        if = ( TID [i].get_type () == LEX_INT) {
                cout << "Input int value for";
                cout <<( TID [i].get_name () << endl;
                cin >> k;
}
else {
        char j [20];
        rep:
                cout << "Input boolean value";
                cout << (true or false) for";
                cout << (TID [i].get_name () << endl;
                cin >> j;
}
                if (! strcmp (j, "true")) k = 1;
                else
                if (! strcmp (j, "false")) k = 0;
                else {
                cout <<  "Error in input: true/ false";
```

```
                cout << endl;
                goto rep;}
        TID [i].put_value (k);
        TID [i].put_assing ();
        break;
case LEX_PLUS:
        args. push ( args. pop  () + args. pop  ()); break;
case LEX_TIMES:
        args. push ( args. pop  () * args. pop  ()); break;
case LEX_MINUS:
        i = args. pop  ();
        args. push ( args. pop  () * args. pop  ()); break;
case LEX_SLASH:
        i = args. pop  ();
        if (! i  { argc. push (args. pop  () / i); ( k = 1;
        args. push ( args. pop  () * args. pop  ()); break;
        else throw "POLIZ: divide by zero";
case LEX_EQ:
        args. push ( args. pop  () == args. pop  ());
break;
case LEX_LSS:
        i = args. pop  ();
        args. push ( args. pop  () < i); break;
case LEX_GTR:
        i = args. pop  ();
        args. push ( args. pop  () >  i); break;
case LEX_LEQ:
        i = args. pop  ();
        args. push ( args. pop  () <=  i); break;
case LEX_GEQ:
        i = args. pop  ();
        args. push ( args. pop  () >=  i); break;
case LEX_NEQ:
        i = args. pop  ();
        args. push ( args. pop  () !=  i); break;

case LEX_ASSING:
        i = args. pop  ();
        j = args. pop  ();
        TID [j].put_value (i);
        TID [j].put_assing (); break;
default: throw "POLIZ: unexpected elem";
```

```
                } // end of switch
                index++;
                } // end of while
        cout << "Finish of executing!!!" << endl;
}

class Interpretator {
                Parser pars;
                Executer E;
        public:
                Interpretator (char*program): pars (program) {};
                void interpretator  ( );
};

void Interpretator : : interpretator  ( ) {
                pars.analyse () ;
                E.execute (pars.prog);
}

int  main ( ) {
        try {
                Interpretator I ("program.txt");
                I Interpretator  ( );
                return ( );
        }
catch (char c) {
        cout << "unexpected symbol" <<  c << endl; return 1;
        }
catch (Lex 1) {
        cout << "unexpected lexeme"  cout  << 1; return 1;
        }
catch (const char* source) {
        cout << source <<  endl; return 1;
        }
}
```

Лекция 18. Распределение памяти

При трансляции языков программирования приходиться решать такую задачу как распределение памяти.

Распределение памяти – это процесс, в результате которого отдельным элементам исходной программы ставятся в соответствие адрес, размер и атрибуты памяти, необходимой для размещения лексических единиц.

Область памяти – это блок ячеек памяти, выделяемых для данных и каким-то образом объединённых логически
.

Распределение памяти выполняется после фазы анализа текста исходной программы *на этапе подготовки к генерации объектного модуля* (перед генерацией кода объектного модуля).

Исходными данными для процесса распределения памяти служат сведения о семантике конструкций языков программирования, таблица идентификаторов, построенная лексическим анализатором и информация, полученная синтаксическим анализатором при анализе декларативной части программы.

Современные компиляторы, в основном работают с относительными, а не с абсолютными адресами ячеек памяти.

Семантика программ подразумевает, что при их выполнении области памяти будут необходимы для хранения:

- Кодов пользовательских программ;
- Данных, необходимых для работы этих программ;
- Кодов системных программ, обеспечивающих поддержку пользовательских программ в период их выполнения;
- Записей о текущем состоянии процесса выполнения программ (например, записей об активации процедур).

По способу использования области памяти делятся на глобальные и локальные, а по способу распределения – на статические и динамические.

Классы памяти

Выделяемую память можно разделить на:
- Локальную / глобальную
- Статическую / динамическую.

Схема классов памяти

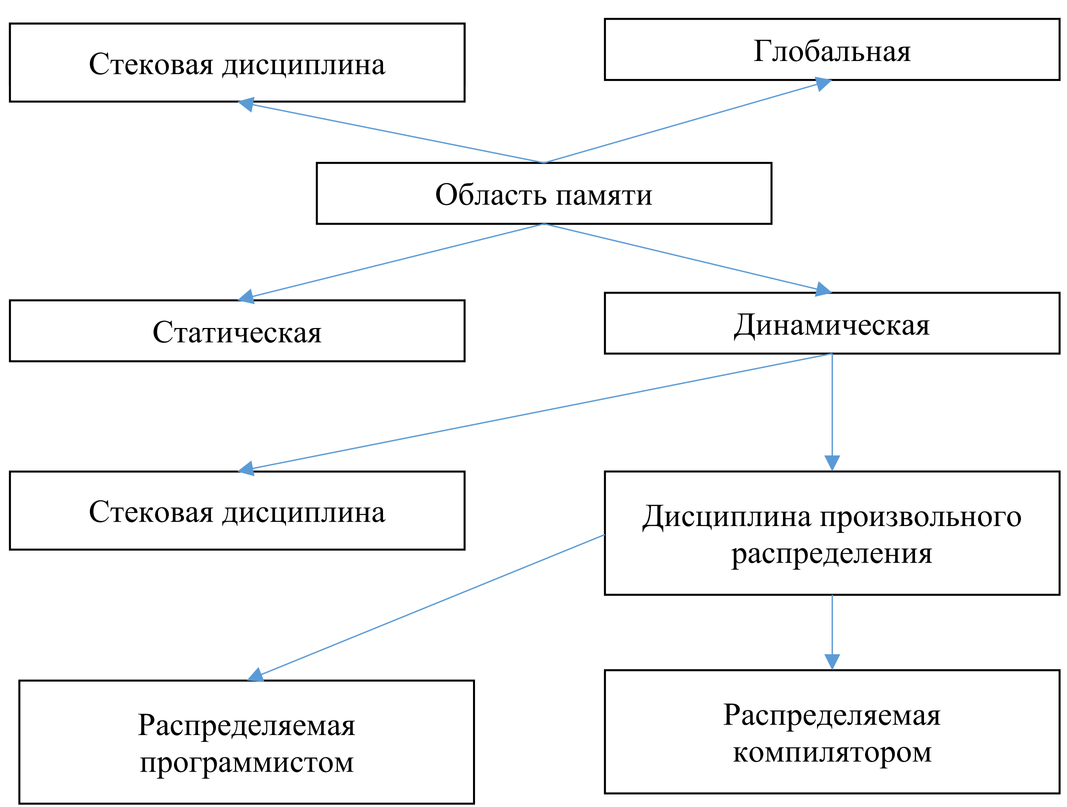


Рис.18.1 Схема классов памяти

Локальная память – это область памяти, которая выделяется в начале выполнения некоторого фрагмента результирующей программы (блока, функции, оператора…) и может быть освобождена по завершении выполнения данного фрагмента.

Глобальная память – это область памяти, которая выделяется один раз при инициализации результирующей программы и действует все время выполнения программы. Как правило, глобальная область памяти доступна из любой части исходной программы.

Статическая память – это область памяти, размер которой известен на этапе компиляции. Для статической памяти компилятор порождает некоторый адрес (как правило, относительный), и дальнейшая работа с ней происходит относительно этого адреса.

Динамическая память – это область памяти, размер которой становится известным только на этапе выполнения результирующей программы. Для динамической памяти компилятор порождает фрагмент кода, который отвечает за распределение памяти (её выделение и освобождение). Как правило, с динамическими областями памяти связаны

многие операции с указателями и с экземплярами объектов (классов) в объектно-ориентированных языках программирования.

Динамические области памяти можно разделить на динамические области памяти, **выделяемые пользователем и непосредственно компилятором.**

Общие принципы генерации объектного кода

При генерации объектного кода компилятор переводит текст программы во внутреннем представлении в текст программы на выходном языке (как правило, машинном).

Генерация объектного кода происходит на основе:

- Определённой на фазе анализа компиляции синтаксической структуры программы,
- Информации, хранящейся в таблице идентификаторов,
- Результата распределения памяти.

Характер и сложность отображения промежуточного представления программы в последовательность команд на машинном языке зависит от языка внутреннего представления и архитектуры вычислительной схемы, на которую ориентирована результирующая программа.

Часто для построения кода результирующей программы компиляторы используют синтаксически управляемый перевод.

Оптимизация программ

Оптимизация программы – это изменение компилируемой программы (в основном переупорядочивание и замена операций) с целью получения более эффективной объектной программы.

Используются два критерия эффективности результирующей программы:
- *скорость* выполнения программы и
- *объём памяти*, необходимый для выполнения программы.

В общем случае задача построения оптимального кода программы алгоритмически неразрешима. К тому же, компилятор обладает весьма ограниченными средствами анализа семантики входной программы в целом.

Основная оптимизация программы должна производится программистом.

Принципиально различаются два основных вида оптимизирующих преобразований:

1. **Машинно-независимые преобразования** исходной программы
2. **Машинно-зависимые преобразования** результирующей объектной программы.

Оптимизация может привести к изменению смысла программы. Например, в случае исключения из программы вызова функции с «побочным эффектом».

У современных компиляторов существует возможность выбора критерия оптимизации и отдельных методов оптимизации.

Машинно-независимые оптимизирующие преобразования

Машинно-независимые преобразования исходной программы производятся в основном над её внутренним представлением и основаны на известных математических и логических преобразованиях.

1) **Удаление недостижимого кода.**
(задача компилятора найти и убрать его)

Пример:

```
if (1)
        S1;     =>;
else
        S2;
```

2) **Оптимизация линейных участков программы.**

В современных системах программирования профилировщик на основе результатов запуска программы выдаёт информацию о том, на какие её линейные участки приходится основное время выполнения.

а) Удаление бесполезных присваиваний
*a = b * c; d = b+c; a = d * c; => d = b+c; a = d*c;*

Однако, в следующем примере эта операция уже небесполезна:
*p = & a; a = b * c; d = *p+c; a = d * c;*

б) Исключение избыточных вычислений
*d = d + b * c; a = d + b * c; c = d + b * c; =>*
*t = b *c; d = d + t; a = d + t; c = a4*

в) Свёртка объектного кода (выполнение во время компиляции тех операций исходной программы, для которых значение операндов уже известны).

*i = 2 + 1; j = 6 * i + i;   = > i = 3; j = 21;*

г) Перестановка операций (для дальнейшей свёртки или оптимизации вычислений).

*a = 2 * b * 3 * c; => a = (2 * 3) * (b * c);*
*a = (b + c) * (d + e); => a = (b +(c + (d + e)));*

д) Арифметические преобразования (на основе алгебраических и логических тождеств).

*a = b * c + b * d; => a = b* (c + d);*
*a * 1 +> a, a * 0 => 0, a + 0 => a.*

е) Оптимизация вычисления логических выражений.

a || b || c || d; => a, если **a** есть **true**,
Но! *a || f (b) || g(c)*   не всегда **a** (при **a = true),**
может быть побочный эффект.

3) **Подстановка кода функции вместо её вызова в объектный код.**

Этот метод, как правило, применим к простым функциям и процедурам, вызываемым непосредственно по адресу, без применения косвенной адресации через таблицы RTTI (Run Time Type Information).

Некоторые компиляторы допускают применять метод только с функциям, содержащим последовательные вычисления без циклов.
Язык С++ позволяет явно (inline), для каких функций желательно использовать inline-подстановку.

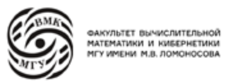
4) **Оптимизация циклов.**

а) Вынесение инвариантных вычислений из циклов.

```
for (i = 1; i <=10; i++)
        a [i]=b * c * a [i];          =>
d = b * c; for (i= 1; I <=10; i++)
        a [i]=d * a [i];
```

б) Замена операций с индуктивными (образующими арифметическую прогрессию) переменными (как правило, умножения на сложение)

```
for (i = 1; i <=N; i++)
        a [i]= i *10;                 =>
t = 10; i = N1; while (i <= N) {
        a [i]= t; t = t + 10; i++;    // упростили тело цикла
        a [i]= t
```



```
}

s = 10; for (i =1; i <= N; i++) {
        r = r + f (s); s = s +10;}
s = 10; m = N *10; while (s <= m {
        r = r + f (s); s = s +10;}
```

(избавились от одной индуктивной переменной (*i*))

в) Слияние циклов

```
for (i = 1; i <=N; i++)
        for (j = 1; i <=k; i++)
        a [i] [j]= 0;                =>

k = N * M;
for (i = 1; i <=N; i++)
        a [i]= 0; (только в объектном коде!)
```

г) Развёртывание циклов (можно выполнить для циклов, кратность выполнения которых известна на этапе компиляции).

```
for (i = 1; i <=N; i++)
        a [i]= i;                =>

        a [1]= 1;
        a [2]=2;
        a [3]=3;
```

Машинно-зависимые оптимизирующие преобразования

Машинно-зависимые преобразования результирующей объектной программы зависят от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. При этом может учитываться объем кэш-памяти, методы организации работы процессора.

Эти преобразования, как правило, являются «ноу-хау», и именно они позволяют существенно повысить эффективность результирующего кода.

1) Распределение регистров процессора.

Использование регистров общего назначения и специальных регистров (аккумулятор, счётчик цикла, базовый указатель) для хранения значения операндов и результатов вычислений позволяет увеличить быстродействие программы.

Доступных регистров всегда ограниченное количество, поэтому перед компилятором встаёт вопрос их оптимального распределения и использования при выполнении вычислений.

2) **Оптимизация передачи параметров в процедуры и функции**

Обычно параметры процедур и функций передаются через стек. При этом всякий раз при вызове процедуры или функции компилятор создаёт объектный код для размещения её фактических параметров в стеке, а при выходе из неё – код для освобождения соответствующей памяти.

Можно уменьшить код и время выполнения результирующей программы за счёт оптимизации передачи параметров в процедуру или функцию, передавая их **через регистры** процессора.

Реализация данного оптимизирующего преобразования зависит от количества доступных регистров процессора в целевой вычислительной системе и от используемого компилятором алгоритма распределения регистров.

Недостатки метода:

- Оптимизированные таким образом процедуры и функции не могут быть использованы в качестве библиотечных, т.к. методы передачи параметров через регистры не стандартизированы и зависят от реализации компилятора.

- Этот метод не может быть использован, если где-либо в функции требуется выполнить операции с адресами параметров.

Языки Си и C++ позволяют явно указать (**register**), какие параметры и локальные переменные желательно разместить в регистрах.

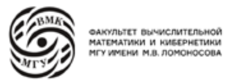
3) **Оптимизация кода для процессоров, допускающих распараллеливание вычислений.**

При возможности параллельного выполнения нескольких операций компилятор должен порождать объектный код таким образом, чтобы в нём было максимально возможное количество соседних операций, все операнды которых не зависят друг от друга.

Для этого надо найти оптимальный порядок выполнения операций для каждого оператора (переставить их).

$a + b + c + d + e + f;$ =>

для одного потока обработки данных: $(((a+b) + c) + d) + e) + f;$
для двух потоков обработки данных: $((a+b) + c) + ((d + e) + f);$



для трёх потоков обработки данных: $(a+b) + (c + d)+(e + f);$

Другие преобразования машинно-зависимые могут очень сильно завесить от архитектуры. Для одной архитектуры такие преобразования могут быть осмыслены, а для другой нет, потому что в ней нет каких-то элементов, позволяющих применить трюк при оптимизации.

Заключение

В целом мы рассмотрели все виды трансляции и некоторые приёмы генерации кода.

В каждом языке программирования можно выделить:
- алфавит, из каких символов строится программа;
- синтаксис, что является особой формой записи контекстно-свободной грамматики;
- семантика, где каждая синтаксическая конструкция имеет определённый смысл.

Алфавит, синтаксис и семантика как правило формально всегда описываются в языках программирования.

Кроме устройства языков программирования при реализации модельного М-языка мы научились ещё одному важному подходу в программировании, который можно назвать синтаксически-управляемая обработка данных.

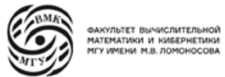
Данные имеют определённую структуру и её можно описать формальной грамматикой. Если это выполнено, то можно написать программу обработки данных, основываясь на этой грамматике.

Каждый язык программирования служит определённой цели, не существует универсального языка. Попытки создать такой язык не увенчались успехом, т.к. есть разные запросы/ прикладные задачи требуют разных средств.

Мы научились следующим языкам:

1 курс – алгоритмы и алгоритмические языки (Паскаль, алгоритм Маркова и машин Тьюринга)

2 курс – C, C++



Язык C++ обладает средствами, которые достались ему из C, позволяющими преодолеть любую защиту, предоставленную объектно-ориентированному программированию.

Вышеуказанные языки, которые мы рассмотрели являются универсальными. То, что мы могли делать, например, на машине Тьюринга с ограниченной лентой, это можно и промоделировать на всех остальных языках: низкоуровневых и высокоуровневых.

На этом мы наш курс заканчиваем.

Желаю вам с новыми знаниями овладевать новыми языками программирования и не позволять технологиям овладевать вами! Всегда сохранять критический взгляд на всё.

ВМК
МГУ
ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА
ФОНД
ВОЛЬНОЕ ДЕЛО
teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ