

Технологии построения программ

- Процессно-ориентированный подход: программа – это запись ряда последовательно выполняемых операций
 - Алгоритм (процесс выполнения) воздействует на данные
- Объектно-ориентированный подход: программа состоит из объектов – элементов данных и фрагментов алгоритмов, обрабатывающих данные и взаимодействующих друг с другом через интерфейсы
 - Данные управляют доступом к связанным с ними алгоритмам
 - Статическая структура системы описывается в терминах объектов и связей между ними, а динамическое поведение системы описывается в терминах обмена сообщениями между объектами

Основные принципы ООП

- Объект – программная сущность, обладающая состоянием (связанные с объектом информационные свойства) и поведением (набор операций, присущих объекту)
- Объектно-ориентированное программирование – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования

- Центральное звено ООП – **абстракция**

Благодаря абстракции сущности произвольной сложности можно рассматривать как единое целое, не вдаваясь в детали их внутреннего построения и функционирования

- Основные механизмы (постулаты) ООП:

- **Инкапсуляция**

- **Наследование**

- **Полиморфизм**

Каждый из этих механизмов в отдельности и все они вместе взятые – это **средства борьбы со сложностью программ**

Основные механизмы ООП

Инкапсуляция

- Обеспечивает сокрытие состояния объекта от остальных программных единиц
- Защищает данные от несанкционированного доступа со стороны алгоритмов, внешних по отношению к рассматриваемым
- С помощью интерфейса (формального описания способов доступа к алгоритмам и данным) жёстко контролирует доступ к алгоритмам и данным, связанным между собой

Основные механизмы ООП

Наследование

- Наследование – процесс, с помощью которого объект программы приобретает свойства другого объекта
- Наследование поддерживает концепцию иерархической классификации, то есть иерархического отношения типов объектов
- Наследование позволяет производному объекту наследовать от базового объекта общие атрибуты (состояние и поведение), а для себя определять только характеристики, делающие его уникальным в классе

Основные механизмы ООП

Полиморфизм

- Полиморфизм – свойство, позволяющее использовать одинаковый интерфейс для целой совокупности действий (“один интерфейс – много методов”)
- Различают
 - статический полиморфизм
 - динамический полиморфизм
 - параметрический (типовый) полиморфизм

Язык программирования Си++.

Приёмы декомпозиции

- Декомпозиция проектируемой системы есть последовательное выполнение следующих шагов:
 - Исследование статической структуры системы
 - Выделение используемых объектов
 - Фиксация связей между объектами
 - Исследование динамической структуры системы
 - Фиксация методов обмена сообщениями между объектами
- *Состояние объекта характеризуется перечнем (обычно статическим, неизменным) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств*

Язык программирования Си++.

Приёмы декомпозиции

- Операцией называется воздействие одного объекта на другой с целью вызвать соответствующую реакцию
- Поведение объекта – это его наблюдаемая и проверяемая извне деятельность, это то, как объект действует и реагирует, поведение выражается в терминах состояния объекта и передачи сообщений
- Состояние объекта представляет суммарный результат его поведения
- Класс – это множество объектов, имеющих общую структуру и общее поведение

Словарь ключевых терминов объектно-ориентированного программирования

- Объект – компонент системы, представленный собственной памятью и набором операций
- Метод – описание того, как выполнять одну из операций объекта
- Сообщение – запрос объекту на выполнение одной из его операций (“обращение к методу”)
- Класс – описание группы подобных объектов
- Экземпляр – один из объектов, описываемых классом

Язык программирования Си++.

Виды операций над объектами

- Модификатор – операция, изменяющая состояние объекта
- Селектор – операция, считывающая состояние объекта, но не меняющая состояния
- Итератор – операция, позволяющая организовать доступ ко всем частям объекта в строго определённой последовательности
- Конструктор – операция создания объекта и (возможно) его инициализации
- Деструктор – операция, освобождающая состояние объекта и разрушающая сам объект

Концепция абстрактного типа данных и её реализация в Си++

- Абстрактным типом данных называется тип данных с полностью скрытым (инкапсулированным) внутренним устройством
- *Работа с переменными абстрактных типов данных* происходит только через специально предназначенные для этого функции
- *Абстрактные типы данных реализуются* с помощью классов и структур, в которых нет открытых членов этих классов и структур, то есть внутреннее устройство данных которых полностью скрыто от пользователя

Язык программирования Си++.

Пример декомпозиции

```
struct student { char * name;    // Имя студента
                 int  year;      // Год обучения
                 double avb;     // Средний балл
                 int  student_id; // Номер зачётки
                 char * studentName ();
                 int  studentYearNumber ();
                 int  studentID ();
};
```

Свойства

*Операции
(методы)*

Определение класса в Си++

- Методы можно вызывать только для переменной соответствующего типа (класса), используя синтаксис доступа к членам структуры данных:

```
Today.init (9, 2, 2012);           Later.addm (5);
```

- При определении метода вне класса следует перед его именем указывать имя того класса, к которому он относится:

```
void Date::init (int dd, int mm, int yy)
    { d = dd; m = mm; y = yy; };
```
- Для объекта *Today* оператор *m = mm* означает *Today.m = mm*, в вызове метода *init* для объекта *Later* этот же оператор означает *Later.m = mm*

Определение класса в Си++

- Можно разрешать или ограничивать доступ к элементам класса из любых функций, не являющихся членами этого класса (методы класса всегда имеют доступ ко всем элементам класса):

```
class Date { int d, m, y; public: void init (int dd, int mm, int yy); };
```

- ***public*** – открытые элементы класса, то есть доступные всем внутри (методам класса) и вне определения класса
- ***private*** – закрытые элементы класса, то есть доступные только внутри определения класса (методам класса)
- ***protected*** – защищённые элементы класса, доступные только методам самого класса и его производных классов
- По умолчанию элементы класса, введённые с помощью ключевого слова ***class***, являются закрытыми, поэтому первую метку ***private*** ставить не обязательно

Определение класса в Си++

- Структуры (*struct*) и объединения (*union*) от классов (*class*) отличаются только правилами определения прав доступа по умолчанию к переменным и функциям класса: методы и данные структур (и объединений) по умолчанию открыты (*public*), а для классов – закрыты (*private*)
- Запись *struct имя_класса { ... };*
эквивалентна: *class имя_класса { public: ... };*
- Запись *class имя_класса { ... };*
эквивалентна: *struct имя_класса { private: ... };*

Работа с состоянием объекта

Встраиваемые функции

- Спецификатор ***inline*** указывает компилятору, что он должен попытаться вставить в место вызова функции саму её программу, а не команды формирования фактических параметров и обращения к функции с помощью стандартного (например, стекового) механизма:

```
inline bool get_declare () { return declare; };
```

- В некоторых случаях встраивание функций может приводить к сокращению размеров программы

Указатели и ссылки на объекты

- Кроме объектов существуют указатели на объекты:

```
int * p, i = 5; // определение указателя и переменной
p = & i; // установка значения указателя
i = i + 1;
cout << i << * p; // напечатается "6 6"
// так как i и * p – одно и то же
```

- Допустимы обычные указатели, указатели на константы, константные указатели и константные указатели на константы:

```
int * p; // простой указатель
int * const cp; // указатель-константа
const int * pc; // указатель на константу
int const * pc1; // указатель на константу (pc1 ≡ pc)
const int * const cc; // указатель-константа на константу
int const * const cc1; // указатель-константа на константу (cc1 ≡ cc)
```

Указатели и ссылки на объекты

- Объявление параметра константой запрещает функции менять его значение:

```
char * strcpy (char * p, const char * q);
```

- Присваивать адрес переменной указателю на константу разрешено, присвоить адрес константы простому указателю нельзя:

```
int a; // простая переменная
const int c = 2; // константа
const int * p1 = & c; // указатель на константу (+)
const int * p2 = & a; // указатель на переменную (+)
// с запретом на её изменение
int * p3 = & c; // простому указателю присваивается
// адрес константы, что запрещено в Си++ (-)
* p3 = 7; // делается попытка изменить константу c (-) 18
```

Указатели и ссылки на объекты

- Тип “ссылка на переменную”:

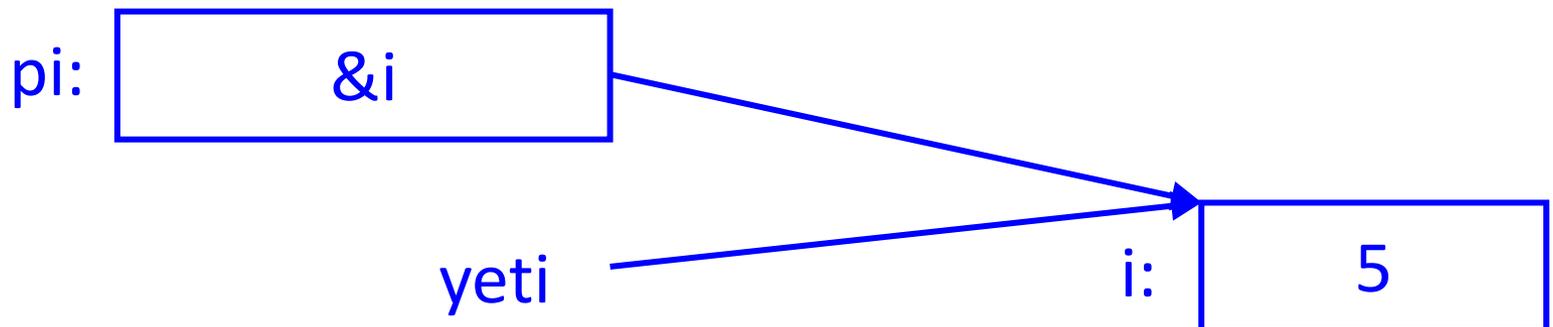
<тип> & <переменная>;

- Ссылка является синонимом имени объекта и представляет сам объект:

```
int    i = 5;      // Инициализация не обязательна
int & yet_i = i;   // Ссылка обязана иметь начальное значение!
        // i и yet_i ссылаются на одно и то же целое число.
        // Инициировать ссылку константным значением можно,
        // если сама ссылка тоже константная:
const int & yet_1 = 1; // ссылка на константу 1
        i = yet_i + 1;
cout << i << yet_i; // напечатается "6 6", поскольку
        // i и yet_i – одна и та же сущность
        yet_i = 10; // i == 10
        & yet_i == & i; // Адрес ссылки равен адресу самого объекта!
```

Указатели и ссылки на объекты

- Значение ссылки нельзя изменить после её инициализации
- Выражение `++yeti_i` не приводит к изменению ссылки. Операция увеличения применяется непосредственно к значению объекта, с которым связана ссылка, в данном случае к целому значению `i`



Передача параметров по ссылке

- На языке Си процедура перемены местами двух объектов выглядит так:

```
void swap (int * px, int * py)
{ int temp; temp = * px; * px = * py; * py = temp; }
```

- Вызов этой процедуры: `int a = 5, b = 6; swap (&a, &b);`

- Та же процедура на Си++:

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

- Вызов этой процедуры: `int a = 5, b = 6; swap (a, b);`

Конструкторы, как инициаторы полей данных классов

```
class Box { double len, wid, hei;
//для задания начальных значений всех трёх параметров параллелепипедов:
    public:   Box (double l, double w, double h)
                { len = l; wid = w; hei = h; }

// если часто используются кубики, то достаточно одного параметра:
    Box (double s)    { len = wid = hei = s; }

// если часто используются коробки одного типа, параметры не нужны:
    Box ()            { len = 24; wid = 12; hei = 6; }

// ещё один вариант конструктора умолчания:
    Box (double l = 24, double w = 12, double h = 6)
                { len = l; wid = w; hei = h; }

};
```

Конструкторы, как инициаторы полей данных классов

- Вызов конструкторов объектов при определении:
Box b1 (1, 2, 3);
Box b2 (5);
Box b3; // конструктор умолчания
~~Box b31 ();~~ // нельзя употреблять как конструктор:
// возникает путаница с описанием заголовка функции!
- Конструктор вызывается при явном создании объекта:
Box * b4 = new Box (2.3); // v1 = b4 -> volume ()
Box b5 = Box (); // конструктор умолчания
// здесь путаница с заголовком функции не возникает!²³

Свойства конструкторов объектов

- У одного класса может одновременно существовать несколько разных конструкторов, каждый из которых используется для инициализации элементов особым образом. Одновременное существование нескольких конструкторов имеет специальное наименование – *перегрузка конструкторов*
- Все конструкторы должны отличаться друг от друга количеством и/или типами параметров
- Конструктор без параметров называется *конструктором умолчания*. Если в классе не описан никакой другой конструктор (и только в этом случае), конструктор умолчания генерируется автоматически
- Конструкторы не возвращают никаких значений

Конструкторы копирования значений объектов

- Прототип конструкторов копирования выглядит так:
Box (Box &a); // или
Box (**const** Box &a);
- Наличие спецификатора **const** означает, что сам копируемый объект при копировании не изменяется
- Если конструктор копирования не определён в классе **явно**, он будет сгенерирован автоматически, при этом производится поверхностное (почленное) копирование данных объекта:

```
Box (const Box &a) { len = a.len; wid = a.wid; hei = a.hei; }
```

Запуск

конструктора копирования

- При определении новых объектов, иницируемых значениями ранее созданных объектов:

```
Box b4 = Box (4, 7, 1);
```

- При создании временного объекта для инициализации указателя на него (без копирования объекта!), но с последующей инициализацией описываемого объекта значением, извлекаемым по указателю:

```
Box * b5 = new Box (2, 3, 5);
```

```
Box b6 = * b5;
```

Свойства

конструкторов копирования

- Конструкторы копирования используются при передаче параметров и при возврате результата функции по значению
- При передаче параметров по ссылке копирования нет!
- Наличие явно описанного конструктора копирования блокирует возможность автоматической генерации конструктора умолчания
- Наличие явно описанного конструктора умолчания никак не влияет на автоматическую генерацию конструктора копирования
- В некоторых компиляторах при обработке инициализации вида `Box b4 = Box (4, 7, 1)` вместо конструктора копирования используется конструктор `Box b4 (4, 7, 1)` с тремя параметрами без создания временного объекта и его копирования

Указатель *this*

- Указатель *this* – это указатель на объект, от имени которого производится вызов метода
- Указатель *this* – это всегда указатель на самый первый (неявно заданный) операнд метода, который является объектом данного класса
- Пример использования указателя *this*

```
Box (Box & a) { if (this != & a)  
    { len = a.len; wid = a.wid; hei = a.hei; }  
} // это конструктор копирования
```

Использование указателя *this*

```
class string { char p [SIZE];           // ...
    public: void    concat (string &); // конкатенация строк
           int     length ();         // длина строки
           void    tolow  ();         // в нижний регистр
           void    trim   ();         // убрать пробелы
};
string s1, s2;
s1.concat (s2);           // присоединение s2 к s1
s1.trim ();              // удаление пробелов спереди и сзади
s1.tolow ();            // перевод в нижний регистр
string s3 = s1;         // копирование s1 в s3
```

- Здесь каждая последующая операция применяется к объекту *s1*, значение которого изменяется в результате предыдущей операции. Объект *s1* всегда является объектом класса *string*

Использование указателя *this*

```
class string { char p [SIZE];           // ...
    public: string & concat (string &); // конкатенация строк
           int      length ();         // длина строки
           string & tolow ();          // в нижний регистр
           string & trim   ();         // убрать пробелы
};
string & string::concat (string & s1) { strcpy (p + length (), s1.p);
    return * this; // разыменованное this, возврат ссылки на сам объект
}
// ... теперь можно писать так:
string s1, s2;
string s3 = s1.concat (s2).trim ().tolow ();
```

- Результатом работы метода *string::concat()* является ссылка на его же объект *s1*, это позволяет сразу вызвать метод *string::trim()*, который аналогичным образом возвращает ссылку на *s1*

Вызов конструкторов класса

- При создании объекта (при обработке описания объекта, при создании временных объектов в выражениях, при создании локальных объектов для значений параметров функций):

```
Box b (3, 4, 5);
```

- При создании объекта в динамической памяти (*new*), при этом, сначала отводится необходимая память, а затем работает нужный конструктор:

```
Box * pb;  
pb = new Box (3, 4, 5);
```

- При включении объектов в состав других объектов наряду с собственным конструктором вызывается конструктор объекта – члена класса
- При создании объекта производного класса дополнительно вызывается конструктор базового класса.
- Для статических объектов конструктор вызывается при запуске программы
- При возбуждении исключительной ситуации для создания временного объекта, передаваемого перехватчикам

При вызове конструктора класса

1. Вызываются конструкторы базовых классов (если есть наследование `class Z: public Y { ... }`)
2. Автоматически вызываются конструкторы умолчания всех вложенных объектов в порядке их описания в классе
3. Вызывается собственный конструктор, при его вызове все поля класса уже существуют, они получили свои значения при их инициализации, под них выделена память, *следовательно*, их можно использовать в теле конструктора

Вызов

конструкторов копирования

- В случае: `Box a (3, 4, 5);`
`Box b = a;` // a – параметр конструктора копирования
- В случае: `Box c = Box (3, 4, 5);`
Сначала создаётся временный объект и вызывается обычный конструктор (не копирования), а затем работает конструктор копирования при создании объекта 'c'. В некоторых компиляторах временный объект может не создаваться, в этом случае вызывается обычный конструктор с нужными параметрами:
`Box c (3, 4, 5);`
- При передаче параметров функции по значению (создание локального объекта)
- При возврате результата работы функции в виде объекта
- При возбуждении исключительной ситуации и передаче объекта-исключения перехватчику

Деструкторы объектов

- Деструктор не имеет параметров и не возвращает никакого значения: `~<имя_класса>() ;`
- Если деструктор не описан, он генерируется автоматически
- Деструктор по умолчанию ничего не делает (его тело – пусто) и нужен только для парности соответствующему конструктору
- Деструктор вызывается:
 1. При выходе из зоны описания объекта
 2. При выполнении операции ***delete*** для указателя, получившего значение при выполнении операции ***new*** (при этом сначала работает деструктор, а затем освобождается память, занятая объектом)

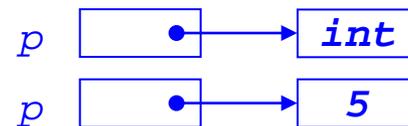
Деструкторы объектов

- При выходе из блока для всех автоматических объектов вызываются деструкторы, в порядке, противоположном порядку выполнения конструкторов:
 1. Собственный деструктор, при этом в момент начала работы программы деструктора поля данных класса ещё не очищены и, следовательно, доступны для использования, доступны в деструкторе также поля базовых классов (если есть наследование)
 2. Деструкторы всех вложенных объектов в порядке, обратном порядку их описания в классе
 3. Деструкторы базовых классов (если есть наследование)

Работа с динамической памятью

- Объекты размещаются в памяти аналогично данным языка Си: существуют статические объекты и автоматические объекты, выбор определяется местом определения объектов (**не классов!**)
- Вне процедур могут определяться только статические (***static***) и глобальные объекты, внутри – ещё и автоматические объекты
- Для создания объекта в свободной памяти используется операция ***new***, для его уничтожения применяется операция ***delete***
- Размер пространства, захватываемого при выполнении операции ***new***, точно соответствует размеру объекта-операнда

```
int * p = new int; // создание нового объекта
* p = 5; // присваивание значения объекту
```



- Явное создание нового объекта в свободной памяти конструктором:

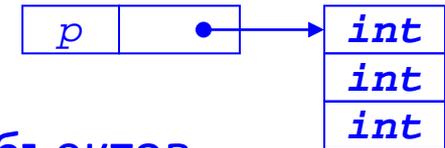
```
Box * b4 = new Box (2, 3, 5);
delete b4;
```
- Операции ***new*** и ***delete*** можно переопределять

Работа с динамической памятью

- Создавать можно и массивы объектов. В этом случае операция создания обозначается как ***new[]***

```
int * p;
```

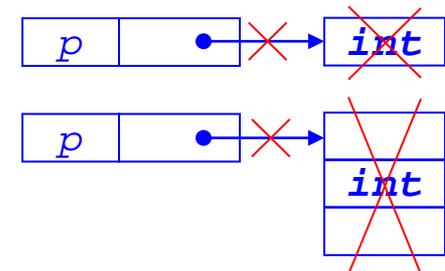
```
p = new int [3]; // создание массива из 3-х объектов
```



- Массив уничтожается операцией ***delete[]***
- Размер освобождаемой памяти контролируется системой и не указывается в операции ***delete[]***

```
delete p; // уничтожение скаляра
```

```
delete [] p; // уничтожение массива
```



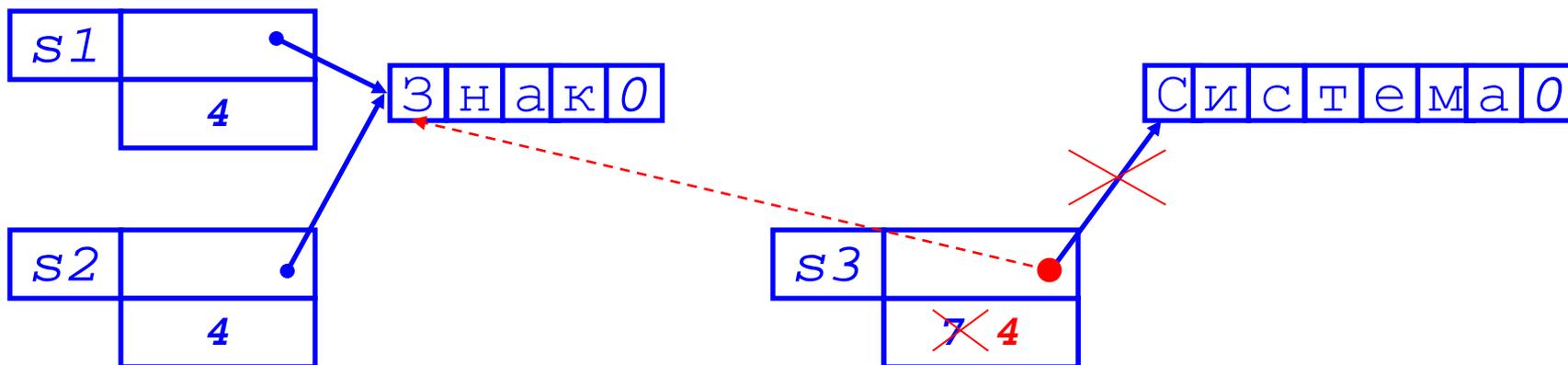
- Применение скалярного варианта операции освобождения памяти к указателю, по которому размещается массив приведёт во время работы программы к ошибке
- Операции ***new[]*** и ***delete[]*** можно переопределять

Работа с динамической памятью

```
class string { char * p; int size;
public:    string (const char * str); // конструктор
           // выполняет выделение памяти под строку
           // и инициализацию этой памяти
           ~string (); // освобождение памяти
};
string::string (const char * str) // для инициализации константами
{ p = new char [(size = strlen (str)) + 1];
  strcpy (p, str);
}
string::~~string () { delete [] p; }
void f() { string s1 ("Знак");
          string s2 = s1;
          string s3 ("Система"); /* ... */  s3 = s1;  // ...
}
```

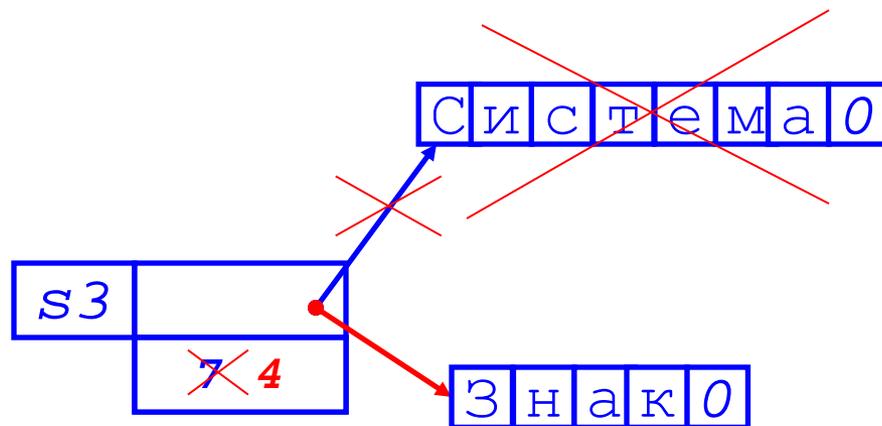
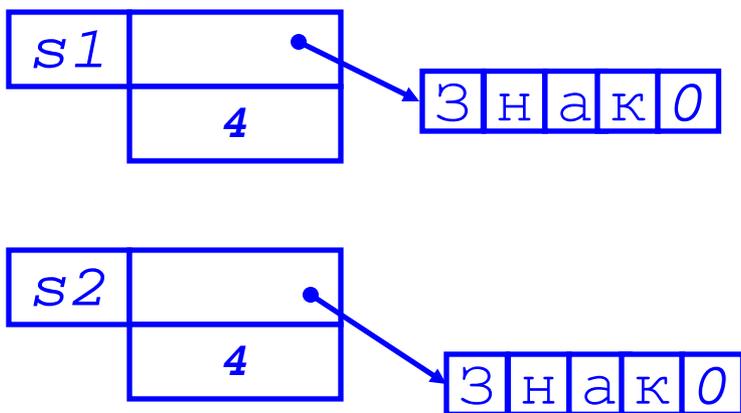
Работа с динамической памятью

```
void f() { string s1 ("Знак");  
          string s2 = s1;           // копирование полей p и size  
          string s3 ("Система");   /* ... */  
          s3 = s1;                 // присваивание полей p и size  
}
```



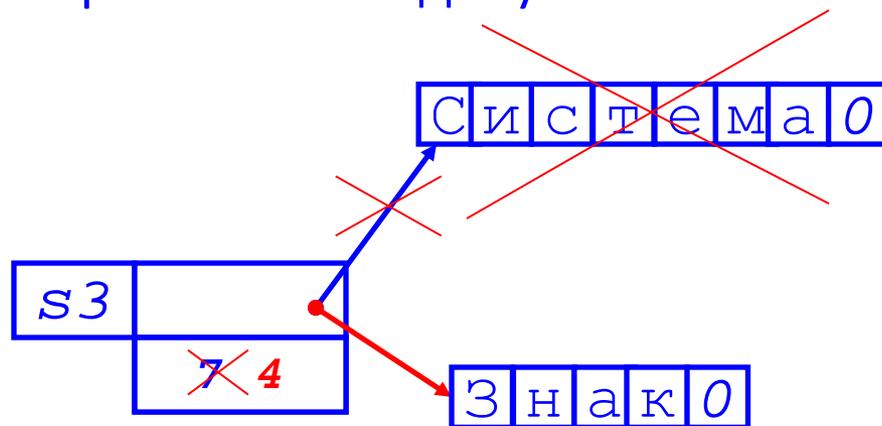
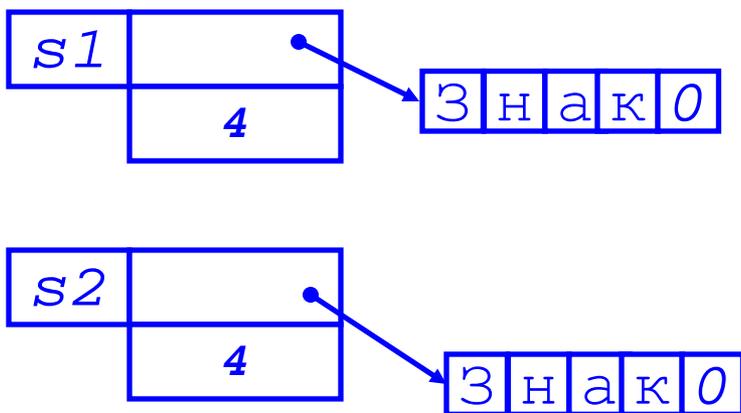
Работа с динамической памятью

```
string::string (const string & a)
{ // копируются не указатели, а значения:
  p = new char [(size = a.size) + 1];      strcpy (p, a.p); }
string::~~string () { delete [] p; }      // деструктор не меняется
void string::operator=(string & a)
{ // присваиваются не указатели, а значения:
  delete [] p; // уничтожение старого значения
  p = new char [(size = a.size) + 1];      strcpy (p, a.p);
}
```



Работа с динамической памятью

```
string::string (const string & a)
{   p = new char [(size = a.size) + 1];           strcpy (p, a.p); }
string::~~string () { delete [] p; }
string & string::operator=(string & a)
{   if (this != & a) {
        delete [] p; // уничтожение старого значения
        p = new char [(size = a.size) + 1];   strcpy (p, a.p);
    } return * this;
}   s3 = s2 = s1; // теперь такое присваивание допустимо
```



Константные члены класса

- Иногда необходима уверенность, что метод не будет менять значения переданных ему параметров

```
string & string::concat (const string & s) { ... }
```

- Ошибочными являются операторы, в которых меняется значение таких параметров, ошибочными являются вызовы методов для таких параметров (константных объектов), поскольку нет информации о том, какие операции они производят над своими объектами:

```
int string::length (); // нет указания неизменности объекта  
string & string::concat (const string & s)  
    { /* ... */ s.length () /* ... */ } // Ошибка!
```

- В языке Си++ можно указывать, что некоторые методы не меняют состояния своих объектов (ключевое слово **const** после скобок группирования формальных параметров):

```
int string::length () const;
```

Константные члены класса

- Как только метод объявлен константным, ему явно разрешается быть вызванным для константных объектов типа *string*. Тем самым, станет допустимой реализация метода *string::concat ()* с константным формальным параметром:

```
int string::length () const;           // длина строки
string & string::concat (const string &s) // конкатенация строк
{
    string temp = * this;
    delete [] p;
    p = new char [(size = temp.length () + s.length ()) + 1];
    strcpy (p, temp.p);    strcpy (p + temp.length (), s.p);
    return * this;
}
```

разрешено для константы "s",
если метод length() тоже константный!

Статические члены класса

- В классах допускаются статические члены (поля данных и методы)
- Статические методы классов не имеют неявного параметра, соответствующего указателю на активный объект, в теле статических методов класса нельзя использовать указатель *this*
- Для статического члена класса память отводится один раз в момент запуска программы, как для обычной статической переменной
- Статическое поле представляется в единственном экземпляре для всех экземпляров этого класса
- Обращаться к статическому члену можно без указания имени объекта, во избежание неоднозначности можно использовать имя самого класса:

```
class A { public: static int x; static void f (char c); };  
/* ... */  
A::x = 10; A::f ('a');
```
- Для работы со статическими полями используют статические методы

Статические члены класса

- При работе со статическими полями классов не возникает необходимости выделять память под экземпляры класса:

```
class X { public: int a; static int count;  
        X (int i) { a = i; count ++; }  
};
```

- Инициализация статических полей выполняется вне описания класса:

```
int X::count = - 1;  
int main () { X g(1), z (10); cout << X::count; }
```

- Обращение к статическому члену класса может происходить с помощью какого-либо из объектов (*g.count*, *z.count*), либо с помощью имени класса (*X::count*), которое можно использовать даже до определения первого объекта: вызов через объект (*g.count*) возможен только после оператора определения этого объекта (*g*)

Статические члены класса

- Инициализацию вне класса можно использовать даже, если статическое поле определено в закрытой части класса:

```
class X { static int count; public: int a;  
    X (int i) { a = i; count ++; }  
    void c_print () { cout << count << endl; }  
};  
int X::count = -1;
```

- Обычные методы нельзя вызывать до создания первого объекта, чтобы вызвать функцию до создания первого объекта, её следует объявить статическим членом класса:

```
static void c_print () { cout << count << endl; }
```

- Статическим методом класса можно объявить только такой метод, в котором используются только статические поля данных класса или другие статические методы
- Вызов статического метода: `X::c_print ();`

Статические члены класса

- Статические методы класса не могут вызывать нестатические методы, нестатические методы могут вызывать статические
- Константные методы класса не могут изменять значения полей данных класса, однако, они могут менять статические поля
- Статический метод может создавать экземпляры классов (объекты), в частности, экземпляры собственного класса:

```
class X {    X () {}    ~X () {}  
    public: static X& createX () {    X * x1 = new X;  
                                    cout << "X created";    return * x1; }  
        static void destroyX (X & x1) { delete & x1;  
                                    cout << "X destroyed";    }  
};  
  
int main () { X & xx1 = X::createX ();    // ...  
            X::destroyX (xx1); return 0; }
```

Статические члены класса

- Объекты, имеющие статические члены, могут быть объявлены константами

```
class X { public: static int c;  
};  
const X xc; // xc - константа
```

- Поле 'c' константой при таком определении объекта 'xc' не становится. С ним можно работать, как с обычной переменной
- Чтобы объявить статическое поле данных константой, это надо сделать явно:

```
class Z { public: const static int c;  
};  
const int Z::c = 25; // задано значение поля 'c'  
const Z zc; // 'zc' – константа, поле 'c' – тоже
```

Перекрытие имён

- Разрешается использовать одинаковые имена для разных объектов (переменных, констант, функций)
- Области видимости в программах могут быть вложенными друг в друга, определение объекта во вложенной области видимости отменяет (перекрывает) определение одноимённого объекта в объемлющей области
- Перекрытие позволяет использовать в разных (вложенных друг в друга) областях видимости одинаковые имена для разных, но одновременно не используемых объектов

Перегрузка функций

- **Перегрузка функций** – механизм, позволяющий одинаково именовать функции, видимые в одной области
- **Перегрузка функций** – проявление статического (полностью разрешаемого на стадии компиляции программ) полиморфизма
- **Перегрузка функций** позволяет использовать в одной области видимости одинаковые имена для разных, но одновременно не используемых функций

Перегрузка функций. Конфликты

- Неоднозначность выбора нужной функции:

```
void f (char);           void f (double);  
void g () {... f (1); ...}
```

- Отсутствие подходящей функции:

```
class X { ... };        class Y { ... };  
void f (int);           void f (Y);  
void g () { ... X a; f (a); ...}
```

- Скрытая неоднозначность выбора функции:

```
void f (int x = 1);     void f ();  
void g () {... f (); ...}
```

- Тип возвращаемого значения функции не участвует в выборе обслуживающей функции:

```
float sqrt (float);     double sqrt (float);
```

Правила работы алгоритма выбора перегруженной функции

- а). Точное отождествление
- б). Отождествление при помощи стандартных целочисленных и вещественных расширений
- в). Отождествление с помощью стандартных преобразований
- г). Отождествление с помощью пользовательских преобразований
- д). Отождествление по эллиптической конструкции “...”
(обычно этот шаг выполняется только при вызовах функций с несколькими параметрами)

Выбор перегруженной функции

а). Точное отождествление

- Точное совпадение типов
- Совпадение с точностью до *typedef* (новый тип не создается!)
- Тривиальные преобразования:
 - Параметр-массив: $T [] \leftrightarrow T *$
 - Передача параметра по ссылке: $T \leftrightarrow T \& \text{ (const } T \leftrightarrow \text{const } T \& \text{)}$
 - Передача переменной на месте формального параметра-константы: $T \rightarrow \text{const } T$
(обратное преобразование не рассматривается)
 - Параметр-функция (через имя функции и через указатель на функцию): $T (...) \leftrightarrow (T^*) (...)$

Выбор перегруженной функции

а). Точное отождествление

```
void f (int);      void f (float);  void f (double);  
void f (int unsigned);  void f (unsigned long);
```

```
void g () { f (1.0); /* f (double) */ f (1.0F); // f (float)  
           f (1);   /* f (int) */     f (1U);   // f (unsigned int)  
           f (1UL); }                // f (unsigned long)
```

```
void f (int * a); <= void g () { int m [5]; f (m); f (& m [0]); }
```

```
void f (int & a); <= void g () { int m; f (m); }
```

```
void f (int a); <= void g () { int m, & k = m; f (k); }
```

```
void f (const int * a); void g () { int k, * m = & k; f (m); }
```

```
void f (int * a); void g () { const int k = 7;  
                               const int * b = & k; f (b); }
```

Выбор перегруженной функции

б). Стандартные расширения

- Целочисленные расширения:
 - `char` (signed/unsigned), `short` (signed/unsigned),
`enum` => `int` (unsigned int, если тип `int` слишком узок)
 - `enum` => `long int` (unsigned long int, если значения не могут быть представлены типом `long int`)
 - `bool` => `int` (false --> 0, true --> 1)
 - Если возможно, то **битовые поля** могут расширяться до `int` или `unsigned int`, иначе они не расширяются
- Расширения чисел с плавающей точкой:
 - `float` => `double`

Выбор перегруженной функции

б). Стандартные расширения

- Расширения типов всегда имеют предпочтения перед другими стандартными преобразованиями

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

- Нет неоднозначности, хотя:

short => int & double float => int & double

- Если бы стандартные преобразования не делились на пункты б и в, то неоднозначность возникала бы, поскольку все преобразования одного шага работы алгоритма равноправны

Выбор перегруженной функции

в). Стандартные преобразования

- Все оставшиеся стандартные преобразования:
 - все оставшиеся неявные стандартные целочисленные и вещественные преобразования
 - преобразования указателей и ссылок:
 - 0 => любой указатель
 - любой указатель => свободный указатель (void *)
 - указатель (ссылка) на производный тип => указатель (ссылка) на базовый тип (для однозначного доступного базового класса наследственной иерархии)

Выбор перегруженной функции

в). Стандартные преобразования

- Все стандартные преобразования равноправны:

```
void f (char);  
void f (double);  
void g () { f (0); }
```

- Неоднозначность возникает из-за отсутствия явного совпадения (**пункт а**) и отсутствия стандартного расширения для типа *int* (**пункт б**): при выборе подходящей функции по правилам **пункта в** преобразования *int* => *char* и *int* => *double* равноправны

Выбор перегруженной функции

г). Преобразования пользователя

- Конструкторы преобразования (с одним параметром)
- Функции преобразования – нестатические методы класса с профилем “*operator <тип> ()*”, которые преобразуют объект класса к типу <тип>

```
struct S { S (long);           // конструктор преобразования: long --> S
          operator int ();    // функция преобразования:      S --> int
};
    void f (char *);          void g (char *);          void h (char *);
    void f (long);           void g (S);           void h (const S &);
void ex (S &a) { f (a); // f ((long) (a.operator int ())) = f (long)      (2 и 6)
                g (1); // g (S ((long) (1)))              = g (S)              (2)
                g (0); // g ((char *) (0))                = g (char *)         (6)
                h (1); // h (S ((long) (1)))              = h (const S &)     (2)
}
```

Выбор перегруженной функции

г). Преобразования пользователя

- Разрешается один раз повторно искать подходящий тип (**а**), расширение (**б**) или преобразование (**в**)
- Пользовательские преобразования применяются неявно только в том случае, если они однозначны

```
struct A { int a; public: A (int i) { ... }; } struct B { int b; public: B (int i) { ... }; }  
int f (A x, int y) { return x.a + y; } int f (B x, int y) { return x.b - y; }
```

Выражение *f(1, 1)* неоднозначно, поскольку может интерпретироваться двояко:

```
f (A (1), 1) /* f (A, int) */
```

```
f (B (1), 1) /* f (B, int) */
```

Выбор перегруженной функции

г). Преобразования пользователя

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

```
class X { ... public: operator int (); ... }; // можно преобразовать в int
class Y { ... public: operator X (); ... }; // можно преобразовать в X
void f (int x, int y);
void g () { Y a; int b; f (b, a); } // неправильно
```

- Требуется: `a.operator X ().operator int ()` – два шага по пункту г
- Если написать это явно, ошибки не будет: цепочки явных преобразований могут быть сколь угодно длинными

Выбор перегруженной функции

г). Преобразования пользователя

- Для использования в неявных преобразованиях и при поиске подходящей функции конструктор должен иметь разрешение на неявный вызов
- Неявный вызов конструктора может быть запрещён спецификатором *explicit*

```
class X { ... public: X (int); ... };  
void f () { X a (1);    // явный вызов, всегда правильно  
           X b = 2;    // неявный вызов  
           X c = X (2); // явный вызов с тем же эффектом  
}
```

Выбор перегруженной функции с несколькими параметрами

1. Отбираются все функции, которые могут быть вызваны с указанным в вызове количеством параметров
 2. Для каждого параметра вызова строится множество функций, оптимально отождествляемых по этому параметру
 3. Находится пересечение этих множеств:
 - если это ровно одна функция – она и является искомой,
 - если множество пусто или содержит более одной функции, генерируется сообщение об ошибке.
- При работе алгоритма для функций с несколькими параметрами обнаружение неоднозначности не приводит к немедленной остановке алгоритма, неоднозначность может быть снята рассмотрением других параметров функции

Выбор перегруженной функции

д). Выбор по ...

- Функции с переменным числом параметров работают в Си++ с помощью макроопределений *va_list* и *va_arg*

```
class Real { /* ... */ public: /* ... */ Real (double); /* ... */ };  
void f (int, Real);   void f (int, ...);  
void g () { f (1, 1);           // f (int, Real)           // пункт d даже  
                                // не привлекается к рассмотрению  
                                f (1, "Строка"); // f (int, ...)           // int, const char *  
}
```

- Многоточие может приводить к неоднозначности

```
void f (int);           void f (int ...);  
void g () { /* ... */ f (1); /* ... */ } // Ошибка, так как отождествление  
                                // по первому параметру даёт обе функции
```

Перегрузка операций

- Можно перегружать обычные арифметические и логические операции, операции отношения, операции присваивания, сдвиги, операции индексирования, разыменовывания указателей на структуры, операции доступа к членам класса через указатель на член класса, операции группировки параметров, захвата и освобождения свободной памяти (всего 42 операции):

	+	-	*	/	%	<<	>>	^		&
=	+=	-=	*=	/=	%=	<<=	>>=	^=	=	&=
	==	!=	!	,	<	<=	>=	>		&&
	++	--	->	->*	[]	()				
~	<i>new</i>		<i>new []</i>			<i>delete</i>			<i>delete []</i>	

Перегрузка операций

- Нельзя перегружать:

::	- разрешение области видимости
.	- выбор члена класса
.*	- выбор через указатель на член класса
?:	- тернарная условная операция
<i>sizeof</i>	- операция вычисления размера объекта
<i>typeid</i>	- операция вычисления имени типа
#	- начало макродирективы
##	- слияние или преобразование лексем

Перегрузка операций

- Для перегрузки операций используется ключевое слово *operator*, например, запись “*operator==*” обычно используется для определения операции проверки на равенство для объектов невстроенных типов
- При определении операций лучше сближать семантику новых операций с семантикой аналогично записываемых операций встроенных типов

Перегрузка операций

- Все перегрузки операций, необходимых в программе, нужно делать явно, для каждой операции (даже если некоторые операции эквивалентны):

`++ a`

`a += 1`

`a = a + 1`

- Имея функции, определяющие сложение `a.operator + ()` и присваивание `a.operator = ()`, нельзя автоматически, без явного определения, использовать операцию `'+='`, то есть функцию `a.operator += ()`
- Операции присваивания (`'='`), индексирования (`'[]'`) и доступа (`'->'`) могут перегружаться только нестатическими методами классов

Перегрузка бинарных операций

- Определение метода конкатенации в классе, созданном для работы со строками:

```
class string { char * p; int size;          /* ... */  
    string& concat (const string& s) { /* ... */  
};
```

- Реализация бинарной операции увеличения **'+='**:

```
string & operator+= (string & s1, const string & s2)  
                    { return s1.concat (s2); }
```

- Определение этой же операции в составе класса:

```
class string { /* ... */  
    string & operator += (const string & s) { return concat (s); }  
};  
    string s1 (" a"), s2 (" b");    s1 += s2;
```

Методы классов и функции-друзья классов

- Свойства (ограничения) обычных методов класса:
 1. Метод класса имеет право доступа к закрытой части объявления класса
 2. Метод класса находится в области видимости класса
 3. Метод должен вызываться только для объекта класса (имеется неявно присутствующий указатель *this*)
- Статические методы не имеют указателя *this* и могут вызываться, когда объектов у класса не существует
- Функции-друзья, обладая доступом к закрытой части объявления класса, не обязаны находиться в области видимости этого класса

Функции-друзья классов

- Иллюстрация различий в определении и использовании методов класса и функций-друзей:

```
class X { int a; public: ...  
        friend void friend_f (X*, int);  
        void member_m (int);  
}
```

```
void friend_f (X * p, int i) { p -> a = i; }  
void X::member_m (int i) { a = i; }
```

```
void f ()  
{ X obj;  
  friend_f (& obj, 10);  
  obj.member_m (10);  
}
```

Функции-друзья классов

- Преимущества использования функций-друзей:
 1. Функции-друзья повышают эффективность программ, позволяя отказаться от использования функций доступа к закрытым членам класса
 2. Объявление функции другом сразу нескольких классов позволяет упростить интерфейсы сразу всех этих классов
 3. Функция-друг не накладывает ограничений на списки собственных параметров и не требует делать главный параметр (используемый объект) первым в списке параметров функции
 4. Функция-друг допускает преобразования главного параметра (используемого объекта), а методы класса этого не допускают

Функции-друзья классов

- Если другом объявляется перегруженная функция, только она из одноимённых функций является другом
- Дружба не передаётся по наследству производным классам: она не обладает транзитивностью

```
class A { friend class B; int a; /* ... */ };  
class B { friend class C; /* ... */ };  
class C { void f (A *p) { p -> a ++; } }; // ошибка  
class D: public B { void f (A *p) { p -> a ++; } };  
// нет доступа к закрытому полю 'а'
```

Перегрузка операций

- Перегрузка может производиться для операций с параметрами встроенных типов:

```
class complex { double re, im; /* ... */  
    public: complex (double r = 0, double i = 0) { re = r; im = i; }  
    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;  
}
```

- Можно использовать операторы

```
complex y = 2, z; double d = 5.3;  
z = y * d; // вызывается функция operator*(y, d)
```

Перегрузка операций

- Ввиду отсутствия варианта перегрузки умножения с первым параметром, имеющим тип *double*, остаётся неверным оператор

```
z = d * y; // Ошибка
```

- Следует определить ещё одну дружественную классу *complex* функцию:

```
complex operator * (double a, const complex &b) {  
    complex temp (b.re * a, b.im * a);  
    return temp;  
}
```

Перегрузка операций

- Имея такие определения, можно написать программу:

```
complex x (1, 2), y (5,8), z;    const complex w (1, 3);  double d = 7.5;
z = x + y; /* x.operator + (y) */      z = x + w; // const внутри скобок
z = z + d; /* z.operator + (complex (d)) */ z = w + x; // const после скобок
z = d + x;      // ошибка: вызов d.operator+ (x), но в классе double
                // нет операции сложения с комплексными числами
```

- Методы класса в качестве своего первого параметра всегда имеют параметр, имеющий тип этого класса
- Функции, не являющиеся методами классов, свободны от этого требования
- Перегрузка операций членами класса обычно используется, когда оба операнда относятся к этому классу

Перегрузка операций

- Перегрузка операции '+' с помощью друга класса:

```
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)  
        { complex temp (re + a.re, im + a.im); return temp; }  
};
```

```
complex x (1, 2), y (5,8), z;    const complex w (1, 3);    double d = 7.5;  
z = x + y;    // operator + (x, y)  
z = z + d;    // operator + (z, complex (d))  
z = d + x;    // operator + (complex (d), x), нет ошибок
```

- Функции-друзья лучше использовать в тех случаях, кода в операциях участвуют операнды разных типов

Перегрузка операций

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

```
class X { public: operator int (); ... }; // можно преобразовать в int
class Y { public: operator X (); ... }; // можно преобразовать в X
void f () { Y a; int b;
           b = a; } // неправильно: b = a.operator X ().operator int ();
```

- Двойной шаг по пункту 2 алгоритма запрещён
- Если написать явные шаги преобразования типа, ошибки не будет: цепочки явных преобразований могут быть сколь угодно длинными
- Присваивание подчиняется тем же правилам, что и другие перегруженные функции

Операция индексирования

- Индексация '`[]`' есть бинарная операция, её перегрузка выполняется только нестатическим методом класса, у которого неявный параметр – это сам объект, к которому применяется операция, а явный параметр – значение индекса

```
class string { char * p; int size; /* ... */
    public: char & operator [] (int i);
        { if (i < 0 || i >= size) { cerr << "string:" << i << endl; exit (1); }
          return p [i];
        }
} s ("Системы программирования");
char c = s [3];    // эквивалентно c = s.operator [] (3); => c == 'т'
```

- Возвращаемое значение в виде ссылки позволяет использовать индексацию в присваивании и справа и слева

Перегрузка унарных операций

- Унарные операции перегружаются методом без параметров (точнее – с одним неявным параметром)

```
class complex { double re, im;  
    public: complex (double r = 0, double i = 0) {re = r; im = i;}  
    const complex operator- () const // -x = y x = -ComplexConst1  
        { complex temp (- re, - im); return temp; }  
};
```

```
complex x (1, 2), z;  
z = - x;    // z.re = - 1, z.im = - 2  
z = - 2;    // здесь нет никакой перегрузки унарного минуса:  
            // работает присваивание значения выражения -2
```

Перегрузка унарных операций

- Унарные операции перегружаются функцией-другом с одним параметром (допустимы и обычные функции)

```
class complex { double re, im;  
    public: complex (double r = 0, double i = 0) {re = r; im = i;}  
    friend complex operator- (const complex & a);  
};  
complex operator- (const complex & a)  
    { complex temp (- a.re, - a.im); return temp; }
```

```
complex x (1, 2), z;
```

```
z = - x;    // z.re = - 1, z.im = - 2: результат тот же
```

Особенности операций '++' и '--'

- При перегрузке нужно указывать, какой именно вариант перегружается (префиксный или постфиксный)
- Для двух операций языка определён специальный синтаксис:

$++a \equiv a.operator++ ()$ $a++ \equiv a.operator++ (0)$
 $--a \equiv a.operator-- ()$ $a-- \equiv a.operator-- (0)$

- Для указания на постфиксную форму используется фиктивный аргумент с типом *int*: *operator++ (int)*
- Как и все остальные унарные арифметические и логические операции (они все префиксные), префиксная операция определяется без фиктивного аргумента: *operator++()*
- Фиктивный аргумент используется только для "необычных" (единственных в своем роде) постфиксных операций '++' и '--'

Особенности операций '++' и '--'

- Префиксные операции обычно возвращают ссылки на объекты, а постфиксные – копии этих объектов
- Префиксная операция-функция получает ссылку на неконстантный объект, копия которого внутри функции не создаётся, а возвращает ссылку на константу, чтобы запретить операции вида `++++ Z` или `++ Z = ...`
- Постфиксная операция-функция возвращает константное значение, чтобы запретить операции вида `Z ++++` или `Z ++ = ...`; ссылка не возвращается: нужна копия неизменённого объекта

```
class complex { double re, im; /* ... */  
    const complex & operator++ () { ++ re; return * this; }  
    const complex operator++ (int pusto)  
        { complex temp = * this; re ++; return temp; }  
};
```

Особенности операций '++' и '--'

```
class complex { double re, im; /* ... */
public: complex (double r = 0, double i = 0) {re = r; im = i;}
      const complex & operator++ () { ++ re; return * this; }
      const complex operator++ (int pusto)
          { complex temp = * this; re ++; return temp; }
};
complex x (1, 2), y, z;
```

z = ++ x; // z.re = 2, z.im = 2, x.re = 2, z.im = 2

z = x ++; // z.re = 2, z.im = 2, x.re = 3, z.im = 2

++ ++ x; // **ОШИБКА**: возвращается не адресное значение

y = (x + y) ++; // **ОШИБКА**: сложение возвращает
// не адресное значение

Перегрузка операций

- Виды определений перегрузки операций:

```
class X { // члены класса (имеют неявный параметр this)
    X operator ++ (int); // постфиксная унарная операция увеличения
    X &operator ++ (); // префиксная унарная операция увеличения
    X * operator & (); // префиксная унарная операция взятия адреса ('&')
    X operator & (X); // бинарная операция логического И ('&')
    X operator & (X, X); // ошибка: заданы три операнда операции '&'
    X operator / (); // ошибка: нет унарной операции деления ('/')
}; // функции, не являющиеся членами класса
    X operator --(X&, int); // постфиксная унарная операция уменьшения
    X &operator -- (X&); // префиксная унарная операция уменьшения
    X operator - (X); // префиксная унарная операция изменения знака
    X operator - (X, X); // бинарная операция вычитания ('-')
    X operator - (); // ошибка: не заданы операнды операции '-'
    X operator - (X, X, X); // ошибка: заданы три операнда операции '-'
    X operator % (X); // ошибка: нет унарной операции остатка ('%')
```

Виды отношений между классами

- Классы могут рассматриваться как автономные абстрактные сущности, однако
 - в безусловном большинстве существующих программ классы взаимосвязаны
- Разновидности взаимосвязей (в том числе – иерархий):
 - Ассоциация классов
 - Агрегация классов
 - Использование одним классом другого класса
 - Инстанцирование (наполнение) класса
 - Наследование одним классом свойств другого класса

Основные понятия ER-модели

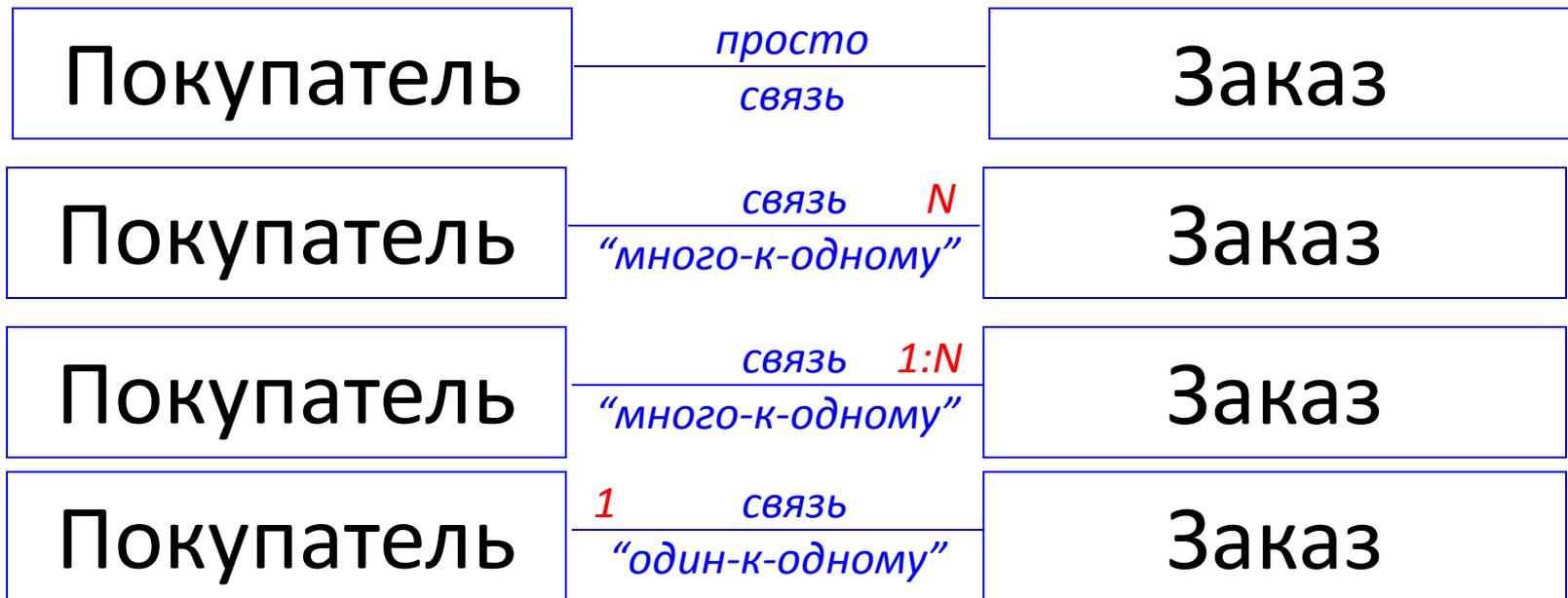
- *Сущность (Entity)* – абстракция, полученная на основе сходных объектов, информация о которых должна сохраняться и быть доступной
- В диаграммах ER-модели сущность представляется в виде прямоугольника, содержащего имя сущности



- Имя сущности – это имя типа (класса), а не некоторого конкретного *экземпляра* этого типа
- *Атрибут* – именованная характеристика сущности
- *Ключ сущности* – совокупность атрибутов, однозначно определяющая конкретный экземпляр

Основные понятия ER-модели

- *Связь (Relationship)* – графически изображаемая ассоциация, устанавливаемая между сущностями, связь – это типовое понятие, все экземпляры связываемых сущностей подчиняются единым правилам связывания



- *Модальность связи* – “должен” или “может”

Взаимодействие и иерархия классов. Ассоциация

- Ассоциация – наиболее общий вид взаимосвязи
На первых стадиях проектирования вид этой связи никак не конкретизируется, а утверждается лишь наличие некоторой связи, которая затем может проявиться в одном из других видов межобъектных связей
- Ассоциация всегда является бинарной и может существовать между двумя разными сущностями или между сущностью и ею же самой (рекурсивная связь)
- При последующем проектировании связь, выраженная как ассоциация, превращается в другую, более точно выраженную связь

Взаимодействие и иерархия классов. Агрегация

- Агрегация выражает отношение между классами по формуле “часть-целое” (“has a”, “содержит”)
- При агрегации объект одного класса внутри себя содержит объекты других классов, как объект “*Треугольник*” содержит три объекта “*Вершина*”, а объект “*Самолёт*” содержит четыре объекта “*Двигатель*”
- Различают **строгую** (сильную) и **нестрогую** агрегацию. При строгой агрегации внутренний компонент существует только одновременно с объектом. Строгую агрегацию иногда называют **композицией** (в этом случае нестрогую агрегацию называют **агрегацией** без каких-либо уточнений)

Взаимодействие и иерархия классов. Агрегация

- Нестрогая агрегация на языке Си++

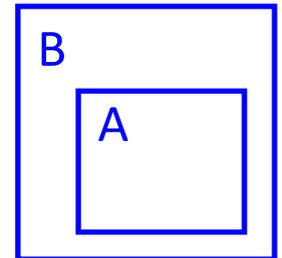
```
class ShareHolder { ... Share * asserts; ... }; // Акции могут исчезнуть,  
// но объект будет существовать
```



- Строгая агрегация или композиция на языке Си++

```
struct A { /* ... */};
```

```
struct B { struct A a; /* Класс B содержит класс A */};
```

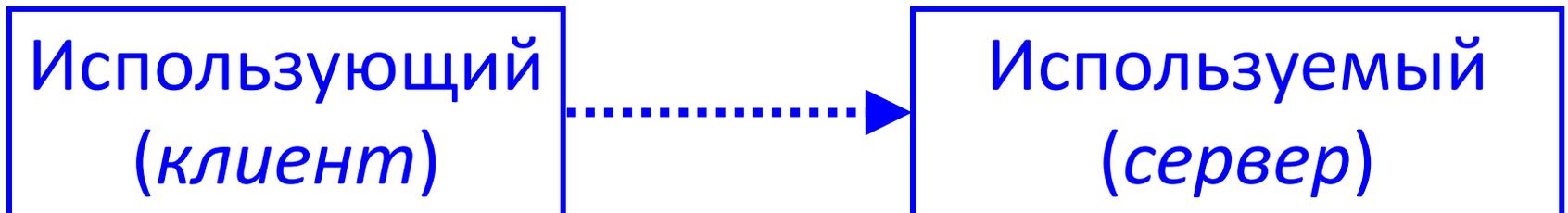


```
class Triangle { Point v1, v2, v3; ... }; // Вершины не могут исчезнуть,  
// пока существует экземпляр класса Triangle
```



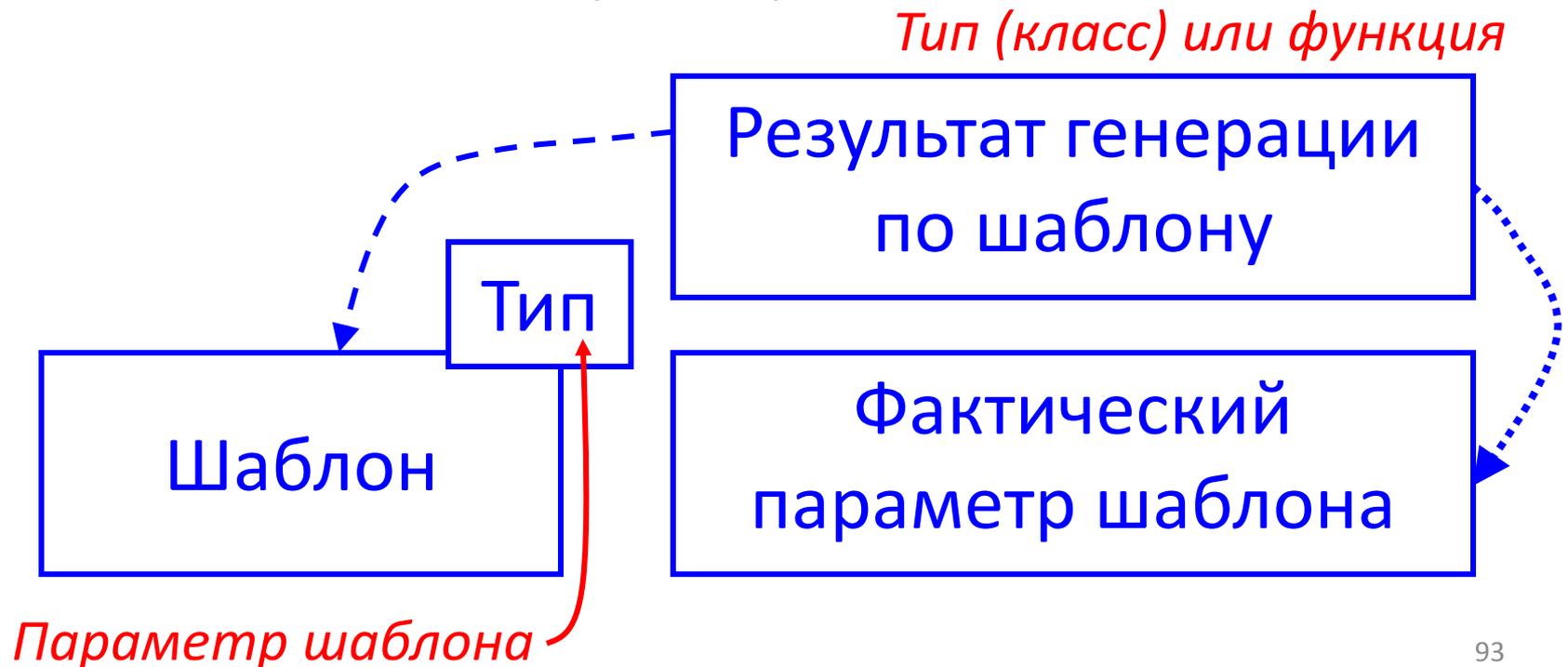
Взаимодействие и иерархия классов. Использование

- Имя одного класса используется в прототипе (профиле) метода другого класса
- В теле метода одного класса создаётся локальный объект другого класса
- Метод одного класса обращается к методу другого класса



Взаимодействие и иерархия классов. Инстанцирование

- **Инстанцирование** – проведение настройки шаблона с помощью типового параметра



Пример иерархии.

Одиночное наследование

- Концепция наследования: один класс является разновидностью (“частным случаем”) другого
- Иерархия наследования образует иерархию структур классов
- Иерархия структур классов определяет отношение “обобщение/специализация” (“is a”, “есть”)
- В иерархии классов вышестоящая абстракция является обобщением, а нижестоящая – специализацией

Наследование в Си++

- Цель – формирование иерархических связей между пользовательскими типами путём расширения базовых классов классами-наследниками
- Наследование выражает отношение вида “частное-общее” (“*is a*”, “*есть*”, “роза” *есть* “цветок”):
объекты производных классов рассматриваются как частные случаи объектов базовых классов
- Наследование – механизм создания нового класса (производного) из старого (базового):
“уточняется” определение базового класса, расширяется его представление и поведение

Наследование в Си++

- Состояния и поведение базового и производного классов связаны между собой
 - Производный класс наследует состояние (набор данных) и поведение (набор методов) от уже существующего базового класса
 - Производный класс может расширять состояние и поведение базового класса, то есть дополнять унаследованную структуру данных и унаследованный набор методов
 - Производный класс может переопределять поведение базового класса, то есть содержание унаследованных функций
 - Производный класс может корректировать доступ к членам базового класса
 - Базовый класс ничего не знает о наличии производного класса, но может предполагать его наличие и принимать меры по облегчению его создания
- Возникающий при наследовании класс есть подтип базового класса, объекты класса-наследника могут использоваться везде, где могут использоваться объекты базового класса

Наследование в Си++

```
struct A { int x; int y; }; A a;  
struct B: A { int z; }; B b;  
        b.x = 1;    b.y = 2;    b.z = 3;  
        a = b;
```

```
B: :  
  A: : int x;  
      int y;  
      int z;
```

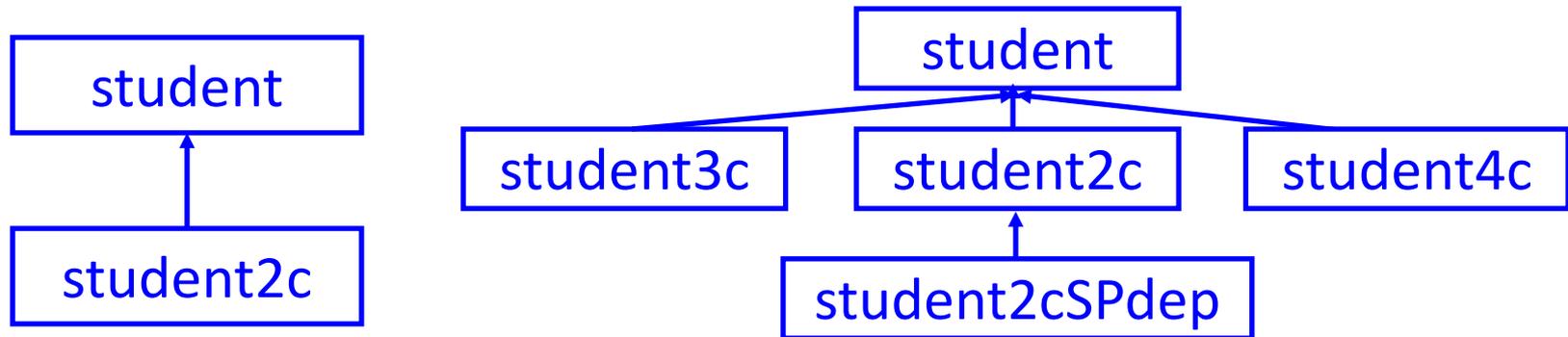
- Объект производного типа *B* наследует свойства базового типа *A*
- Допустимо присваивание $a = b$
- Обратное присваивание $b = a$ неверно: у объекта *a* нет достаточной информации для заполнения полей объекта *b*
- Классы *A* и *B* могут служить базовыми классами для других классов без ограничения их числа:

```
struct C: A { int t; };
```

```
struct D: B { int t; };
```

Наследование в Си++

- Иерархия классов, описывающих данные о студентах:



- Самый младший наследник изображается внизу иерархии
- Стрелка направляется от производных классов к базовому
- Число входящих стрелок для любого базового класса может быть сколь угодно большим, от этого наследование для каждого из порождаемых классов в отдельности не перестаёт быть **ОДИНОЧНЫМ**

Видимость при наследовании

- Наследование контролируется с помощью ограничивающих видимость спецификаторов доступа
- Спецификаторы доступа могут только уменьшить видимость, но не расширить её
- По умолчанию структуры наследуют своим базовым классам открытым способом, а классы — закрытым:

```
class C { public: int c; };
struct S { int s; };
class CC: C { CC () { c = 1; /* ОШИБКА! */ } // класс наследует закрытым
class CS: S { CS () { s = 1; /* ОШИБКА! */ } }; // образом,
struct SC: C { SC () { c = 1; } }; // а структура – открытым
struct SS: S { SS () { s = 1; } };
class CC: public C { CC () { c = 1; } }; // так надо исправить ошибки!
class CS: public S { CS () { s = 1; } };
```

Видимость при наследовании

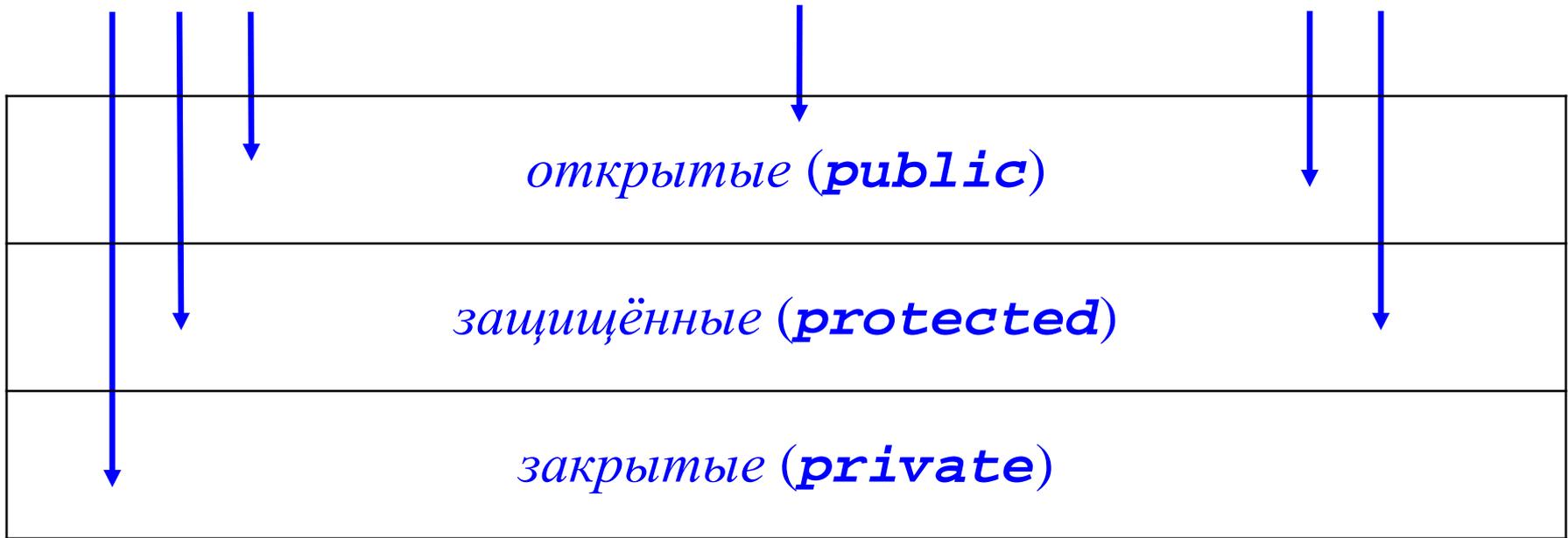
- Порождённый класс наследует все члены базового класса и имеет свободный доступ к открытым (*public*) и защищённым (*protected*) членам базового класса
- Причиной введения защищённых членов класса является необходимость открыть доступ к некоторым из элементов базовых классов, недоступным для посторонних
- Защищённые члены базового класса ведут себя как открытые по отношению к методам производных классов и как закрытые по отношению к другим функциям
- Закрытые (*private*) элементы базового класса недоступны для всех, включая методы производных классов

Видимость при наследовании

методы и
друзья
самого
класса

посторонние
пользователи

методы и
друзья
производного
класса



Видимость при наследовании

- Ограничение видимости действует только на данный производный класс и на его возможных потомков

```
struct A { int x; int y; }; class D: protected A { int t; };  
D d, * pd = & d; // строится указатель на производный класс  
pd -> t; // ОШИБКА: доступ к закрытому полю D::t (d.t)  
pd -> x; // ОШИБКА: доступ к защищённому полю D::x
```

- Работа с указателем происходит в контексте прав доступа этого указателя:

```
A * pa = (A*) pd; // строится указатель на базовый класс  
pa -> x; // правильно: поле A::x – открытое
```

- Каждый класс создаёт собственную область видимости

Видимость при наследовании

- Если в производном классе объявлен метод, одноимённый методу базового класса с совпадающим профилем, то имеет место перекрывание метода

```
struct M { void f (int x); };      struct N: public M { int x; void f (int x); };
M m, *pm;                        N n, *pn;
pn = &n;                          pn -> f (1);          // вызывается N::f (1)
pm = (M*) pn;                     pm -> f (1);          // вызывается M::f (1)
```

- Если метод некоторого класса должен вызвать другой метод, описанный в одном из базовых классов, используется явная операция разрешения области видимости ‘::’

```
pn -> N::f (1);                   // вызывается N::f (1)
pm -> M::f (1);                   pn -> M::f (1);       // вызывается M::f (1)
```

- Явное использование объекта всегда трактуется однозначно:

```
n.f (1);                          // вызывается N::f (1)
m.f (1);                          // вызывается M::f (1)
```

Видимость при наследовании

```
int x = 7; void f (int);
class P      { int x; public: void f (int);          };
class Q: public P {          void f (int); void g  (); };
void Q::g()
{
    f (1);      // Вызов Q :: f (1)
    P :: f (1); // Вызов P :: f (1)
    :: f (1);   // Вызов глобальной void f (1)
    :: x = 1;   // Изменение глобальной переменной x
    x = 2;     // Кажется, что: ≡ :: x = 2, так как P :: x скрыто в базовом классе,
               // Но ОШИБКА, так как x класса P перекрывает глобальную x,
               // и в классе Q глобальная переменная x недоступна!
}
```

- При создании объекта типа *Q* вызывается конструктор *P::P ()*, затем собственный конструктор *Q::Q ()*
- При разрушении объекта типа *Q* вызывается деструктор *Q::~~Q ()*, затем деструктор базового класса *P::~~P ()*

Создание и уничтожение объектов при наследовании

- При создании объекта производного класса сначала создаются элементы данных базового класса, сам базовый класс, потом элементы данных производного класса, наконец, сам производный класс
- Уничтожение объектов проходит в обратном порядке: сначала уничтожается производный класс, его элементы, базовый класс, элементы базового класса
- Элементы данных создаются в порядке объявления в классе и уничтожаются в обратном порядке

Специальные методы при наследовании

- Конструкторы, деструкторы и операции присваивания (функции *operator=()*) не наследуются, в каждом классе их определяют заново:

```
struct E { E & operator = (int i) { /* ... */ }  
        E (int i)                { /* ... */ } /* ... */  
};  
struct F: E { /* ... */ };  
void f () { F x (1); // ОШИБКА: конструктор не наследуется  
           // и не проникает в производный класс  
           x = 2; // ОШИБКА: операция присваивания  
           // не наследуется  
}
```

Создание, инициализация и уничтожение подобъектов

- Объекты внутри объектов других классов:

```
class Point { int x; int y; public: Point (); Point (int, int); };  
class Z     { Point p; int z; public: Z (int); };
```

- Объекты создаются в порядке вхождения вложенных объектов в составной объект
- Конструктор составного объект вызывается, когда все подобъекты уже существуют и инициированы:

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

- Деструкторы вызываются в обратном порядке:

```
delete z;                   // ~Z (); ~Point ();
```

Список инициализации

- Инициализировать вложенный объект до вызова собственного конструктора с помощью списка инициализации:

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

```
class Z        { Point p; int z; public: Z (int); };
```

```
Z::Z (int c): p (1, 2) { z = c; } // есть аналогия с Point p (1, 2)
```

```
Z::Z (int c): p (1, 2), z (c) {} // но нельзя делать одно определение  
// в составе другого определения
```

- Если задан список инициализации собственных полей, будут вызваны конструкторы из списка с заданными параметрами
- Если для члена класса инициализация в списке не указана, для инициализации используется конструктор умолчания
- Конструкторы из списка вызываются ***в порядке включения элементов в состав класса, но не списка!***

Инициализация и присваивание

- Инициализация объекта отличается от присваивания этому объекту нового значения
- Присваивание может повторяться много раз, инициализация выполняется лишь однажды
- Присваивание константам невозможно, инициализация константы – процесс естественный
- Инициализация выполняется для ничем не заполненной памяти, присваивание должно правильно работать с уже созданным объектом, значение которого заменяется **НОВЫМ**
- В инициализаторах членов класса возможно задание начальных значений ссылкам и константам

Инициализация и присваивание

- В инициализаторах членов класса возможно задание начальных значений ссылкам и константам:

```
class error { int i; const int ci; int & ri;
public: error (int ii) { i = ii;
    ci = ii; // ОШИБКА – нельзя присваивать константе
    ri = i; } // ОШИБКА – ссылка должна быть инициирована
};
```

- Следует изменить конструктор и записать инициализацию таким образом, чтобы она не смешивалась с присваиванием:

```
error::error (int ii): ci (ii), ri (i) // здесь нет ошибок!
    { i = ii; } // это обычное присваивание
```

Инициализация полей данных

- Инициализации полей, унаследованных производными классами от базовых классов, можно не делать, если для этих полей достаточно вызывать конструкторы по умолчанию
- В список инициализации, имеющийся у конструктора производного класса, необходимо включать обращение к конструктору базового класса с параметрами, обычно строящимися на основе значений параметров соответствующего конструктора производного класса:

```
class T { int n; double x; /* ... */  
    public: T (int i, double y): n (i), x (y) { /* ... */ }  
};  
class U: public T { bool b; /* ... */  
    public: U (bool t, int i, double y): b (t), T (i, y) { /* ... */ }  
};
```

Инициализация полей данных

- Имена полей базового класса использовать в списке инициализации конструктора производного класса нельзя:

```
class T { int n; double x; /* ... */  
    public: T (int i, double y): n (i), x (y) { /* ... */ }  
};  
class U: public T { bool b; /* ... */  
    public: U (bool t, int i, double y): b (t),  
        n (i), x (y) { /* ... */ } // ОШИБКА  
// так можно инициализировать только собственные поля  
};
```

Одиночное наследование в Си++

- Определим базовый класс, описывающий атрибуты студентов, этот класс послужит основой создания некоторого производного класса, описывающего только студентов 2 курса

```
class student { protected: char * name; int year; // год обучения
                double avb;           // средний балл
                int student_id;
public: student (char* nm, int y, double b, int id):
            year (y), avb(b), student_id (id)
            { name = new char [strlen (nm) + 1];
              strcpy (name, nm); }
            char * get_name () const { return name; }
            void print ();
            ~student () { delete [] name; }
};
```

Одиночное наследование в Си++

- Студент второго курса обладает всеми атрибутами студента – именем, номером года обучения, средним баллом на экзаменах, у него есть и свои, только ему присущие атрибуты, например, наименование темы практической работы по Си++ и имя руководителя практикой:

```
class student2c: public student { // указание на базовый класс
    protected: char* pract; char* tutor;
    public: student2c (char* n, double b, int id,
                     char* p, char* t) : student (n, 2, b, id)
    { pract = new char [strlen (p) + 1]; strcpy (pract, p);
      tutor = new char [strlen (t) + 1]; strcpy (tutor, t); }
      void print ();
      ~student2c () { delete [] pract; delete [] tutor; }
};
```

Одиночное наследование в Си++

- У нового класса, уточняющего ранее введённый класс, должны быть свои конструкторы и деструктор
- Функция *print ()* производного класса скрывает одноимённую функцию базового класса (из другой области видимости)
- Полностью унаследован производным классом и может в нём использоваться селектор *get_name ()*
- В новом классе можно заново строить все методы (может быть, копируя фрагменты текста из определения базового класса), но можно воспользоваться уже сформированными методами базового класса, как сделано в этом примере, где конструктор производного класса обращается к конструктору базового класса для инициализации унаследованных полей, которые нельзя инициировать поимённо

Одиночное наследование в Си++

- Производный класс может устанавливать дополнительные ограничения на доступ к унаследованным от базового класса элементам, которые будут распространяться и на него самого и на все те классы, которые станут его наследниками
- Все открытые и защищённые элементы базового класса рассматриваются в производном классе как защищённые, а все закрытые элементы остались бы закрытыми:

```
class student3c: protected student { /* ... */ };
```

- Все элементы, унаследованные производным классом от базового, становятся закрытыми:

```
class student4c: private student { /* ... */ }; // эквивалентно:  
class student4c: student { /* ... */ };
```

Одиночное наследование в Си++

- Во вновь порождённом производном классе могут быть определены новые члены (как новые поля данных, так и новые методы): в конструкторе класса студентов 2 курса уже не нужно задавать год обучения, но необходимо задать тему и руководителя практикой
- При вызове конструктора для экземпляра класса студентов 2 курса сначала проработает конструктор базового класса *student ()*, а затем конструктор производного класса *student2c*, иерархически вложенного в базовый, который вправе ожидать, что имя студента, номер зачётной книжки, средний балл и год обучения будут уже определены, когда придётся заполнять поля темы и имени руководителя
- При работе производного деструктора *~student2c ()* поле года обучения ещё будет существовать

Одиночное наследование в Си++

- Функция *print ()* базового класса не видна объектам производного класса, но путём уточнения её имени именем класса, которому принадлежит функция, её можно сделать видимой, если она не имеет атрибута *private*

```
void student :: print ()
```

```
{ cout << "ФИО           = " << name           << endl;
```

```
  cout << "Курс           = " << year           << endl;
```

```
  cout << "Средний балл   = " << avb           << endl;
```

```
  cout << "Номер зачётки  = " << student_id << endl;
```

```
}
```

```
void student2c :: print ()
```

```
{ student :: print (); // выдаёт в файл name, year, avb, student_id
```

```
  cout << "Тема курсовой   = " << pract           << endl;
```

```
  cout << "Преподаватель  = " << tutor           << endl;
```

```
}
```

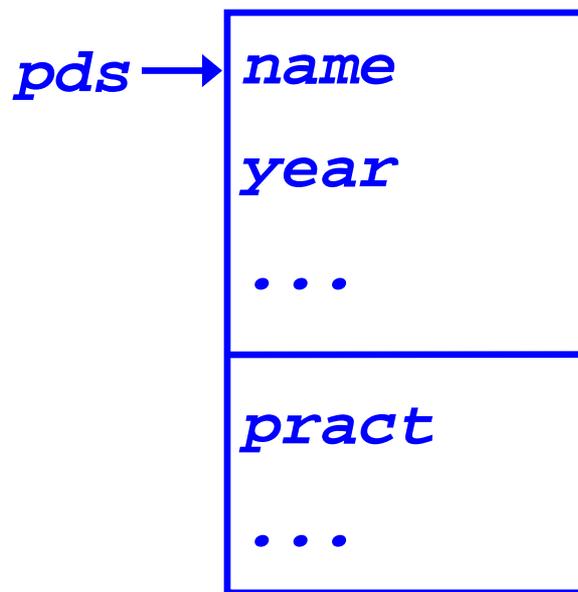
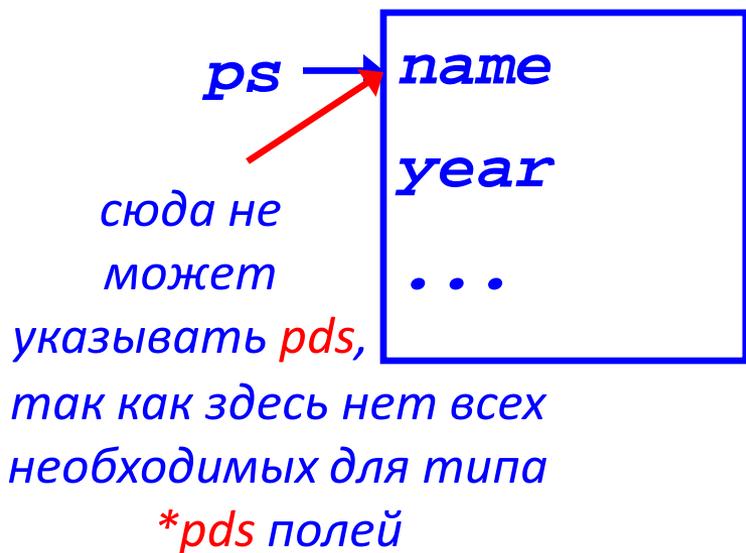
Одиночное наследование в Си++

- Манипуляции с объектами классов в глобальной функции $f()$:

```
void f ()
{ student  s ("Катя", 2, 4.18, 20100210);
  student2c ds ("Таня", 4.08, 20100211, "Компилятор Си++",
               "Виктор Петрович");

  student  * ps  = & s;    // указатель на базовый класс (курс = 2)
  student2c* pds = & ds;  // указатель на производный класс
  ps -> print (); // student  :: print () напечатает главное
  pds -> print (); // student2c :: print () напечатает всё
  ps = pds;      // допустимо (преобразование по умолчанию)
  ps -> print (); // student  :: print () выбирается
                  // статически по типу указателя
}
```

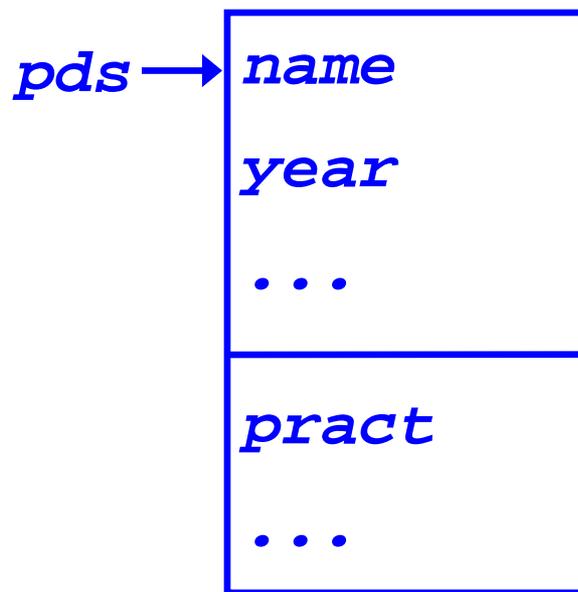
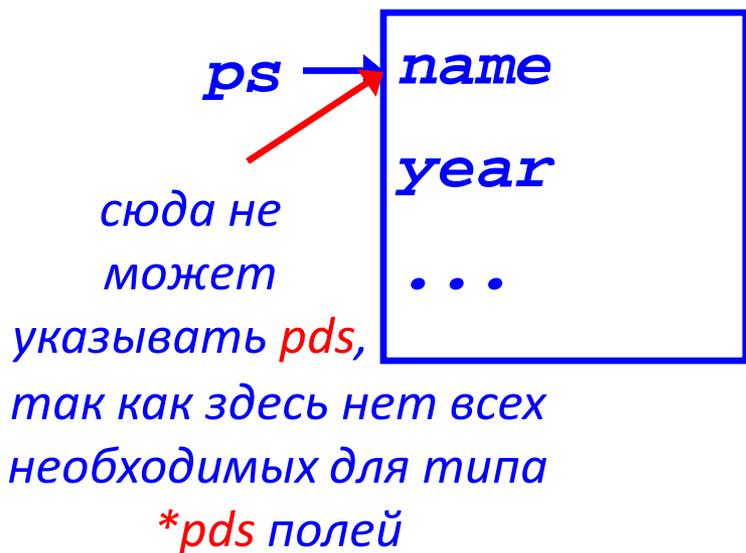
Генеалогические преобразования



← сюда может указывать *ps*, так как в этом объекте имеются все поля, относящиеся к типу **ps* однако, доступ ко всем полям по указателю *ps* невозможен, полный доступ возможен только по указателю *pds*

```
ps = & s; pds = & ds; ps = pds; // допустимо!  
pds = ps; // ОШИБКА: недопустимо!  
pds = & s; // ОШИБКА: недопустимо!
```

Генеалогические преобразования



```
ps = pds;    // base * = derived * (безопасно)  
ps = & ds;   // base * = derived * (безопасно)
```

С объектом производного класса можно обращаться как с объектом базового класса при обращении к нему при помощи базовых указателей и ссылок

Генеалогические преобразования

- Возможность доступа к объектам по указателям определяется статически по типу используемого указателя
- Правило статического выбора объекта по типу указателя работает даже в условиях длинных цепочек производных классов:

```
class X1 { public: void f (int); /* ... */ };  
class X2 : public X1 { /* ... */ }; /* ... */  
class X10 : public X9 { public: void f (double); /* ... */ };  
void g (X10 * p) { p -> f (2); } // X10::f или X1::f?
```

Виды полиморфизма

- Полиморфными называются методы, позволяющие выразить алгоритм один раз, но использовать его с множеством разных типов
- Статический полиморфизм действует только на этапе компиляции и характеризуется использованием одноимённых функций с разными профилями
- Параметрический (типовый) полиморфизм времени компиляции связан с введением шаблонов как средства параметрического описания классов
- Динамический полиморфизм реализуется с помощью виртуальных функций и доступа к функциям по указателям, когда на этапе компиляции не удаётся определить точную вызываемую функцию, так как значение указателя не известно

Динамический полиморфизм

- Поддержку динамического полиморфизма выполняют виртуальные методы, связанные с полиморфными объектами
- Виртуальные методы вводятся в классах, когда предполагается, что впоследствии будут определены производные от них классы, в которых эти виртуальные методы будут переопределены и сделаны конкретными, имеющими более точные реализации
- В каждом из производных классов переопределение виртуальных методов может быть своё, не похожее на сделанное в других классах
- Целью введения виртуальных методов является замена статического выбора метода (по типу указателя), на динамический выбор (по типу объекта, а не указателя)

Работа виртуальной функции

- *Полиморфным называется тип, в состав которого входят виртуальные функции*

```
class student { /* ... */ public: void print () const; };
class student2c: public student { /* ... */ public: void print () const; };
student  :: ~student () { delete [] name; }
student2c:: ~student2c() { delete [] pract; delete [] tutor; }
void student :: print ()
{ cout << "ФИО          = " << name          << endl;
  cout << "Курс          = " << year          << endl;
  cout << "Средний балл  = " << avb          << endl;
  cout << "Номер зачётки = " << student_id << endl; }
void student2c :: print ()
{ student :: print (); // выдаёт в файл name, year, avb, student_id
  cout << "Тема курсовой = " << pract          << endl;
  cout << "Преподаватель = " << tutor          << endl; }
```

Работа виртуальной функции

- Манипуляции с объектами классов в глобальной функции $f()$:

```
void f ()
{ student  s ("Катя", 2, 4.18, 20100210);
  student2c ds ("Таня", 4.08, 20100211, "Компилятор Си++",
               "Виктор Петрович");

  student  * ps  = & s;    // указатель на базовый класс
  student2c* pds = & ds;  // указатель на производный класс
  ps -> print (); // student  :: print () напечатает главное
  pds -> print (); // student2c :: print () напечатает всё
  ps = pds;      // допустимо (преобразование по умолчанию)
  ps -> print (); // student  :: print () выбирается
}                // статически по типу указателя
```

Работа виртуальной функции

- *Полиморфным называется тип, в состав которого входят виртуальные функции*

```
class student { /* ... */ public: virtual void print () const; };
class student2c: public student { /* ... */ public: void print () const; };
student  :: ~student () { delete [] name; }
student2c:: ~student2c() { delete [] pract; delete [] tutor; }
void student :: print ()
{ cout << "ФИО          = " << name          << endl;
  cout << "Курс          = " << year          << endl;
  cout << "Средний балл  = " << avb          << endl;
  cout << "Номер зачётки = " << student_id << endl; }
void student2c :: print ()
{ student :: print (); // выдаёт в файл name, year, avb, student_id
  cout << "Тема курсовой = " << pract          << endl;
  cout << "Преподаватель = " << tutor          << endl; }
```

Работа виртуальной функции

- Манипуляции с объектами классов в глобальной функции $f()$:

```
void f ()
{ student  s ("Катя", 2, 4.18, 20100210);
  student2c ds ("Таня", 4.08, 20100211, "Компилятор Си++",
               "Виктор Петрович");

  student  * ps  = & s;    // указатель на базовый класс
  student2c* pds = & ds;  // указатель на производный класс
  ps -> print (); // student  :: print () напечатает главное
  pds -> print (); // student2c :: print () напечатает всё
  ps = pds;      // допустимо (преобразование по умолчанию)
  ps -> print (); // student2c :: print () выбирается
}                // динамически по типу объекта
```

Правила создания виртуальных функций

1. Имеется иерархия классов (без иерархии нет виртуальности!), хотя бы из двух классов – базового и производного
2. В базовом классе функция объявлена с ключевым словом *virtual*
3. В производном классе есть функция с таким же именем, с таким же списком параметров (количество, типы и порядок параметров в точности совпадают) и с таким же типом возвращаемого значения
4. Вызов функции осуществляется через указатель или ссылку на объект базового класса без указания самого объекта и уточнения области видимости

Правила создания виртуальных функций

- В производном классе слово *virtual* может отсутствовать
- Простое использование слова *virtual* в базовом классе не запускает механизм виртуальности
- Нарушение условий приводит к тому, что вместо виртуальной функции возникает обычное перекрытие имён базового класса производным
- В базовом классе, как таковом, никаких виртуальных функций существовать не может

Правила создания виртуальных функций

- Типы возвращаемых значений могут иметь отличия: если в базовом классе этот тип есть тип указателя (ссылки) на базовый класс, а в производном классе – тип указателя (ссылки) на производный класс, то виртуальность всё же достигается
- Допускается исключение: если в виртуальной функции базового класса типом возвращаемого значения является указатель на сам тип этого класса, то в функции производного класса допускается иметь в качестве возвращаемого значения также указатель на тип производного класса:

```
class student { public: virtual student * cv () const; };  
class student2c:public student { public: virtual student2c * cv () const; };
```

Правила создания виртуальных функций

- Если в производном классе нет объявления функции, одноимённой с функцией, обозначенной в базовом классе как виртуальная, функция из базового класса наследуется, причём со словом *virtual*, то есть становится потенциальной виртуальной функцией на случай возникновения нового поколения наследующих классов

Правила создания виртуальных функций

- Вызов функции через объект (*x.f()*) приводит к раннему связыванию даже для виртуальных функций, виртуальности не возникает
- При непосредственной работе с объектами (без указателей и ссылок) их тип всегда известен:

```
void fvirt ()  
{ /* ... */  
    s.print (); // ≡ student    :: print (); // явные вызовы делают  
    ds.print (); // ≡ student2c :: print (); // полиморфизм ненужным  
}
```

Правила создания виртуальных функций

- При вызове методов из методов механизм виртуализации тоже работает, так как вызовы сопровождаются (часто неявными) операциями доступа ' \rightarrow ' по указателю *this*:

```
class A { /* ... */ public: int fnvir() { return fvirt ();} // this -> fvirt ()
    virtual int fvirt () { return 'A'; } };
class B: public A {public: int fnvir() { return fvirt ();}
    int fvirt () { return 'B'; } };
int main () { B b; A * p = & b; return p -> fnvir () - 'A';} // возвращается 1
```

- При явном указании класса виртуальность отключается, даже при вызове через указатель: `ps -> student :: print ()`

Виртуальные деструкторы

- Деструкторы не наследуются
- Деструкторы следует делать виртуальными
- Удаление объекта деструктором базового класса может привести к потере памяти, так как некоторые поля данных в наследнике не будут освобождены:

```
void f () { student2c * pds = new student2c ("Таня", 4.08, 20050211,  
                                           "Компилятор Си++", "Виктор Петрович");  
          student    * ps = pds;                               /* ... */  
          delete ps;  
          // вызовется ~student (), и некоторые поля не будут  
          // освобождены, так как компилятор не в состоянии  
          // знать состав объекта, на который указывает ps  
}
```

Виртуальные деструкторы

- Виртуальные деструкторы:

```
virtual    ~student ();    // в классе student
```

```
virtual    ~student2c (); // в классе student2c
```

- При выполнении операции *delete ps* будет вызван деструктор производного класса *~student2c ()* (сработает динамический полиморфизм)
- Особенность виртуальных деструкторов: имена у деструкторов базового и производного классов разные, но для деструкторов сделано исключение из общего правила (как если бы все они имели одно условное имя “Деструктор”)
- **Конструкторы не могут быть виртуальными**: производные классы не должны и не могут подменять конструкторы базовых классов

Пример виртуальной функции

- Виртуальные функции могут помочь при внедрении полиморфических свойств в такие объекты, которые изначально не обладали полиморфизмом
- Не являются полиморфическими операции ввода и вывода, определённые в стандартной библиотеке Си++ для потоков данных, что означает невозможность добиться полиморфности операции вывода в поток значения производного типа:

```
class Token { ... };  
class ident: public Token { ... };  
class number: public Token { ... };  
class Parser { Token * lex; /* ... */  
    public: /* ... */ { /* ... */ cout << lex; /* ... */}  
    // ОШИБКА: операция не определена
```

Пример виртуальной функции

- Полиморфизм внедряется в объект с помощью виртуальной функции базового класса:

```
class Token { public: virtual ostream& print (ostream& s) { /* ... */ }  
            friend ostream& operator << (ostream & s, Token * t)  
            { return t -> print (s); }  
};
```

- В производных классах функция определяется в том виде, который наиболее удобен для данного типа:

```
class ident:    public Token { public: ostream& print (ostream& s) {...}};  
class number: public Token { public: ostream& print (ostream& s) {...}};
```

- Точный тип лексемы (объекта ** lex*) не известен, но вид значения объекта будет определяться его типом

Пример виртуальной функции

- Если первым параметром операции является параметр непалиморфного типа, она может обратиться к функции, первым параметром которой является указатель на полиморфный класс
- Функция *print ()* является виртуальной, выбор её реализации будет осуществляться в программе не статически по типу указателя, а динамически по типу объекта, на который построен указатель:

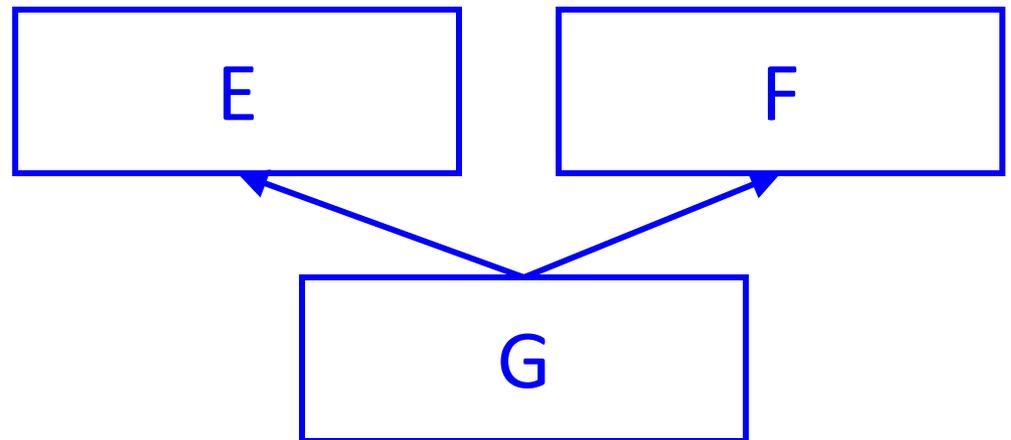
```
void f (Token * lex, ident * id, number * nmb)  
    { cout << lex << id << nmb; };
```

Пример иерархии.

Множественное наследование

- Множественное наследование есть одновременное наследование одним классом свойств сразу нескольких других классов

```
struct E    { /* ... */ };  
struct F    { /* ... */ };  
struct G: E, F { /* ... */ };
```



Множественное наследование

- Производный класс может быть создан на основе произвольного числа базовых классов:

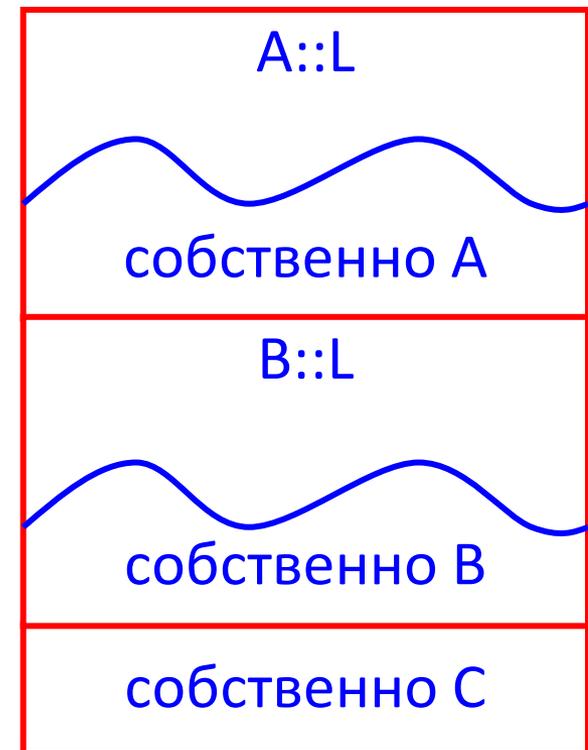
```
class A (/* ... */);    class B (/* ... */);    class C (/* ... */);  
class D: public A, protected B , private C (/* ... */);
```
- При множественном наследовании сохраняются основные принципы действия механизма наследования, но возникают дополнительные проблемы, связанные с потенциальной неоднозначностью множественного наследования
- Каждый спецификатор уровня доступа действует только на одно упоминание базового класса, для последующих классов из списка базовых снова начинает действовать принцип умолчания (***private*** при наследовании для класса, ***public*** при наследовании для структуры):

```
class D: public A, B { /* ... */ };
```

Множественное наследование

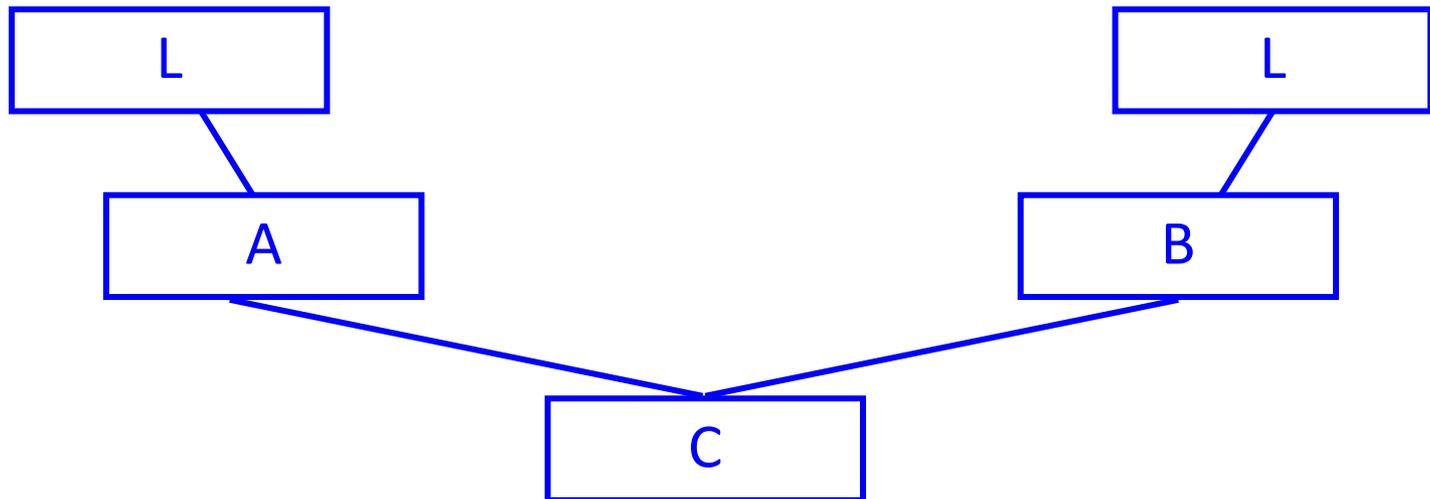
- В спецификации базовых классов ни один класс не может появиться дважды
- Непрямое повторное наследование вполне возможно и является обычной практикой программирования:

```
class L { public int n; /* ... */ };  
class A: public L { /* ... */ };  
class B: public L { /* ... */ };  
class C: public A, protected B  
        { /* ... */ void f (); /* ... */ };
```



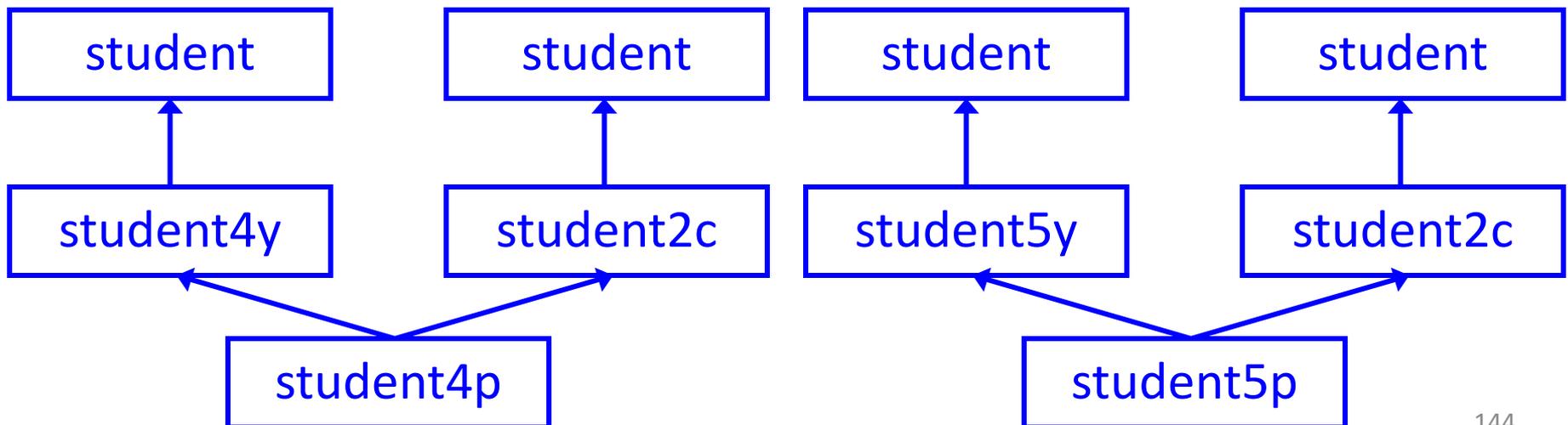
Множественное наследование

- Свойства класса L наследуются дважды (опосредованно), а значит, поля, относящиеся к этому типу, дважды войдут в состав полей класса C
- При работе с наследственными иерархиями составляют решётки смежности:



Множественное наследование

```
class student {protected: char * name; /* ... */ int student_id; };  
class student2c: public student {protected: char* pract; char* tutor;};  
class student3c: public student {protected: /* ... */ public: /*...*/ };  
class student4y: public student {protected: char* certificate; /*...*/ };  
class student5y: public student {protected: char* diploma; /*...*/ };  
class student4p: public student2c,public student4y{ bool test; /*...*/ };  
class student5p: public student2c,public student5y{ int exam; /*...*/ };
```



Множественное наследование

- Определение объекта производного класса:

`student5p АВ;`

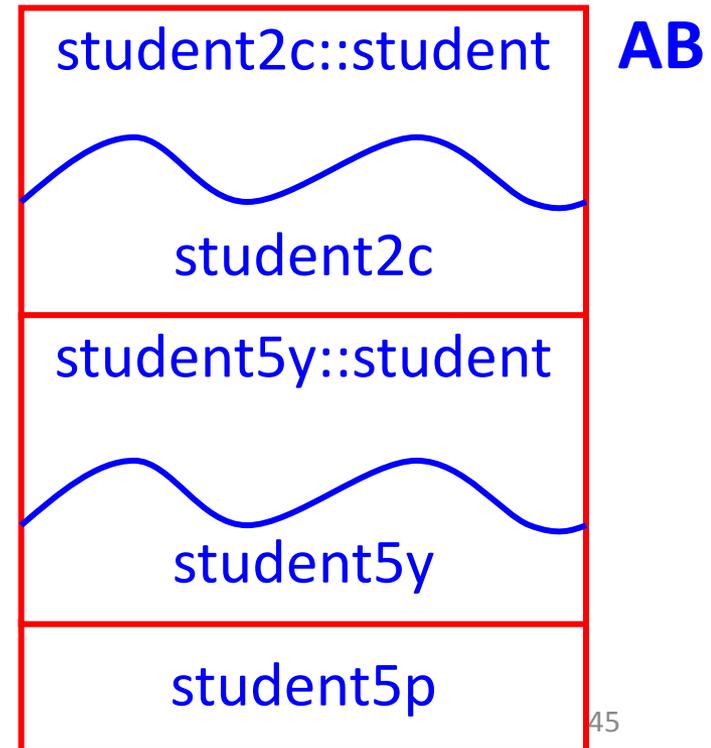
- Невозможно осуществить доступ к полю, содержащему средний балл, полученный студентом за время обучения (совершенно неясно, какое именно поле с именем *avb* имеется в виду):

`АВ.avb = 4.7;`

- Два таких варианта доступа к полям с этим именем:

`АВ.student5y::avb = 4.7; // или`

`АВ.student2c::avb = 4.7;`



Множественное наследование

- С учётом множественности наследования необходимо следить, чтобы неявное преобразование проводилось только тогда, когда базовый класс доступен и определяется однозначно:

```
void g ()  
{ student4p* p4 = new student4p;  
  student * ps = p4; // ОШИБКА, нет однозначности  
  ps = (student *) p4; // явное преобразование не помогает,  
} // так как выполнено только условие доступности
```

- Возможно такое преобразование (явное преобразование может быть управляемым):

```
ps = (student*)(student4y*) p4; // => student4y => student
```
- Правильным будет и такое преобразование:

```
ps = (student*)(student2c*) p4; // => student2c => student
```

Множественное наследование

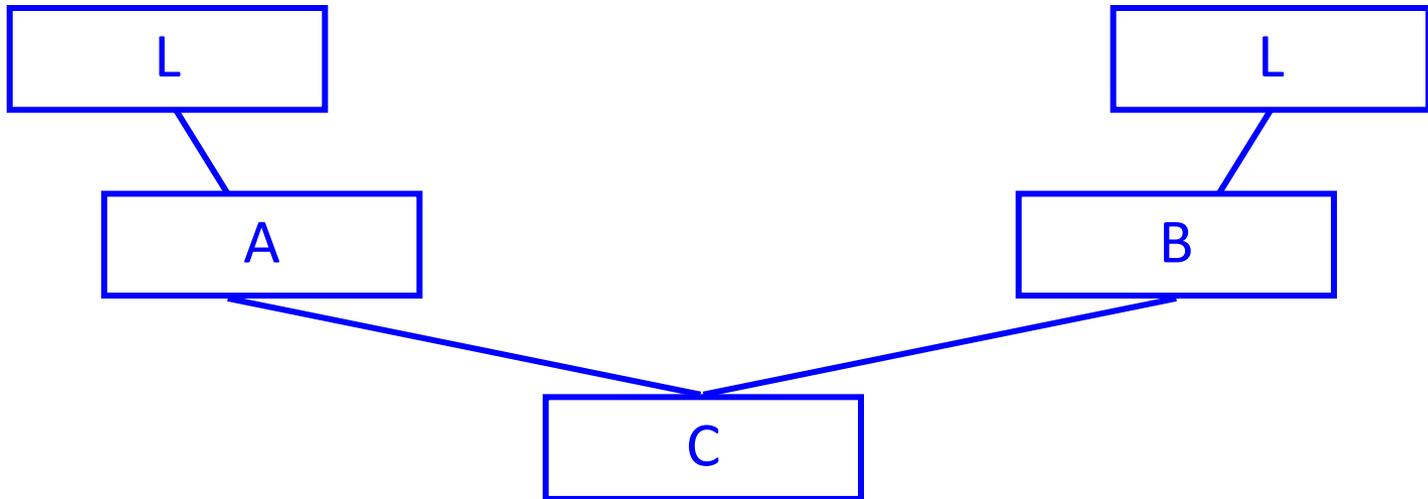
- Базовый класс считается доступным в некоторой области видимости, если доступны его открытые члены (поля и/или методы)

```
class B { public: int a; /* ... */ };  
class D: private B { /* ... */ };  
void g ()  
{ D * pd = new D;  
  B * pb = pd; // ОШИБКА, так как в g () открытые члены B,  
               // наследуемые классом D, недоступны  
}
```

- Показанное преобразование возможно, когда его выполняют метод класса *D*, либо функции-друзья класса *D*, так как в них класс *B* становится доступным

Виртуальное наследование

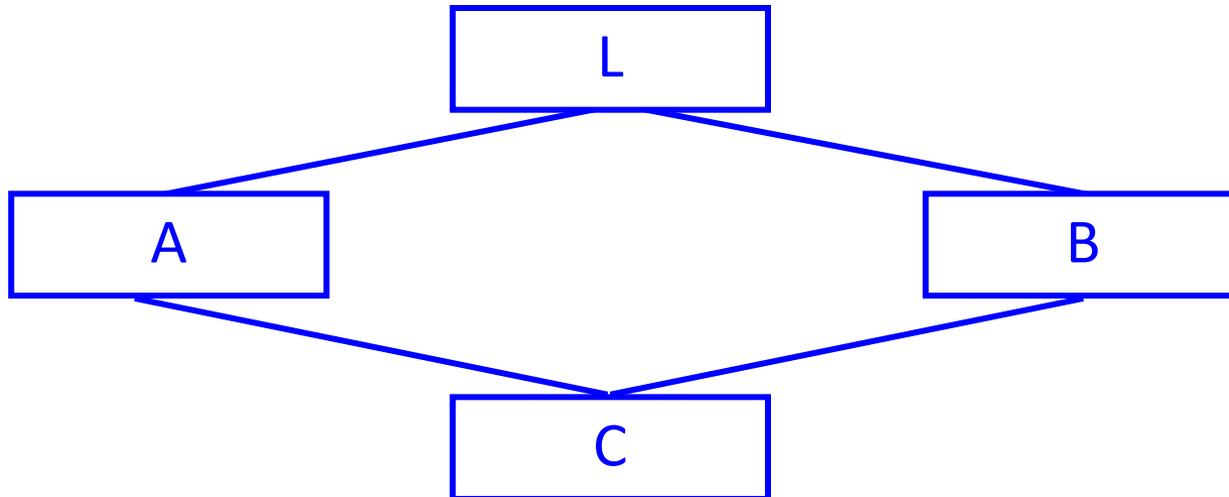
```
class L { public: int n; /* ... */ };  
class A: public L { /* ... */ };  
class B: public L { /* ... */ };  
class C: public A, protected B  
        { /* ... */ void f (); /* ... */ };
```



```
C c; c.n = 1; C * pc = new C; L * pl = pc;  
// нет ошибки: всё доступно и однозначно
```

Виртуальное наследование

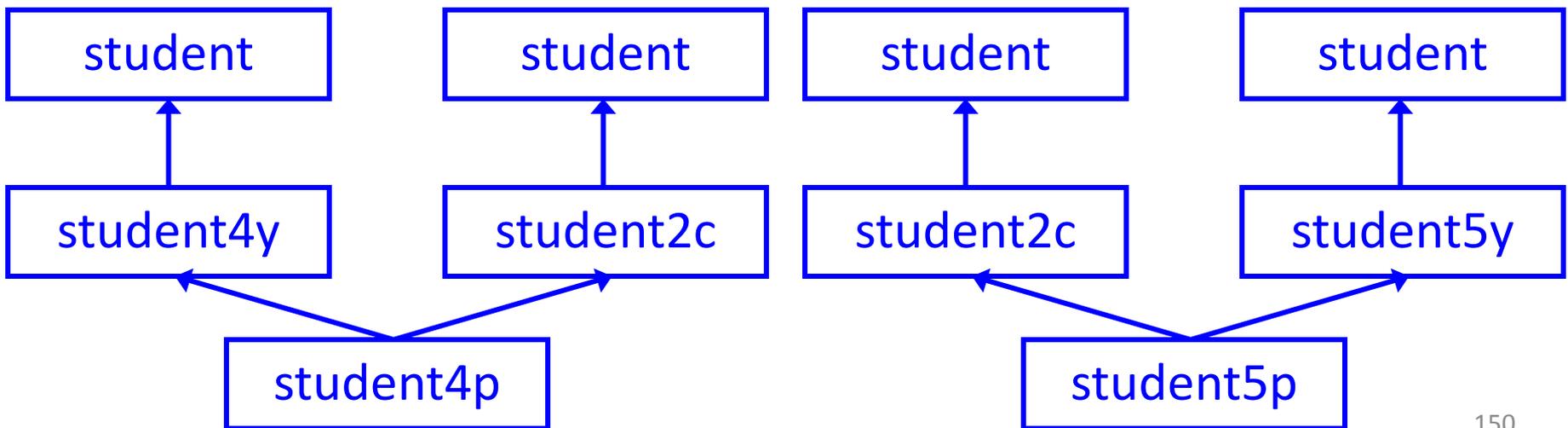
```
class L { public: int n; /* ... */ };  
class A: virtual public L { /* ... */ };  
class B: virtual public L { /* ... */ };  
class C: public A, protected B  
    { /* ... */ void f (); /* ... */ };
```



```
C c; c.n = 1; C * pc = new C; L * pl = pc;  
// нет ошибки: всё доступно и однозначно
```

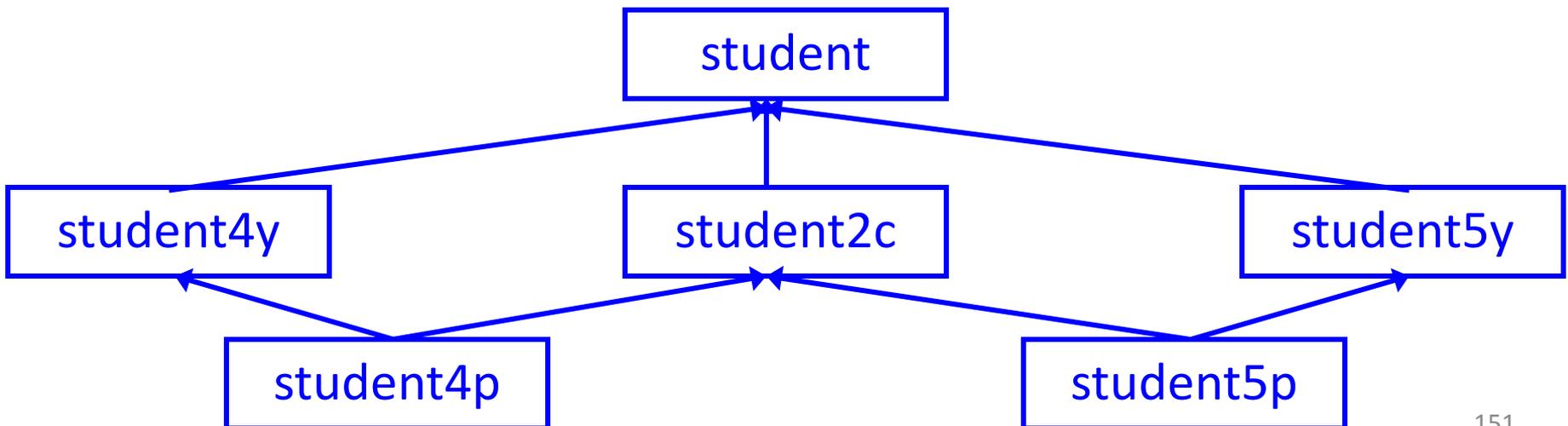
Виртуальное наследование

```
class student { protected: char * name; /* ... */ int student_id; };
class student2c: public student { protected: char* pract; char* tutor; };
class student3c: public student { protected: /* ... */ public: /*...*/ };
class student4y: public student { protected: char* certificate; /*...*/ };
class student5y: public student { protected: char* diploma; /*...*/ };
class student4p: public student2c,public student4y { bool test; /*...*/ };
class student5p: public student2c,public student5y { int exam; /*...*/ };
```



Виртуальное наследование

```
class student { protected: char * name; /* ... */ int student_id; };
class student2c: virtual public student { protected: char* pract; char* tutor; };
class student3c: virtual public student { protected: /* ... */ public: /*...*/ };
class student4y: virtual public student { protected: char* certificate; /*...*/ };
class student5y: virtual public student { protected: char* diploma; /*...*/ };
class student4p: virtual public student2c,public student4y { bool test; /*...*/ };
class student5p: virtual public student2c,public student5y { int exam; /*...*/ };
void g () { student4p * p4 = new student4p; student * ps = p4; } // Нет ошибок
```



Виртуальное наследование

- При формировании наследственной иерархии с виртуальным наследованием нужно следить, чтобы функции виртуального базового класса вызывались только из самого “нижнего” производного класса
- Для класса *student* правильными будут операторы:
`student5p n5p (“Иван”, 4.78, 20110217, “Память”, “Пушкин”, “Транслятор”, 5);`
`student * n5 = & n5p; n5 -> print ();`
- Ответ на вызов функции может быть таким:
ФИО = Иван Петрович Белкин
Курс = 2
Средний балл = 4.78
Номер зачётки = 20110217
Тема курсовой = Распределение памяти при трансляции выражений
Преподаватель = Александр Сергеевич Пушкин
Тема диплома = Транслятор Си++
Экзамен сдан = 5

Неоднозначность при множественном наследовании

- Вторая проблема неоднозначности при множественном наследовании связана с возможным наличием одинаковых имён в разных базовых классах:

```
class A { public: int a; void (* b) (); void f (); void g (); /* ... */ };
class B { int a; void b (); void h (char);
public: void f (int); void h (); void h (int); int g; /* ... */ };
```

- Классы *A* и *B* проблемы не создают, но наследующий им класс *C* выявляет неоднозначность:

```
class C: public A, public B { /* ... */ };
void gg (C * pc) { C c; c.a = 1; // ОШИБКА: неясно, A::a или B::a
pc -> a = 1; } // ОШИБКА: неясно, A::a или B::a
```

- Неверны рассуждения: в классе *A* поле *a* – открытое, в классе *B* оно закрыто, значит можно пользоваться открытым полем *A::a*

Неоднозначность при множественном наследовании

- Процесс поиска определяющего вхождения члена класса (поля или метода) начинается в точке использующего вхождения:
 - Шаг 1: контроль однозначности
 - Шаг 2: выбор перегружаемой функции
 - Шаг 3: проверка доступности

Неоднозначность при множественном наследовании

- **Шаг 1**: контроль однозначности
- Выясняется, определено ли анализируемое имя в одном базовом классе или сразу в нескольких
- Контекст использования имени не привлекается (неважно, что это за имя – в одном классе это может быть имя поля данных, а в другом – имя метода класса)
- Совместное использование имени в одном классе (определение одного имени в разных контекстах в одном из базовых классов) допускается

Неоднозначность при множественном наследовании

- **Шаг 2**: выбор перегружаемой функции
- Если однозначно определённое имя есть имя перегруженной функции, делается попытка разрешить анализируемый вызов:
 - Ищется функция, способная обслужить данный конкретный вызов имени
 - Такая функция должна быть единственной
- **Шаг 3**: проведение проверки доступности
- Контроль доступа проводится только, если два первых шага завершились успешно

Неоднозначность при множественном наследовании

- Каждый из последовательно выполняемых шагов может привести к фиксации ошибки:

Однозначность
(с точностью до
совместного
использования)

=>

Единственность
перегруженной
функции

=>

Доступ

Неоднозначность при множественном наследовании

- Вторая проблема неоднозначности при множественном наследовании связана с возможным наличием одинаковых имён в разных базовых классах:

```
class A { public: int a;          void (* b) ();   void f ();      void g (); /* ... */ };
class B {          int a;          void b ();      void h (char);
                public: void f (int); void h ();      void h (int);  int g;      /* ... */ };
class C: public A, public B { /* ... */ };
void gg (C * pc)   {
    pc -> a = 1;    // ОШИБКА: нет однозначности
    pc -> b ();    // ОШИБКА: нет однозначности
    pc -> f ();    pc -> f (1); // ОШИБКА: нет однозначности
    pc -> g ();    pc -> g = 1; // ОШИБКА: контекст не привлекается
    pc -> h ('a'); // ОШИБКА: проверка доступности h(char)
    pc -> h ();    pc -> h (1); } // нет ошибок: однозначно и доступно
```

Неоднозначность при множественном наследовании

- Вторая проблема неоднозначности при множественном наследовании связана с возможным наличием одинаковых имён в разных базовых классах:

```
class A { public: int a;          void (* b) ();   void f ();      void g (); /* ... */ };
class B {          int a;          void b ();      void h (char);
               public: void f (int); void h ();      void h (int);  int g;      /* ... */ };
class C: public A, public B { /* ... */ };
void gg (C * pc)   {
    pc -> A::a = 1;
    pc -> A::b ();
    pc -> A::f ();      pc -> B::f (1);
    pc -> A::g ();      pc -> B::g = 1;
    pc -> h ('a');          // ОШИБКА: проверка доступности h(char)
    pc -> h ();          pc -> h (1); } // нет ошибок: однозначно и доступно
```

Чистые виртуальные функции

- При программировании могут возникать ситуации, когда в базовых классах виртуальные функции не могут выполнять никаких реальных действий, становясь “чистыми” виртуальными функциями, например:

```
class shape    { public:    virtual double area ()    { return 0; } };
class rectangle: public shape { /* ... */ private: double height, width; /*...*/
                public:    double area ()    { return height *width; }          };
class circle:   public shape { /* ... */ private: double radius;          /*...*/
                public:    double area ()    { return PI * radius * radius;}      };
```

- Заданные описания позволяют создать массив указателей на базовый (абстрактный) класс и в одном цикле подсчитать сумму площадей разнородных геометрических фигур:

```
shape * p [N];  int i;  double total_area = 0.0;
for (i = 0; i < N; i ++) total_area += p [i] -> area ();
```

Чистые виртуальные функции

- Реально могут существовать только объекты производных классов, именно на них могут указывать элементы массива указателей
- Если будут введены дополнительные плоские фигуры и описаны их классы и методы вычисления площади, то указатели на них также можно внести в общий массив, и без изменения основной программы вычислить общую площадь
- Если тело виртуальной функции (например, функции *area ()*) из базового класса вообще не используется в программе, функция является “чистой” виртуальной функцией
- Для таких функций используется специальный синтаксис:

```
class shape { /* ... */ public: /* ... */ virtual double area () = 0; };
```

Абстрактные классы

- *Абстрактным* называется класс, содержащий хотя бы одну чистую виртуальную функцию
- Чистая виртуальная функция – это функция вида:
virtual <тип_результата><имя_функции> (<параметры>) = 0;
- Абстрактный класс может использоваться только как база для построения других классов
- Объекты абстрактного класса создавать нельзя, а указатели на них заводить можно
- Абстрактный класс может содержать неконстантные члены-данные, описания конструкторов и деструкторов
- Класс, производный от абстрактного класса, может остаться абстрактным, если в нём конкретизированы не все чистые виртуальные функции базовых классов, поскольку чистые виртуальные функции наследуются и остаются виртуальными

Реализация виртуальных функций

- Для реализации аппарата виртуальных методов используется механизм косвенного вызова через специальные связанные с полиморфным типом объекта массивы указателей на функции-члены, такие массивы называются таблицами виртуальных методов (ТВМ)
- Таблица виртуальных методов создаётся в одном экземпляре для каждого класса, в каждый полиморфный объект компилятор неявно помещает указатель *tvm** *ptvm* на соответствующую таблицу ТВМ, в которой хранятся адреса виртуальных методов (число ссылок на ТВМ соответствует числу созданных объектов)

Реализация виртуальных функций

- Пусть есть иерархия классов: A, B, C

```
class A { int a;  
    public: virtual void f (int);  
            virtual void g (int);  
            virtual void h (int);  
};  
class B: public A { int b;  
    public: void g (int);  
};  
class C: public B { int c;  
    public: void h (int);  
};
```

A_TVM:

```
&A::f  
&A::g  
&A::h
```

B_TVM:

```
&A::f  
&B::g  
&A::h
```

C_TVM:

```
&A::f  
&B::g  
&C::h
```

Таблица виртуальных функций

- Пусть есть иерархия классов: A, B, C
- Память для таблиц виртуальных функций автоматически отводится компилятором в статической области
- Ссылки на одноимённые функции в таблицах виртуальных методов всех классов одной наследственной иерархии всегда находятся на одном и том же месте

A_TVM:

&A::f

&A::g

&A::h

B_TVM:

&A::f

&B::g

&A::h

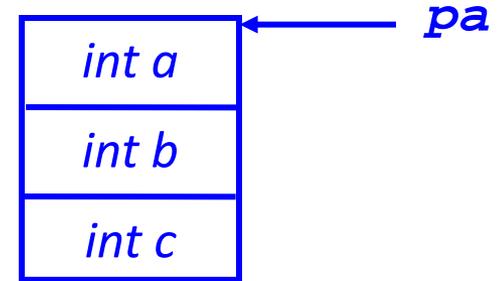
C_TVM:

&A::f

&B::g

&C::h

Реализация виртуальных функций



- Виртуальная функция $g ()$ может быть вызвана так:

$C\ x; \quad A * pa = \& x; \quad pa \rightarrow g ('a');$

- Активный объект относится к типу C , но реализация метода полностью наследуется из класса B
- При работе функции $B::g ()$ указатель **this** имеет доступ ко всем полям класса B и к открытым полям класса A (**this** $\equiv pa$):

$(* (pa \rightarrow c_tvm [index (g)])) (pa, 'a');$

- Издержки по памяти для каждого полиморфного объекта выливаются в неявное хранение дополнительного указателя

Реализация виртуальных функций

- Сложнее реализовать виртуальные функции при множественном наследовании:

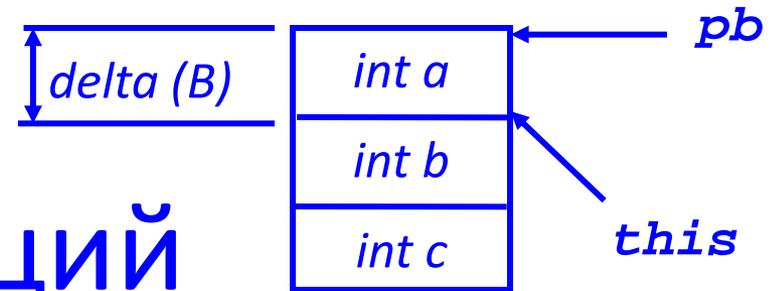
```
class A                { public: virtual void f (); };  
class B                { public: virtual void f ();  
                        virtual void g (); };  
class C: public A, public B { public:          void f (); }
```

```
C x, * pc = & x;
```

```
A * pa = & x;   B * pb = & x;
```

```
pb -> g (1);
```

Реализация виртуальных функций



- Активный объект относится к типу C , но реализация метода полностью наследуется из класса B
- При работе функции $g()$ указатель `this` имеет доступ только к той части активного объекта x типа C , которая унаследована от класса B (но не от A):

`this`

```
(* (pb -> c_tvm [index (g)])) ((B *) ((void *) (pb) + delta (B)), 1);
```

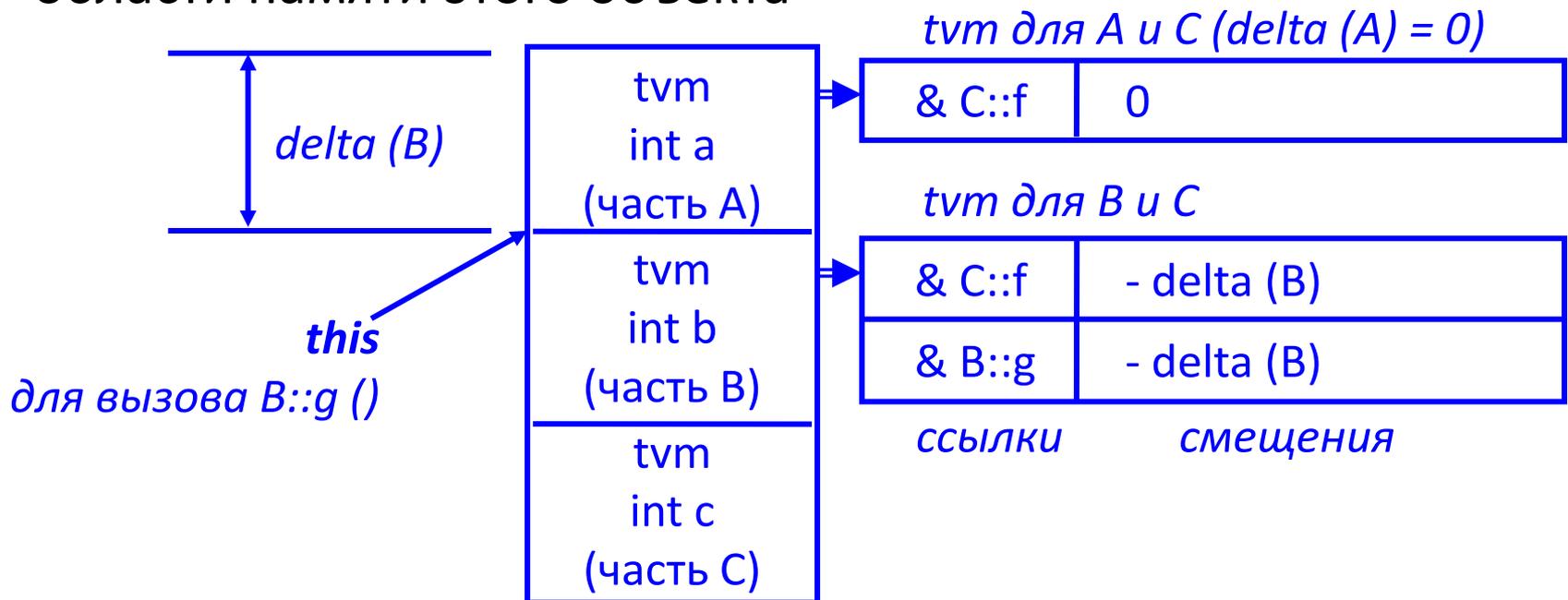
- Смещение `delta (B)` определяет место в объекте типа C , начиная с которого размещаются поля, унаследованные от класса B

Реализация виртуальных функций

- В таблице виртуальных методов класса A размещается одна запись, так как в этом классе имеется только одна виртуальная функция $f()$
- В таблице виртуальных методов класса B размещаются две записи: в этом классе определены сразу две виртуальные функции – $f()$ и $g()$
- В классе C имеются две функции
 - функция $g()$ унаследована от класса B
 - функция $f()$ есть переопределение виртуальных функций $f()$, имеющих в базовых классах A и B

Таблица виртуальных функций

- При множественном наследовании в строке таблицы виртуальных методов находятся сразу два значения: адрес тела функции и смещение начала доступной части данных объекта в области памяти этого объекта



- Строка таблицы имеет структуру:

```
struct tvn_entry { void * (* fct) (); int delta; };
```

Реализация виртуальных функций

- Значения смещений известны во время компиляции, они вычисляются непосредственно компилятором
- Во время компиляции неизвестно, какое именно из заранее рассчитанных смещений надо использовать (указатель *pb* ссылается на часть объекта *C*, унаследованную от класса *B*)
- Во время исполнения вызовов *pa -> f ()* или *pb -> f ()* формируется доступ к нужной функции *f ()*, например:

```
tvm_entry * vt = & pb -> tvn [index (f)];
```

```
(* vt -> fct) ((B *) ((void*) (pb) + vt -> delta));
```

Пространства именования

- Программные элементы (классы, объекты, функции), относящиеся к обработке связанной информации можно объединять в единое пространство имён:

```
namespace Student { class student { ... };  
                    class student1c: public student { ... };  
                    class student2c: public student { ... };  
                    class student3c: public student { ... };  
                    class student4c: public student { ... };  
                    class student5c: public student { ... };  
}
```

```
Student :: student2c :: print () { ... }
```

Пространства именования

- Объявления функций можно объединять в пространствах имён, создавая описания интерфейсов:

```
namespace Trigon
{ double acos  (double x);
  double asin  (double x);
  double atan  (double x);
  double atan2 (double y, double x);
  double cos   (double x);
  double sin   (double x);
  double tan   (double x);
}
```

Пространства именования

- Отдельно даются определения функций, описывающие реализацию функций:

```
double Trigon :: acos (double x) { ... }  
double Trigon :: asin (double x) { ... }  
double Trigon :: atan (double x) { ... }  
double Trigon :: atan2 (double y, double x) { ... }  
double Trigon :: cos (double x) { ... }  
double Trigon :: sin (double x) { ... }  
double Trigon :: tan (double x) { ... }
```

Пространства именования

- Новый член пространства имён нельзя объявить вне определения этого пространства имён:

```
double Trigon :: ctg (double x);    // ОШИБКА!
```

- Такой подход позволяет обнаруживать ошибки на стадии компиляции программы:

```
double Trigon :: cas (double x);    // ОШИБКА! Нет функции
```

```
double Trigon :: sin (float x);     // ОШИБКА! Неверный тип
```

Пространства именовани

- Пространства имён есть область видимости
- Обычные локальные и глобальные области видимости, а также классы являются пространствами имён
- Все имена могут относиться к некоторому пространству имён, единственное исключение – функция *main ()*
- *Пространства имён могут вкладываться друг в друга*

Пространства именования

- Имена из других пространств именования следует дополнительно специфицировать именами этих пространств:

```
double Trigon :: tan (double x)
{ double c = cos (x);
  if (c) return sin (x) / c;
  return Error :: error;
}
```

- Объявление используемого пространства именования:

```
double Trigon :: tan (double x)
{ using Error :: error;
  double c = cos (x);
  if (c) return sin (x) / c;
  return error;
}
```

Пространства именования

- Объявления об использовании некоторых элементов из других пространств именования вводят локальные синонимы для имён этих элементов, их можно сосредоточить непосредственно в определении собственного пространства имён:

```
namespace Trigon
{ double acos (double x); /* ... */
  double tan  (double x);
  using Error::error;
}
```

Пространства именования

- Одно пространство имён можно сделать полностью доступным из другого:

```
namespace Trigon
{ double acos (double x); /* ... */
  double tan  (double x);
  using namespace Error; // таких директив может быть много
}
```

- Директива ***using*** выполняет композицию пространств имён
- Одинаковые имена в разных пространствах именования снабжаются спецификаторами и не приводят к конфликтам

Пространства именования

- В каждой единице трансляции может быть одно неименованное пространство именования, в разных единицах трансляции неименованные пространства считаются разными:

```
namespace { double my_cos (double x); /* ... */ }
```

- Имена, введенные в неименованных пространствах, доступны в объемлющих их областях видимости:

```
namespace Уникальное_имя { double my_cos (double x); ... }  
using namespace Уникальное_имя;
```

- Глобальные объекты могут стать доступными из вложенных пространств именования при помощи спецификатора без имени:

```
:: p ++; // обращение к глобальной переменной p
```

Пространства именования

- Псевдонимы пространств именования:

```
namespace MSU = Moscow_State_University;  
/* ... */  
MSU :: double count_avb (const & student);
```
- Пространства имён открыты: в любом месте программы можно продолжить определение ранее определённого пространства именования:

```
namespace Student { ... }  
#include <list>  
namespace Student { ... }
```

Пространства именования

- Пространства именования не создают границ, препятствующих работе механизма перегрузки функций:

```
namespace A { void f (int);    /* ... */ }
namespace B { void f (char);  /* ... */ }
namespace C {
    using namespace A;
    using namespace B;
    void g ()
    {   f ('a');                // вызовется f (char)
        /* ... */
    }
}
```

Пространства именования

- Пространство *std* используется для группирования функций стандартной библиотеки:

```
printf ("Привет\n");    // ОШИБКА, нет глобальной printf ()  
std::printf ("Привет\n"); // правильно
```

- Средства стандартной библиотеки доступны через стандартные заголовочные файлы, например, функция *printf ()* доступна после выполнения вставки:

```
#include <cstdio>
```

- Для работы с именами библиотечных функций, рекомендуется использовать директиву

```
using namespace std;
```

Ошибки в программных продуктах

- Ошибки, обнаруживаемые компиляторами (или другими компонентами системы программирования)
 - Ошибки в записи лексем (*55Ident*)
 - Нарушение баланса скобок в арифметических выражениях
 - Использование в операторах операндов с неподходящими типами
 - Бесконечная рекурсия
 - Использование одной функции там, где следует вызывать другую
- Ошибки, обнаруживаемые при исполнении программ
 - Ошибки в программах, не обнаруживаемые компонентами систем программирования
 - Ошибки в данных, поступающих в программу
 - Ошибки и сбои в работе аппаратного обеспечения

Типовые реакции на ошибки во время исполнения программы

- Прекращение выполнения программы (например, вызовом системной функции *exit ()*)
- Возврат в вызвавшую функцию значения “ошибка” и/или установка специального признака (например, ненулевого значения глобальной переменной *errno*)
- Возврат в вызвавшую функцию какого-либо допустимого значения и продолжение работы
- Вызов специальной функции для обработки ошибочной ситуации, которая реализует те же три первых вида реакции на ошибку
- Выдача диагностических сообщений, затем – продолжение по одному из ранее упомянутых решений

Исключительные ситуации в языке Си++

- Функция, обнаружившая проблему, генерирует исключительную ситуацию («*исключение*»), чтобы передать решение проблемы той функции, которая её вызвала (непосредственно или опосредованно)
- Функция, которая готова решать данные проблемы, заранее указывает, что будет перехватывать такие исключения

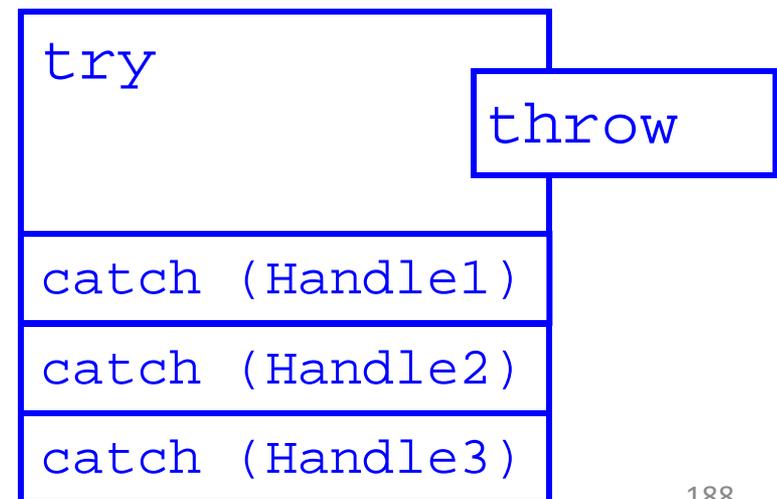
Исключительные ситуации в языке Си++

- Фиксация ошибочной ситуации отделена от обработки
- Этапы управления исключительными ситуациями:
 1. Выделение блока программы, в котором возможно возникновение исключительной ситуации и для которого организован перехват этой ситуации
 2. Фиксация исключительной ситуации и передача управления обработчику
 3. Обработка исключительной ситуации обработчиком

Исключительные ситуации в языке Си++

- try-блок – блок, в котором возможно возникновение исключительной ситуации
- catch-блок – блок обработки исключения
- throw – оператор возбуждения исключения

```
void f ()  
{ /* ... */  
  try  
    { /* ... */ throw Exception ();  
    }  
  catch (Handle) { /* ... */ }  
}
```



Переопределение операции индексации в классе `string`

```
char & string::operator [] (int i) // С остановкой работы
{ if (i < 0 || i >= size) { cerr << "ошибка:" << i << endl; exit (1); }
  return p [i];
}

char & string::operator [] (int i) // С возбуждением исключения
{ if (i < 0)      throw "string: задан отрицательный размер";
  if (i >= size) throw i;      // слишком большой размер
  return p [i];
}

void g (int i) { /*...*/ try { string a (i), b (7); b [i] = a [i - 1]; /*...*/ }
                catch (int n) { /* ... */ }
                catch (const char * diagn) { /* ... */ } }
```

Исключительные ситуации в языке Си++

- Некоторый обработчик *catch (Handle)* будет вызван:
 1. Если тип *Handle* имеет тот же тип, что и *Exception*
 2. Если тип *Handle* является однозначным доступным базовым типом для типа *Exception*
 3. Если *Handle* и *Exception* являются указателями, и условия 1) или 2) выполняются для типов, на которые они ссылаются
 4. Если *Handle* является ссылкой, и условия 1) или 2) выполняются для типа, на который ссылается *Handle*
- Любой указатель совместим со свободным указателем: перехватчик *catch (void*)* пригоден для обработки любого указателя, не перехваченного предыдущими перехватчиками

Действия при возбуждении ИСКЛЮЧЕНИЯ

```
class X { /* ... */ public: X (char, double);    X (const X&);  
                    ~X ();                    /* ... */ };  
void f (double y) { /* ... */ X z ('+', y);    throw z;    /* ... */ }  
main ()           { /* ... */ try { /* ... */ f (1.2);    /* ... */ }  
                  catch (X) {                          /* ... */ }  
                  /* ... */ }  
}
```

- **Шаг 1.** При выходе из функции *f ()* из-за возбуждения исключительной ситуации, создаётся временный объект (копия объекта *z*), с которым работает обработчик
- Копия существует всё время до тех пор, пока исключение не будет обработано (произойдёт выход из ***catch***-блока)

Действия при возбуждении ИСКЛЮЧЕНИЯ

- Шаг 2. Выполняется “свёртка стека”: для объектов *try*-блока это эквивалентно выходу из блока с помощью оператора *goto*
- Конструктор копирования для типа *X* и все необходимые деструкторы должны быть доступны в точке возбуждения исключения
- Внутри обработчика исключения само это исключение можно возбудить повторно: внутри обработчика (либо внутри функции, вызываемой из обработчика) можно вставить оператор *throw* без параметров

Действия при возбуждении ИСКЛЮЧЕНИЯ

- Шаг 3. При возбуждении исключения (выполнении оператора *throw* с параметром) в списке обработчиков объемлющего блока ищется нужный обработчик (*статическая ловушка*)
- Если подходящий перехватчик найден, выполняется его составной оператор (тело перехватчика), затем управление передаётся оператору, расположенному следом за последним перехватчиком той группы перехватчиков, в которую входит сработавший перехватчик

Действия при возбуждении ИСКЛЮЧЕНИЯ

- **Шаг 4.** Если нет подходящего перехватчика, осуществляется выход в объемлющий блок
- Далее выполняется свёртка стека с чередованием *статических и динамических ловушек*

```
int main () { /* ... */  
try { /* ... */  
    f ();  
}  
catch (int) { /* ... */ }  
catch (float) { /* ... */ }  
}
```

```
void f () { /* ... */  
try { /* ... */  
    g ();  
}  
catch (float) { /* ... */ }  
catch (char) { /* ... */ }  
}
```

```
void g () { /* ... */  
try { double d = 1.0;  
    throw d;  
}  
catch (int) { /* ... */ }  
catch (long) { /* ... */ }  
}
```

Действия при возбуждении исключения

- Шаг 5. Если ни один обработчик вплоть до функции *main ()* не перехватил возбуждённое исключение, работа программы прекращается: вызывается функция завершения работы *terminate ()*
- Работа функции *terminate ()* по умолчанию состоит в вызове системной функции остановки *abort ()*, но её можно подменить, обратившись заранее с нужным параметром к функции *set_terminate ()*:

```
typedef void (* pf) ();  
pf set_terminate (pf); // возвращается предыдущая функция
```

Вызов функции `terminate ()`

- Функция `terminate ()` вызывается:
 1. Если в программе нет подходящего обработчика возбуждённого исключения
 2. Если делается попытка повторно возбудить исключение (выполняется оператор `throw` без параметра), а активного исключения нет (то есть перевозбуждение происходит не в перехватчике и не в вызванной из него функции)
 3. Если деструктор, вызванный в процессе свёртки стека, сам пытается завершить свою работу, возбуждая исключительную ситуацию
 4. Если система не в состоянии справиться со свёрткой стека (стек разрушен)

Исключительные ситуации

- В программе можно организовать перехват вообще всех исключительных ситуаций, которые могут в ней возникнуть, использование синтаксической конструкции “эллипс” (*catch (...)*) означает присутствие любого аргумента (*шаг 1* не выполняется, копирования объекта-исключения не происходит):

```
void f () { /*...*/ try { /*...*/ } // Основная работа f ()
           catch (...) // Перехват “всех остальных ситуаций”
           { /*...*/ } // Обработка исключений
```

- С *try*-блоком можно связать несколько блоков обработчиков исключений
- При поиске подходящего обработчика они просматриваются в том текстуальном порядке, в котором записаны в программе
- Перехват производных типов должен предшествовать перехвату базового
- Перехват “всех остальных исключений” с помощью эллиптической конструкции *catch (...)* должен всегда находиться в списке обработчиков последним

Исключительные ситуации

- В обработчике исключения в качестве параметра перехвата можно указывать не только тип, но и имя формального параметра исключения, имеющего перехватываемый тип (это приведёт к повторному копированию объекта-исключения с возможным преобразованием типа от производного к базовому):

```
char lex () { char c; /*...*/ if (c < 21) { /*...*/ throw c;} /*...*/ return c; }  
try { char m = lex (); /* ... */ }  
catch (char c) { cout << "неверный символ " << c << endl; }
```

или

```
catch (char &c) { cout << "ссылка на плохой символ " << c << endl; }
```

- К типу, используемому для перехвата исключения, можно добавить спецификатор **const**, запрещая модификацию параметра в перехватчике
- Указание в заголовке обработчика ссылочного типа блокирует повторное копирование. В этом случае возможно изменение (`c += '0'`) значения исходного объекта-исключения, точнее его первой копии, которое может быть заметно при выполнении оператора **throw** без параметров.

Исключительные ситуации

- Ошибки разных типов можно разделить между исключениями с различными именами, передавая их обработку разным обработчикам:

```
struct Except { int i;    Except (int k) { i = k; }}; /* ... */  
try { /* ... */ throw    Except (67); }  
catch (const Except Err) // Обработка исключительной ситуации  
    { cerr << "Exception (" << Err.i << ")\n"; }
```

- Исключительная ситуация считается обработанной в самый начальный момент входа в обработчик
- Любые исключения, возникшие во время выполнения обработчика, обрабатываются обработчиками объемлющих блоков, циклов обработки не возникает:

```
class Overflow {          /* ... */ };  
void f () {    try {      /* ... */          /* ... */ throw Overflow (); }  
              catch (Overflow) {          /* ... */ throw Overflow (); }  
              // Обработка исключения вне функции f ()  
}
```

Стандартные исключения

- В языке Си++ имеются стандартные исключительные ситуации, генерируемые при выполнении тех или иных операций:

Имя	Генерирующая операция	Заголовочный файл
<code>bad_alloc</code>	<i>new</i>	<code><new></code>
<code>bad_cast</code>	<i>dynamic_cast</i>	<code><typeinfo></code>
<code>bad_typeid</code>	<i>typeid</i>	<code><typeinfo></code>
<code>bad_exception</code>	спецификация исключения	<code><exception></code>
<code>out_of_range</code>	<code>at (); bitset<>::<i>operator</i> []()</code>	<code><stdexcept></code>
<code>invalid_argument</code>	конструктор <code>bitset</code>	<code><stdexcept></code>
<code>overflow_error</code>	<code>bitset<>::to_ulong ()</code>	<code><stdexcept></code>
<code>ins_base::failure</code>	<code>ins_base::clear ()</code>	<code><ios></code>

Динамическое преобразование

- Бинарная операция языка динамического преобразования типа возвращает правильный указатель (или ссылку) в случае правильного предположения о типе объекта:

dynamic_cast <T*> (p) *dynamic_cast* <T&> (r)

- Операция имеет смысл только при работе с виртуальными функциями и указателями на объекты:

```
class student { /* ... */ }; // полиморфный класс
class student2c : public student { /* ... */ }; // производный класс
int main ()
{
    student s, * ps;   student2c ds, * pds;
    pds = & ds; // derived * = derived *
    ps  = pds; // base   * = derived *
    pds = ps;  // опасно, но возможно после предыдущего присваивания
}
```

Динамическое преобразование

- Обычно динамическое преобразование типа указателя проводится непосредственно в условном операторе:

```
void f (student * ps) { // ps указывает на некоторый класс
    student2c * pds;
    if ((pds = dynamic_cast<student2c*> (ps)) != 0)
        { pds -> print ();
        }
    else
        { // реакция на получение объекта другого типа
        }
}
```

Динамическое преобразование

- Результат операции динамического приведения типа указателя ***dynamic_cast*** $\langle T^* \rangle (p)$ эквивалентен приведению типа указателя p к типу указателя T^*
- Динамическое приведение типа не допускает нарушений правил доступа к закрытым и защищённым базовым классам
- Операция возвращает *нулевое значение*, если обнаруживается:
 1. Передача в качестве исходного значения преобразования нулевого операнда ($p == 0$)
 2. Неоднозначность при поиске базового класса типа T

Динамическое преобразование

- Параметр операции должен быть ссылкой или указателем на полиморфный тип
- Результирующий тип не обязан быть полиморфным (исходный объект при этом всё равно должен быть полиморфным):

```
student * ps; student2c * pd; int * pi;  
void * pv1 = dynamic_cast<void *> (ps); // правильно  
void * pv2 = dynamic_cast<void *> (pd); // правильно  
void * pv3 = dynamic_cast<void *> (pi); // ОШИБКА
```

Статическое преобразование

- Статическое приведение типа не анализирует объект, который оно приводит
- Статическое приведение типа формирует новый указатель нужного типа:

```
student2c * g (void * p) {  
    student * ps = static_cast<student*> (p);  
    // ps указывает на неизвестный класс  
    return dynamic_cast<student2c*> (ps);  
    // результат не может быть предсказан  
}
```

Статическое преобразование

- Ограничения, наложенные на возможности статического приведения типов, позволяют проводить такое преобразование только для:

(1) родственных типов из одной иерархии классов (проверка родства классов отсутствует лишь при преобразовании указателя из типа *void**)

(2) арифметических типов (даже не очень близких), например,

float => *int*, *int* => *enum*

Преобразование типов ссылок

- Динамическое приведение ссылок имеет особенности, отличающие его от приведения указателей
- Приведение указателя к новому типу может привести к получению нулевого значения, означающего, что полученный указатель не указывает ни на какой объект
- Динамическое приведение указателя p к новому типу T `dynamic_cast<T*> (p)` отвечает на вопрос: “*Может ли указатель p указывать на объекты типа T?*”
- Ссылка всегда связана с некоторым объектом, нулевых ссылок в программах быть не может
- Для ссылки r операция динамического приведения типа `dynamic_cast<T&> (r)` является утверждением: “*Ссылка r может быть связана с объектом типа T!*”

Фундаментальные отличия между указателями и ссылками

- Если операция динамического приведения ссылки не принадлежит ожидаемому типу, возбуждается стандартная исключительная ситуация *bad_cast*:

```
void f (student * p, student & r)
{ student2c *pp;    student2c & pr;
  if (pp = dynamic_cast<student2c*> (p))
    { /* использование указателя pp */ }
  else { /* указатель pp не имеет нужного типа */ }
  pr = dynamic_cast<student2c&> (r);
    /* использование ссылки pr */
}
```

Фундаментальные отличия между указателями и ссылками

- Для защиты от неудачных приведений к ссылке необходимо организовывать перехват стандартного исключения:

```
void g ()
{ try { f (new student2c, * new student2c);
        // нормальный вызов с нужным типом
        f (new student5c, * new student5c);
        // неверный тип, выход в перехватчик
    }
    catch (bad_cast) { /* обработка исключения */ }
}
```

Операция определения типа

- Унарная операция *typeid*, имеющая смысл для указателей и ссылок на объекты полиморфных типов, позволяет точно определить тип исследуемого объекта
- Если операнд нужного типа имеет значение 0, операция возбуждает исключительную ситуацию *bad_typeid*
- Результатом операции является ссылка на класс стандартной библиотеки по имени *type_info*
- Библиотечными средствами гарантируется, что в этом классе обязательно определены операции сравнения на равенство и на неравенство, а также метод, выдающий указатель на строку, содержащую символьное представление имени типа объекта (зависящее от реализации)

Операция определения типа

- Предполагается, что операция *typeid* должна использоваться примерно в таком контексте:

```
class type_info // это определение есть часть библиотеки!  
{ /* ... */ public: bool operator==(const type_info&) const;  
    public: bool operator!=(const type_info&) const;  
    const char * name () const; // выдаёт имя типа  
};
```

```
if (typeid (* pb) == typeid (D))  
{ /* ... */  
    pd = dynamic_cast<D*> (pb); // на 0 можно не проверять  
    cout << typeid (* pb).name () << ", " << typeid (* pd).name ();  
    // Будет напечатано, например:  
    // class student, class student2c  
} // Показанные преобразования лучше, чем pd = (D*) pb
```

Виды преобразования типов

- Кроме динамического и статического преобразований типов, имеются ещё два вида преобразования:

- небезопасная операция *const_cast* отменяет константность или произвольную изменяемость объекта:

```
const int * q = /* ... */; int * p = const_cast<int *> (q);
```

- ещё более опасная операция *reinterpret_cast* присваивает указателю значение, относящееся к другой иерархии наследования, то есть может преобразовывать несвязанные указатели (или числа в указатели):

```
int t = 0xf0;
```

```
p = (int *) t; // так пишут на языке Си
```

```
p = reinterpret_cast<int *> (t); // так пишут на языке Си++
```

Виды преобразования типов

- Все преобразования типов, унаследованные от языка Си, могут в Си++ выражаться с помощью некоторых комбинаций операций *static_cast*, *const_cast* и *reinterpret_cast*
- Этот стиль преобразований более точно указывает различия между исходным и результирующим типами
- В преобразовании вида *(T) expr* невозможно предсказать, какое именно преобразование выполняется, поэтому от него следует отказываться, выбирая одно из преобразований, характерных для Си++
- Наиболее безопасным является преобразование полиморфных типов *dynamic_cast* с динамической проверкой во время выполнения программы

Обобщённое программирование

- Концепция обобщённого программирования подразумевает использование типов в качестве параметров определений классов и функций
- Механизм шаблонов позволяет описывать произвольные обобщения множества алгоритмов
- Шаблон класса определяет данные и операции потенциально неограниченного множества родственных классов, а шаблон функции определяет неограниченное множество родственных функций
- Особенно полезно использование шаблонов при проектировании и реализации стандартной библиотеки

Обобщённое программирование

- **Инстанцирование** – это процесс формирования описания класса или функции по шаблону и его фактическим параметрам
- Получаемое для конкретного значения фактических параметров шаблона описание типа называется **специализацией шаблона**
- Сформированные классы или функции становятся самыми обычными классами и функциями, которые подчиняются всем правилам языка для классов и функций

Обобщённое программирование

- Шаблон может иметь несколько параметров (не менее одного),
- Стандарт языка Си++ допускает параметры шаблонов:
 - типовые (*typename*, *class*)
 - параметры, которые сами являются шаблонами
 - интегральные типы: знаковые и беззнаковые целые типы, *bool*, *char*, *wchar_t*
 - перечислимые типы (не относятся к интегральным, но их значения приводятся к интегральным в результате целочисленного расширения)
 - указатели на объекты, функции или на члены классов
 - ссылки на объекты или функции

Обобщённое программирование

- Параметр шаблона не может иметь тип *void*, не может быть объектом пользовательского типа или иметь плавающий тип:

```
template <double* d> class X; // ОК
```

```
template <double& d> class X; // ОК
```

```
template <double d> class X; // ОШИБКА
```

- Фактические параметры, соответствующие формальным параметрам интегральных и перечислимых типов, должны быть константами
- В теле шаблона все параметры, не являющиеся типами рассматриваются как константы
- Параметры шаблонов могут использоваться также и для определения следующих параметров этих же шаблонов:

```
template <class T, T par2_name> class C { /* ... */ }
```

Шаблоны функций

- Шаблон функции объявляется следующим образом:

```
template <список_формальных_параметров_шаблона>  
        возвращаемый_тип  
        имя_функции (список_формальных_параметров_функции)  
        { /* ... */ }
```

- Примером может служить шаблон функции вычисления степени целого числа, имеющий в качестве параметра значение целого типа:

```
template<int n> inline int power (int m)  
{ int k = 1;  
  for (int i = 0; i < n; ++ i) k *= m;  
  return k;  
}
```

Шаблоны функций

- При обращении к функции-шаблону после имени функции в угловых скобках указываются фактические параметры шаблона – имена реальных типов или значения объектов, и лишь затем обычные фактические параметры функции:

имя_функции<список_фактических_параметров_шаблона>
(список_фактических_параметров_функции)

- Пример обращения к функции-шаблону с параметром:

```
int main() { int m = 4; cout << power<3> (m) << endl; }
```

Шаблоны функций

- Имея обычную функцию, можно определить для неё шаблон, который позволит использовать данные различных типов:

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

- При вызове функции автоматически определяется, какая версия шаблона будет использована, то есть фактические параметры шаблона выводятся из фактических параметров функции
- При вызове `void f () { /*...*/ max (1, 2); /*...*/ }` // *T ≡ int*
формируется вариант функции `int max (int, int)`
- При вызове `void f () { /*...*/ max ('a', 'a'); /*...*/ }` // *T ≡ char*
формируется второй вариант функции `char max (char, char)`

Шаблоны функций

- Параметры могут иметь разные типы:

```
void f () { /* ... */ max ('a', 100); /* ... */ }  
void f () { /* ... */ max (2.5, 1); /* ... */ }
```

1. Перед одним из параметров можно употреблять операцию приведения типа, которая сделает возможным использование нужного шаблона и формирование нужного варианта функции, то есть *double max (double, double)*:

```
void f () { ... max (2.5, (double) 1); ... }
```

Шаблоны функций

- Параметры могут иметь разные типы:

```
void f () { /* ... */ max ('a', 100); /* ... */ }  
void f () { /* ... */ max (2.5, 1); /* ... */ }
```

2. Можно написать новый шаблон, который умеет сравнивать данные различных типов
3. Можно явно указывать выбираемый шаблон с помощью спецификации типом:

```
void f () { ... max<int> ('a', 100);  
           max<double> (2.5, 1);  
           max<long> (4455, 777777);  
           ... }
```

Шаблоны функций

- Параметры могут иметь разные типы:

```
void f () { /* ... */ max ('a', 100); /* ... */ }  
void f () { /* ... */ max (2.5, 1); /* ... */ }
```

4. Можно перегрузить шаблон функцией, то есть к определению шаблона добавить определение одноимённой функции:

```
int max (char x, int y) { return x > y ? (int) x : y; } // или так:  
int max (int x, int y) { return x >= y ? x : y; }
```

Шаблоны функций

- При конфликтах шаблонов и функций работает модифицированный алгоритм определения перегруженной функции, отличающийся тем, что в нём дополнительно применяется шаг **a'**, который рассматривается после того, как не сработал шаг по пункту **a** алгоритма
- Если будет обнаружена функция с формальными параметрами, в точности соответствующими фактическим параметрам вызова (без каких-либо дополнительных преобразований фактических параметров), она и будет вызвана
- Только в противном случае будут искаться пригодные для вызова шаблоны функций, так как явно написанная с точными параметрами функция всегда имеет предпочтение перед шаблонами функций

Шаблоны функций

- Исследование возможности генерации функции по шаблону может привести к генерации новой функции или к использованию уже сгенерированной по шаблону функции:

```
void f () { /* ... */ max ('a', 'a'); /* ... */ }  
==>      max<char> ('a', 'a');
```

- Следующие шаги модифицированного алгоритма будут обычными: расширения целочисленных и вещественных параметров, другие стандартные, а затем пользовательские преобразования параметров, именно в соответствии с этими шагами алгоритма будет обрабатываться вызов

```
void f () { /* ... */ max ('a', 100); /* ... */ }
```

Перегрузка шаблонов функций

- При выборе подходящей функции осуществляется поиск специализации, наилучшим образом соответствующей списку фактических параметров вызова, фактически выбор функции проводится по её собственным уже специализированным параметрам, что позволяет вводить несколько шаблонов функций с одним и тем же именем:

```
template<class T>          T  max (T* p, int size)  { /* ... */ }
template<class T>          T& max (T&X, T&Y)       {return ( X < Y ? Y : X);}
template<class T,class Pr> T& max (T&X, T&Y, Pr P) {return (P (X, Y)?Y : X);}
```

- Шаблонной функцией можно пытаться находить максимум в классе комплексных чисел `complex`:

```
complex a (1, 2), b (3, 4); void f () { /* ...*/ max (a, b); /* ...*/ }// T≡complex
```

Шаблоны функций

- Механизм шаблонов обеспечивает проверку отсутствия ошибок не только при определении, но и при проведении их инстанцирования
- Если в шаблоне предполагается использование некоторых операций, определённых для одного типа данных, но не определённых для другого, то выявить возможные ошибки можно, при проверке конкретных специализаций шаблона
- Для комплексных чисел можно создать, например, такую операцию сравнения на “больше”:

```
#include <math.h>
```

```
bool complex::operator > (const complex & a) const
```

```
{ return re * re + im * im > a.re * a.re + a.im * a.im; }
```

Шаблоны классов

- Шаблоны создаются не только для функций, но и для классов (структур), имеющих общую логику работы:

`template`

`<список_формальных_параметров_шаблона_типа>`

`class имя_класса { /* ... */ };`

- Шаблоны классов могут иметь не только типовые параметры, но и обычные параметры-переменные разрешённых типов (интегральные, перечислимые, указатели и ссылки), использующиеся в дальнейшем в теле шаблона:

`имя_класса<список_фактических_параметров>объект;`

Шаблоны классов

- Класс векторов лучше оформлять шаблоном класса:

```
template <class T> class Vector { T * p; int size; /* ... */  
    public:  explicit Vector (int);  
            T & operator [] (int); // индексация  
};
```

```
Vector<int> X (20);           // Для массивов нужен
```

```
Vector<complex> y (10); // конструктор умолчания!
```

- Любой метод шаблонного класса есть шаблонная функция:

```
template <class T> T & Vector <T> :: operator [] (int) { ... }
```

Шаблоны классов

- Инстанцирования шаблона и определения объектов:

```
template<class T, int size> class buffer { /* ... */ };
```

```
buffer <char, 1024> X;
```

```
buffer <char, 512*2> Y;
```

```
buffer <int, 1024> Z;    // Объекты X и Z имеют разные типы!
```

- Типы X и Y считаются одинаковыми
- Значения перечислимых типов при инстанцировании шаблона приводятся к интегральным типам
- Значения неинтегральных типов не могут быть параметрами шаблонов

Шаблоны классов

- Шаблон может в качестве своего параметра иметь другой шаблон (это может быть только шаблон класса):

```
template<class T, template<class> class C> class Cont  
    { C<T> mem;      C<T*> ref;      /* ... */ };
```

- Такое определение класса *C*, заданного как класс-шаблон с одним типовым параметром, даёт возможность создавать специализации этого типа во время инстанцирования типа *Cont*:

```
Cont<Entry, vector> vect1; // ссылки в виде вектора  
Cont<Record, list> list2; // ссылки в виде списка
```

Шаблоны классов

- Имена шаблонов классов (как и классов вообще) нельзя перегружать
- Если в программе имеется определение шаблона, попытка включить в ту же область видимости этой программы любое из последующих определений класса с тем же именем будет ошибкой:

```
template<class T> class Tmax { T * p; int size; public: Tmax (); };  
template<class T> class Tmax          { ... }; // ОШИБКА  
template<class T> struct Tmax         { ... }; // ОШИБКА  
template<class U> class Tmax          { ... }; // ОШИБКА  
template<class T, class U> class Tmax { ... }; // ОШИБКА  
                                class Tmax { ... }; // ОШИБКА
```

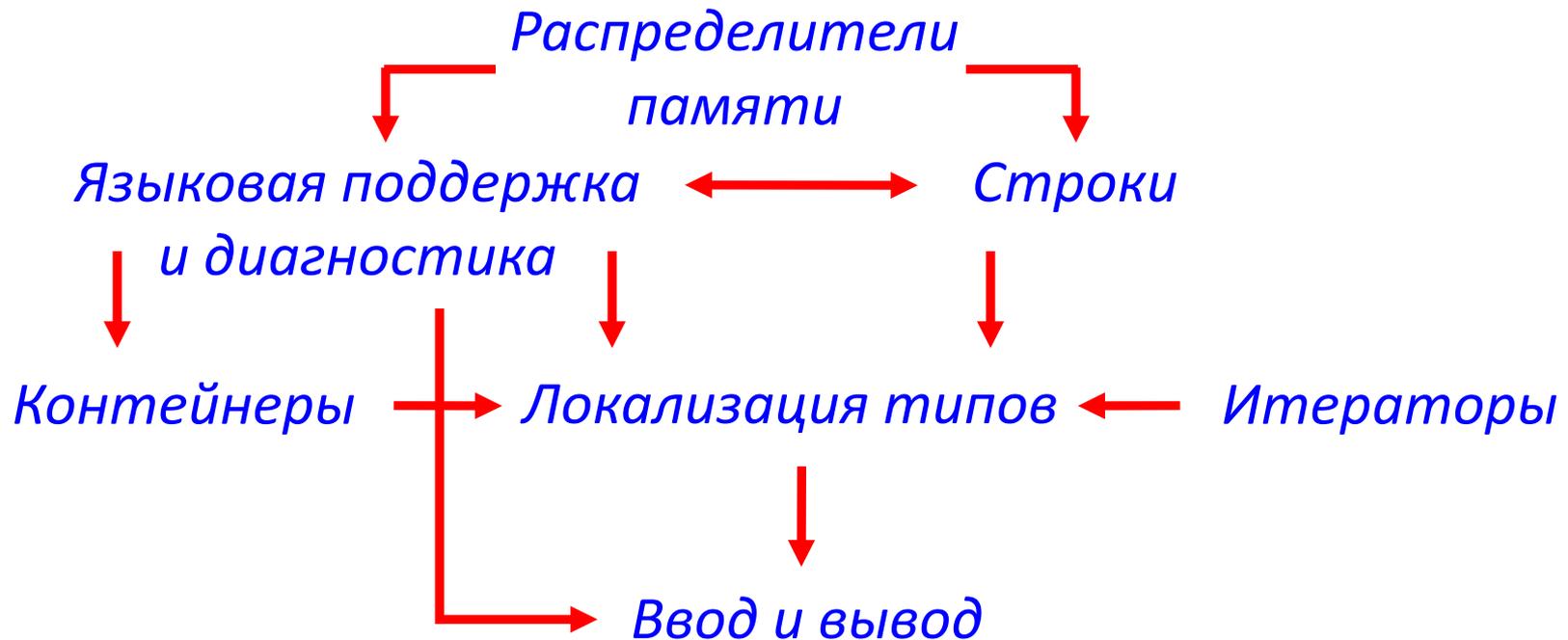
Стандартная библиотека

- Адекватная поддержка объектно-ориентированного программирования, проектирование библиотечных элементов в виде иерархий классов, использование абстрактных классов и виртуальных функций
- Активное использование шаблонов для повышения уровня абстракции базовых алгоритмов
- Использование механизма исключительных ситуаций и иерархии стандартных классов для передачи значений при возбуждении исключительных ситуаций
- Технология “обобщённого программирования”: максимальное обобщение структур и алгоритмов при одновременном сохранении их эффективности

Стандартная библиотека

- Стандартная библиотека – средство разработки контекстно-ориентированных библиотек
- Стандартная библиотека предоставляет совокупность понятий, в терминах которых можно проектировать программы, и набор типовых решений с использованием этих понятий
- Стандартная библиотека – это объединение нескольких компонентов, каждый из которых предоставляет набор примитивов и типовых решений, позволяющих строить наборы конкретных классов и функций для некоторой прикладной области

Взаимодействие элементов стандартной библиотеки



Средства стандартной библиотеки

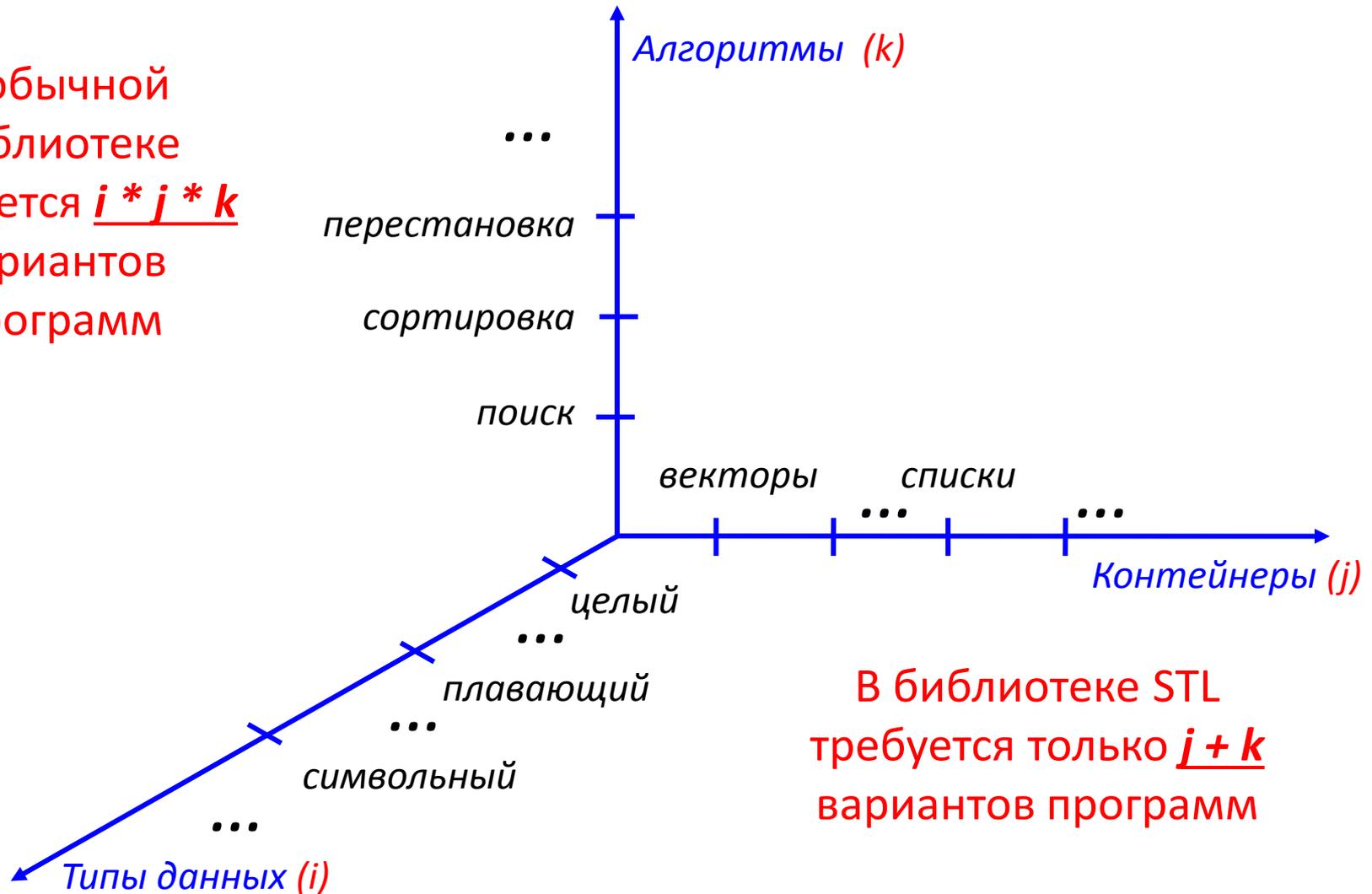
- Средства стандартной библиотеки определены в едином пространстве имён *std* с помощью заголовочных файлов
- Стандартная библиотека предоставляет:
 - Расширяемый набор классов и определений, необходимых для поддержки понятий языка Си++ (управление памятью, информация о типах во время выполнения программы, обработка исключений, средства запуска программы, зависящие от реализации аспекты языка)
 - Поддержки диагностики пользовательских приложений
 - Утилиты общего назначения
 - Контейнеры и итераторы
 - Обобщённые структуры данных и алгоритмы
 - Средства локализации программ
 - Классы и функции для математических вычислений
 - Средства работы со строками
 - Ввод/вывод

Принципы разработки STL

- Библиотека STL представляет собой набор совместно работающих, хорошо структурированных компонентов Си++
 - Все используемые в шаблонах алгоритмы работают не только с библиотечными структурами данных, но также и со встроенными структурами данных языка, в частности, все алгоритмы работают с обычными указателями
 - Использован *принцип ортогональности*: имеется возможность использовать библиотечные структуры данных с собственными алгоритмами, а библиотечные алгоритмы с собственными структурами данных
- *Эффективность реализации*: каждый шаблонный компонент библиотеки имеет обобщённую реализацию, не отличающуюся по эффективности от “ручной” реализации более, чем на несколько процентов
- *Теоретическая обоснованность*

Программные компоненты STL

В обычной библиотеке требуется $i * j * k$ вариантов программ



В библиотеке STL требуется только $j + k$ вариантов программ

Базовые компоненты STL

- Контейнеры, управляющие элементами данных
- Итераторы, обеспечивающие доступ к элементам данных
- Распределители памяти (аллокаторы), инкапсулирующие информацию о моделях памяти
- Алгоритмы, определяющие вычислительные процедуры, отделённые от конкретных реализаций структур данных и не связанные с конкретными контейнерами
- Функциональные объекты
- Адаптеры, обеспечивающие преобразования интерфейсов

Контейнеры STL

- Контейнеры управляют наборами ячеек памяти
- Контейнеры – это объекты (классы), содержащие другие объекты (списки, векторы, очереди, деревья, простые и ассоциативные массивы и т. д.)
- Контейнеры не являются производными от некоторого общего базового класса, каждый контейнер реализует все стандартные контейнерные интерфейсы
- Некоторые контейнеры обладают свойством реверсивности, в дополнение к обычным требованиям к ним предъявляются требования обеспечения доступа с помощью обратных итераторов
- Для любого контейнера определены операции, позволяющие вводить указатели и итераторы на объекты, содержащиеся внутри него, определять расстояние между элементами контейнера и размеры этих элементов
- Среди элементов контейнера выделяются начальный и конечный элементы
- Элементы контейнера можно сравнивать между собой и менять местами

Контейнеры STL

- Выделяются две категории контейнеров: **контейнеры-последовательности** и **ассоциативные контейнеры**
- Последовательности организованы в виде конечных множеств элементов одного типа, эти множества имеют строгую линейную упорядоченность
- Ассоциативные контейнеры позволяют обеспечить быстрый доступ к элементам с помощью ключей
- Два ключа считаются равными, если ни один из них не меньше и не больше другого: неправильно считать, что для ключей можно использовать операцию сравнения '==' и что для определяют равенство ключей так:

`k1 == k2` // **ОШИБКА**

- Правильным подходом к выявлению равенства ключей является последовательное использование пользовательского критерия сравнения:

`comp (k1, k2) == false && comp (k2, k1) == false`

где `comp ()` есть сравнивающий функциональный объект, который передаётся шаблону контейнера в качестве фактического параметра

Контейнеры STL

- Для разных контейнеров определяются семантически сходные и одинаково поименованные функции:
 - для контейнеров-последовательностей определены функции *insert ()* (вставка элементов) и *erase ()* (удаление элементов)
 - для контейнеров-последовательностей имеются функции *push_back ()*, создающая один дополнительный элемент в конце контейнера, и *pop_back ()*, уничтожающая элемент в конце контейнера
 - для ассоциативных контейнеров определены операции *insert ()* и *erase ()* (вставка и уничтожение элементов по заданному ключу), а также поиска элементов и диапазонов по заданному ключу (*find ()*)
- Вставка элементов в ассоциативный контейнер (и исключение элемента из него), в отличие от похожих операций над последовательностями, не разрушает ранее построенные итераторы, указывающие на какие-либо другие элементы этих же контейнеров

Контейнеры STL

- К контейнерам-последовательностям относятся вектор или динамический массив (*vector*), линейный список (*list*), двусторонняя очередь (*deque*)
- К ассоциативным контейнерам относятся ассоциативный массив (*map*), множественный ассоциативный массив (*multimap*), множество (*set* – массив с ключами и без элементов) и множество с одинаковыми ключами (*multiset*)
- Квазиконтейнеры: встроенный массив (*array*), строка (*string*), массив значений (*valarray*), битовый набор (*bitset*)
- Квазиконтейнеры содержат внутри себя элементы, как обычные контейнеры, но обладают некоторыми ограничениями: для них не определяются отдельные операции, которые нельзя реализовать с требуемой для библиотечных элементов эффективностью
- На основе стандартных контейнеров-последовательностей с помощью адаптеров строятся производные контейнеры очередь (*queue*), стек (*stack*), очередь с приоритетами (*priority_queue*)

Контейнеры STL

- Каждый контейнер в своей открытой части содержит серию определений типов, где введены стандартные имена типов:
 - *value_type* – тип элемента
 - *allocator_type* – тип распределителя памяти
 - *size_type* – тип, используемый для индексации
 - *iterator*, *const_iterator* – тип итератора
 - *reverse_iterator*, *const_reverse_iterator* – тип обратного итератора
 - *pointer*, *const_pointer* – тип указателя на элемент
 - *reference*, *const_reference* – тип ссылки на элемент
- Эти имена определяются внутри каждого контейнера так, как это необходимо в каждом конкретном случае, что позволяет писать программы с использованием контейнеров, не зная о настоящих типах, в частности, можно составить программу, которая будет работать с любым контейнером

Итераторы STL

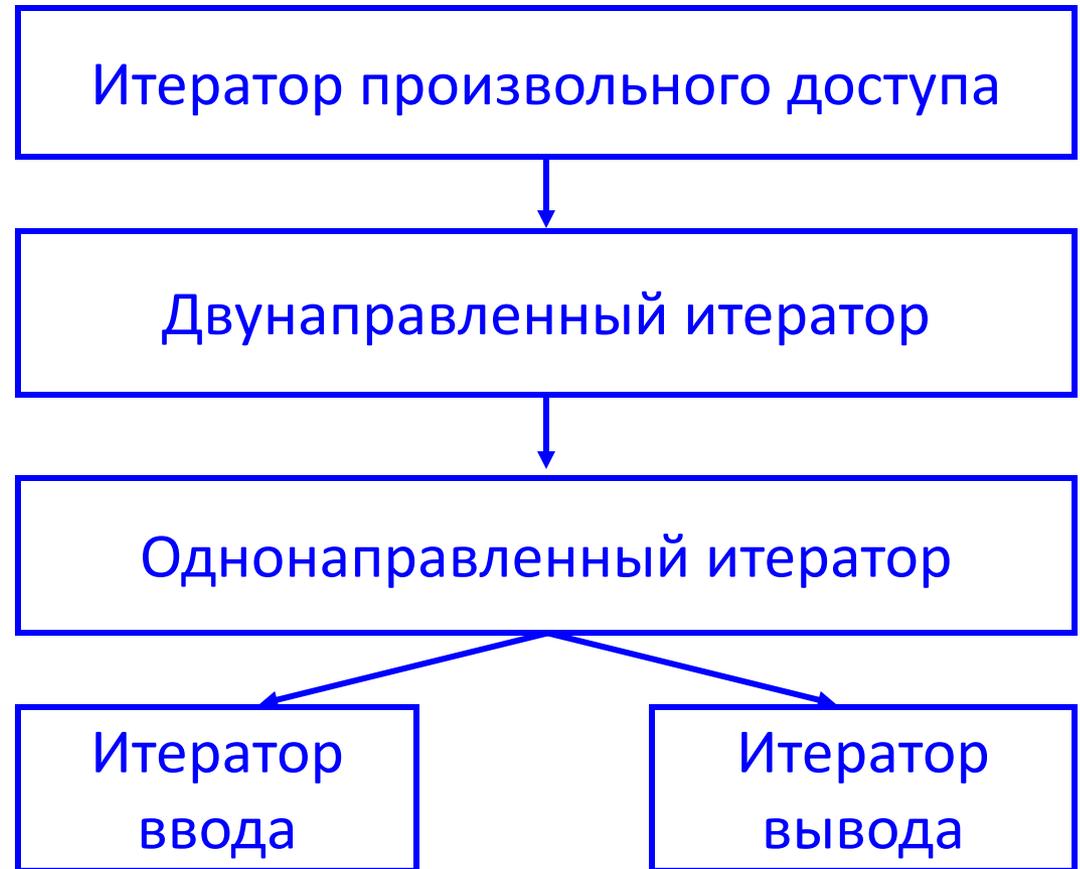
- *Итераторы* (обобщение указателей) предоставляют алгоритмам средства для перемещения по контейнерам и доступа к данным контейнеров
- *Итераторы* поддерживают абстрактную модель данных как последовательности объектов
- Понятия “*нулевой, никуда не указывающий итератор*” не существует, при организации циклов происходит сравнение с концом последовательности
- Каждый контейнер обеспечивает свои итераторы, также поддерживающие стандартный набор итерационных операций со стандартными именами и смыслом
- Описания классов итераторов находятся в заголовочном файле `<iterator>`
- *Итераторы* есть объекты, для которых определена унарная операция *operator**, возвращающая значение некоторого класса или встроенного типа, называемого типом значения (*value type*) итератора
- Всякая шаблонная функция, работающая с итераторами, одновременно может работать и с обычными указателями

Итераторы STL

- Имеется пять видов итераторов:
 1. Итераторы ввода (*InputIterator*)
 2. Итераторы вывода (*OutputIterator*)
 3. Однонаправленные итераторы (*ForwardIterator*)
 4. Двухнаправленные итераторы (*BidirectionalIterator*)
 5. Итераторы произвольного доступа (*RandomAccessIterator*)
- Различные итераторы иерархически вложены друг в друга:
 - *Однонаправленные итераторы объемлют итераторы ввода и вывода*
 - *Двухнаправленные итераторы объемлют однонаправленные итераторы*
 - *Итераторы произвольного доступа объемлют двухнаправленные итераторы*

Итераторы STL

- Итераторы различных видов иерархически вложены друг в друга
- Иерархически объемлющие итераторы удовлетворяют всем требованиям вложенных в них итераторов и могут использоваться вместо них



Операции в иерархии итераторов

Итераторы	Чтение	Доступ	Запись	Изменение	Сравнение
Вывода			$*p=e$	$p++ \ ++p$	
Ввода	$x=*p$	$p->f$		$p++ \ ++p$	$p==q \ p!=q$
Однонаправленные	$x=*p$	$p->f$	$*p=e$	$p++ \ ++p$	$p==q \ p!=q$
Двунаправленные	$x=*p$	$p->f$	$*p=e$	$p++ \ ++p$ $p-- \ --p$	$p==q \ p!=q$
Произвольный доступ	$x=*p$	$p->f$ $p[n]$	$*p=e$	$p++ \ ++p$ $p-- \ --p$ $p+n \ n+p$ $p-n \ p-q$ $p+=n \ p-=n$	$p==q \ p!=q$ $p<q \ p>q$ $p>=q \ p<=q$

Итераторы STL

- При описании алгоритмов, входящих в STL, принято соглашение об использовании стандартных имён формальных параметров, в зависимости от названия итератора в профиле алгоритма, должен использоваться итератор уровня “не ниже, чем”
- По именам параметров шаблона можно понять, какого рода итератор нужен, к какому контейнеру применим этот алгоритм:

```
template <class InputIterator, class T>  
    size_type count (InputIterator start,  
                    InputIterator finish, const T& value);
```

Итераторы STL

- Для продвижения итераторов на заданное расстояние (прибавление целого числа к итератору) и определения расстояния между элементами контейнеров (вычисление разности итераторов) в библиотеке имеются шаблоны функций: *advance ()* и *distance ()*
- Операция *advance (i, n)* может иметь отрицательное значение параметра *n* только для двунаправленных итераторов и итераторов произвольного доступа, эта функция продвигает итератор *i* на *n* позиций вперёд или назад

```
template <class InputIterator, class Distance>
```

```
    inline void advance (InputIterator & i, Distance n);
```

```
InputIterator p;    int n; /* ... */ advance (p, n); // эквивалентно: p += n;
```

- Операция *distance (first, last, n)* прибавляет к *n* число, равное количеству продвижений первого итератора (*first*) ко второму (*last*):

```
template <class InputIterator, class Distance>
```

```
    inline void distance (InputIterator first, InputIterator last, Distance& n);
```

```
InputIterator p, q; int n; /* ... */ distance (p, q, n); // эквивалентно: n += q - p;
```

Итераторы вывода STL

- Присваивание, проводимое с помощью одного и того же значения итератора вывода, можно сделать для данного объекта только один раз, причём пропускать присваивание и переходить сразу к следующему значению итератора нельзя:

```
i, i++; i++; // ОШИБКА
```

- Итератор вывода в каждый момент времени может иметь только одну активную копию:

```
i = j; * ++ i = a; *j = b; // ОШИБКА
```

- Единственным правильным использованием операции разыменования с итераторами вывода является использование этой операции в левой части операции присваивания

```
while (first != last) * result ++ = * first ++; // нет ошибки
```

Итераторы STL

- Итераторы могут указывать на элемент, который гипотетически расположен за последним элементом контейнера, доступ к этому гипотетическому элементу никогда не осуществляется
- Итераторы могут иметь значение, не указывающее ни на какой элемент контейнера, для таких значений результаты большинства операций не определены
- Диапазон есть пара итераторов, задающая начало и конец вычислений
- Диапазон $[i, j)$ относится к элементам структуры данных, начинающейся с элемента, на который указывает i , и кончающейся (но не включающей) элементом j , диапазон $[i, i)$ есть пустой диапазон
- Все алгоритмы должны применяться только к правильным диапазонам, в которых второй итератор достижим из первого за конечное число выполнения операций ***operator ++***

Итераторы STL

- *Суть использования итераторов вместо обычных указателей состоит в том, что итераторы обладают гораздо большей общностью*
- Свойства итераторов описаны значительно более точно, чем свойства простых указателей
- Те свойства итераторов, которые зависят от конкретных реализаций, скрыты в реализации библиотеки, что повышает переносимость программ, написанных с использованием итераторов, но не снижает эффективности этих программ

Итераторы STL

- Каждый контейнер содержит ряд ключевых методов, позволяющих найти концы последовательности элементов в виде значений итераторов:

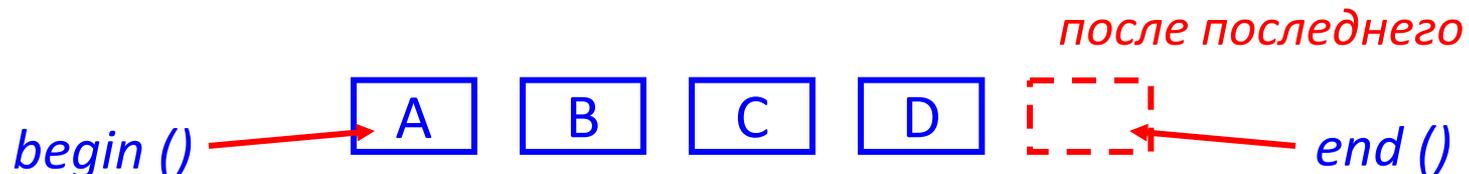
`iterator begin(), const_iterator begin() const`

*// возвращают итератор, который указывает
// на первый элемент последовательности*

`iterator end(), const_iterator end() const`

*// возвращают итератор, который указывает на элемент,
// следующий за последним элементом последовательности*

- Константные итераторы позволяют контролировать модификацию элементов контейнеров и запрещать её там, где это нужно
- С помощью итераторов последовательный доступ к элементам данных контейнерных типов осуществляется от первого элемента к последнему:



Итераторы STL

- Запись `iterator p = v.begin ()` верна независимо от того, к какому контейнеру `v` она применяется, после такого определения `*p` есть первый элемент контейнера `v`
- Шаблонная функция `find ()` ищет итератор заданного элемента в заданном диапазоне итераторов и выдаёт в результате значение итератора, которым поиск заканчивается
- Для достижения целей функции `find ()` достаточно использовать итератор ввода из контейнера, поскольку изменения значений и повторного чтения элементов не потребуется:

```
template <class InputIterator, class T>
```

```
    InputIterator find (InputIterator first, InputIterator last, const T& value)  
        { while ( first != last && * first != value ) ++ first;  
          return first;
```

```
} // Ответ: если first != last, то искомый элемент доступен, как * first
```

Распределители памяти STL

- Распределители памяти (аллокаторы) управляют переносимыми средствами упрятывания информации о моделях памяти, типах указателей, типах разности указателей, типе размеров объектов в данной модели памяти, а также примитивами для размещения и освобождения памяти для данной модели
- Распределители памяти позволяют свести задачу распределения памяти для сложных, составных объектов к совокупности более простых задач распределения памяти для более простых (составляющих) объектов

Распределители памяти STL

- Распределители памяти представляют собой классы объектов, которые можно включать в качестве параметров в описания шаблонов других классов:

```
template<class T, class A = allocator<T> > class vector {
public:
    typedef typename A::pointer iterator;    /* ..... */
private: A alloc;                          /* ..... */
public:                                     /* ..... */
    void reserve (size_type n)
        { if (n <= Nmax) return;
          iterator p = alloc.allocate (n);    /* ..... */
        }
};
```

Распределители памяти STL

- В классы распределителей памяти включаются определения типов, используемых для индексации соответствующих объектов (например, тип *size_type*)
- Эти типы могут представить самый большой объект исходного типа, соответствующий модели памяти
- В распределителях памяти описывается тип, соответствующий типу результата вычитания двух итераторов (тип *difference_type*)
- Обычно эти типы соответствуют типам *size_t* и *ptrdiff_t*, но они могут быть и иными, что зависит от природы объектов и от выбранной модели памяти

Распределители памяти STL

- Для распределителей памяти в библиотеке STL выработаны требования, сформулированные в виде перечня операций, которые можно выполнять над объектами классов распределителей для каждой отдельной модели памяти, к таким операциям относятся:
 - операции захвата памяти *allocate ()*
 - операции перераспределения памяти *deallocate ()*
 - операции конструирования указателей на объекты *address ()*
 - и т. д.

Алгоритмы STL

- Алгоритмы определяют вычислительные процедуры (*просмотр, сортировка, поиск, удаление элементов, ...*), не реализованные методами контейнеров, но пригодные для работы с разными составными структурами данных, в том числе с разными контейнерами
- Алгоритмы являются универсальными для любого из контейнеров и поэтому они реализованы без использования методов, входящих в контейнеры
- Алгоритмы выражаются шаблонами функции или наборами таких шаблонов
- Стандартные алгоритмы находятся в пространстве имён *std*, а их объявления – в заголовочном файле *<algorithm>*

Алгоритмы STL

- Алгоритмы библиотеки STL отделены от конкретных структур данных, над которыми они выполняются: алгоритм поиска данных не зависит от того, выполняется поиск в линейном массиве или списке
- Доступ к данным осуществляется только с помощью итераторов: о каждом алгоритме из библиотеки известно, с помощью какого вида итератора доступны используемые в алгоритме данные
- Благодаря параметризации итераторами, алгоритмы работают и со встроенными структурами данных, и со структурами данных, определёнными пользователями

Алгоритмы STL

- Обобщённые алгоритмы библиотеки:
 1. Немодифицирующие операции над последовательностями данных (*поиск и подсчёт числа элементов*):
 - *find ()* – первое вхождение элемента с заданным значением
 - *find_if ()* – первое вхождение элемента, удовлетворяющего заданному условию
 - *count ()* – количество вхождений элемента с заданным значением
 - *for_each ()* – операция-параметр применяется к каждому элементу, не меняя его

Алгоритмы STL

- Обобщённые алгоритмы библиотеки:
 2. Модифицирующие операции над последовательностями данных (*копирование, перестановки, преобразования, ...*), которые меняют либо сами элементы контейнера, либо их порядок, либо их количество:
 - *transform ()* – операция-параметр применяется к каждому элементу так, что содержимое контейнера меняется
 - *reverse ()* – переставляет элементы в последовательности
 - *copy ()* – создаёт новый контейнер

Алгоритмы STL

- Обобщённые алгоритмы библиотеки:
 3. Операции сортировки, поиска минимума и максимума, ускоренного поиска и т. д.
 - *sort ()* – простая сортировка, имеется также сортировка по возрастанию (в данном контейнере для типа элемента должна быть определена операция сравнения на меньше, то есть '*<*')
 - *stable_sort ()* – сортирует, но сохраняет порядок следования одинаковых элементов
 - *merge ()* – объединяет отсортированные последовательности

Алгоритмы STL

- Обобщённые алгоритмы библиотеки:
 4. Обобщённые численные алгоритмы
(*суммирование, смежные разности, слияние, обобщённое скалярное произведение, ...*)

Алгоритмы STL

- В языке Си++ критерий сравнения для функции сортировки *sort ()* реализуется с помощью параметра шаблонной функции:

```
template<class Iterator, class Predicate>
    inline void sort (Iterator First, Iterator Last, Predicate Pr)
    { _Sort_0 (First, Last, Pr, Val_type (First)); }
/* ... */
    { for (;;) ++ First) { for (; Pr (* First, Piv); ++ First);
                          for (; Pr (Piv, * -- Last));
                          if (Last <= First) return First;
                          iter_swap (First, Last);
                          }
    }
```

Алгоритмы STL

- Обобщённая функция сортировки `sort ()` может быть instantiated так, что её специализация будет упорядочивать элементы контейнера по убыванию:

```
class IntGreater
{ public: bool operator()(int x, int y) const {return x > y;} };

int main ()
{ int x [1024]; /* ..... */ // Инициализация
  sort (&x [0], &x [1024]); // По возрастанию
  sort (&x [0], &x [1024], IntGreater ());
                                     // По убыванию
}
```

Функциональные объекты STL

- Функциональные объекты управляют инкапсуляцией функций в объекте для их использования другими объектами
- Функциональные объекты – объекты, для которых определена операция группирования фактических параметров, то есть операция *operator ()*
- Использование функциональных объектов позволяет шаблонам алгоритмов работать не только с указателями на функции, но также и с любыми объектами, для которых возможна операция *operator ()*

Функциональные объекты STL

- Для проведения поэлементного суммирования двух векторов *a* и *b*, содержащих вещественные числа, и передачи результата в вектор *a* с помощью алгоритма *transform ()* и бинарного функционального объекта *plus ()* можно выполнить следующее:

```
template < class InputIterator1, class InputIterator2, class OutputIterator,
           class BinaryOperation >
    OutputIterator transform (InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, OutputIterator result,
                             BinaryOperation binary_op);

template < class T > struct plus : binary_function<T, T, T>
    { T operator () (const T& x, const T& y) const {return x + y;} };

transform (a.begin(), a.end(), b.begin(), a.begin(), plus<double>());

// res = op1 + op2 – для каждого элемента a и b
```

Функциональные объекты STL

- Для изменения знака элементов вектора можно использовать вариант алгоритма *transform()*, пригодный для унарных операций, и унарный функциональный объект *negate()*, выполняющий изменение знака без обращений к функциям:

```
template < class InputIterator,   class OutputIterator,  
           class UnaryOperation >  
    OutputIterator transform (InputIterator first, InputIterator last,  
                             OutputIterator result, UnaryOperation op);
```

```
template < class T > struct negate : unary_function<T, T>  
    { T operator () (const T& x) const {return -x;} };
```

```
transform (a.begin(), a.end(), a.begin(), negate<double>());
```

// res = - op – для каждого элемента *a*

Адаптеры STL

- Адаптеры используются для преобразования интерфейсов
- В библиотеку включены
 - Адаптеры контейнеров
 - Адаптеры итераторов
 - Адаптеры функциональных объектов
- Применение адаптеров позволяет на основе базовых классов строить производные классы, обеспечивающие удобное и эффективное представление данных и операций над ними

Адаптеры контейнеров STL

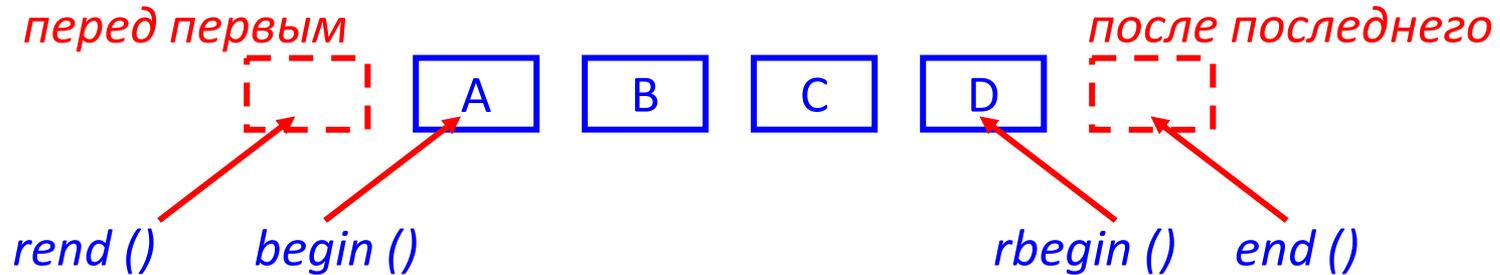
- **Адаптеры контейнеров** позволяют строить ограниченные интерфейсы (из базовых интерфейсов удаляются лишние операции):
 - для векторов (`vector`) => `priority_queue`, `stack`
 - для списков (`list`) => `queue`, `stack`
 - для двусторонних очередей (`deque`) => `queue`, `priority_queue`, `stack`
- **Адаптеры контейнеров** не имеют своих итераторов, предполагая использование базовых итераторов, в каждом конкретном случае для доступа к данным стека используются итераторы векторов, списков или двусторонних очередей:

```
stack<vector<int> >    // стек целых чисел на базе вектора
queue<list<char> >    // очередь символов на базе списка
```
- **Адаптеры контейнеров** – простые интерфейсы, создаваемые для тех контейнеров, типы которых передаются адаптерам в качестве фактических параметров

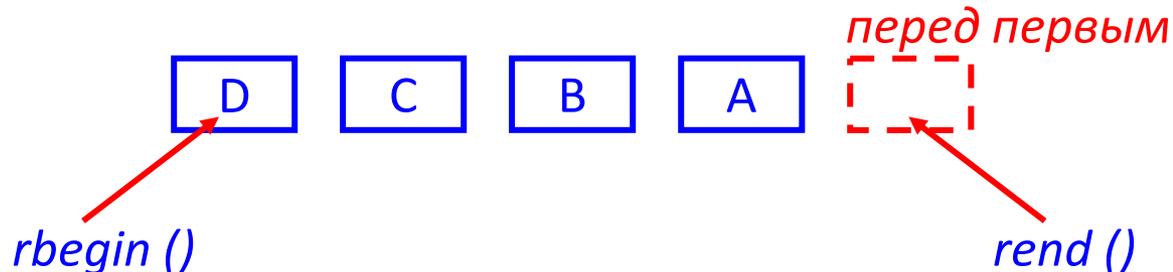
Адаптеры итераторов STL

- *Адаптеры итераторов* выполняют функции, аналогичные адаптерам контейнеров, но преобразования интерфейсов производятся в отношении классов итераторов
- Двунаправленные итераторы и итераторы с произвольным доступом имеют соответствующие *адаптеры обратных итераторов*, которые могут продвигаться по структурам данных в обратном направлении
- Контейнеры, допускающие работу с двунаправленными итераторами и итераторами произвольного доступа, содержат методы:
 - *reverse_iterator rbegin ()*, *const_reverse_iterator rbegin ()* – возвращают итератор, указывающий на первый элемент в обратной последовательности
 - *reverse_iterator rend ()*, *const_reverse_iterator rend ()* – возвращают итератор, указывающий на элемент, следующий за последним в обратной последовательности

Адаптеры итераторов STL



- С помощью обратных итераторов последовательный доступ к элементам данных контейнерных типов осуществляется от последнего элемента к первому:



Адаптеры итераторов STL

- При проходе последовательности как прямым итератором p , так и обратным итератором rp переход к следующему элементу записывается с помощью операции увеличения $++rp$ (но не $--rp$):

```
template<class C> typename C::value_type sum_twice (const C& c)
{
    typename C::value_type s1, s2;
    typename C::const_iterator p_p = c.begin ();
    s1 = 0; while (p_p != c.end ()) s1 += * (p_p ++);
    typename C::const_reverse_iterator p_r = c.rbegin ();
    s2 = 0; while (p_r != c.rend ()) s2 += * (p_r ++); return (s1 + s2) / 2; }
```

- **Примечание**: Шаблонные конструкции вида " $T::x$ ", в которых используется имя типа T , введённое в параметрах шаблона, независимо от контекста интерпретируются как "член-данное x из класса T ". В таких случаях для использования *типа* x из класса T пишут " $typename T::x$ ".

Адаптеры итераторов STL

- Использование в алгоритмах той же операции увеличения '++', что и для обычного итератора, позволяет использовать обратные итераторы с библиотечными функциями в тех случаях, когда использование этих функций могло бы оказаться затруднительным
- Например, для организации поиска в контейнере в обратном порядке (от конца к началу) обычно пишутся такие циклы:

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

Адаптеры итераторов STL

- Применив обратный итератор, можно воспользоваться библиотечной функцией поиска со всеми её преимуществами и без потери эффективности
- Операция `i = ri.base ()` выдаёт значение `i` типа `iterator`, указывающее на один элемент вперёд позиции обратного итератора `ri`.

`&*(reverse_iterator (iterator)) == &*(iterator - 1)`

```
template<class C> typename C::const_iterator find_last
    (const C& c, typename C::value_type v)
{
    typename C::const_reverse_iterator ri =
        find (c.rbegin (), c.rend (), v);
    if (ri == c.rend ()) return c.end ();
    typename C::const_iterator i = ri.base ();    return -- i;
} // Известно, что функция find () пользуется операцией '++'!
```

Адаптеры итераторов STL

- В библиотеку STL включён адаптер вставки, который заменяет обычную операцию изменения значения элементов контейнера на операцию вставки элементов в контейнер
- Для обычных классов итераторов фрагмент программы

```
while (first != last) *result ++ = *first ++;
```

означает копирование элементов диапазона *[first, last)* в диапазон, начинающийся с итератора *result*, однако, если итератор *result* является итератором вставки, тот же самый фрагмент будет производить вставку дополнительных элементов в контейнер.

- Адаптеры итераторов неинициализированной памяти применяют для записи результатов операций в неинициализированную память, то есть в память, не содержащую никаких объектов данных, про которую известно лишь, что её размер достаточен для хранения соответствующего результата

Контейнер векторов (*vector*)

- Векторы, строящиеся на основе контейнеров класса *vector*, по своим свойствам напоминают обычные одномерные массивы:

```
#include <vector>
```

```
using namespace std;
```

```
template<class T, class A = allocator<T> > class vector;
```

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

- Конструктор *vector<int> v (10)* задаёт вектор из 10 целых чисел, имеются и другие виды конструкторов:

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

```
vector (iterator first, iterator last, const A& = A ());
```

```
explicit vector (const A& = A ()); // требуется явный вызов:
```

```
// vector<int> x (10);
```

```
explicit vector(size_type size, const T&value = T(), const A& a = A());
```

Контейнер векторов (vector)

- Как и массив, вектор представляет собой непрерывную последовательность элементов, но, в отличие от обычных массивов, размер вектора не известен статически

```
bool empty    () const { /* ... */ } //истина, если контейнер пуст  
size_type size () const { /* ... */ } //выдача текущего размера
```

- Методы, связанные с итераторами:

```
iterator begin ();          const_iterator begin () const;  
Iterator end   ();          const_iterator end   () const;
```

- Операции, возвращающие не значения итераторов на элементы вектора, а сами значения этих элементов:

```
reference front () { return * begin (); }  
reference back  () { return *(end () - 1); }
```

Контейнер векторов (vector)

- Для векторов легко написать программы копирования элементов, как в прямом, так и в обратном порядке, то есть от конечного элемента к начальному:

```
int main ()
{ vector<int> v (100, 5); // 100 элем. с начальным значением 5
  vector<int>::const_iterator p = v.begin ();
  vector<int>::const_reverse_iterator q = v.rbegin (); // ...
  while (p != v.end ()) { cout << * p << ' '; ++ p; } // ...
  while (q != v.rend ()) { cout << * q << ' '; ++ q; } // ...
  return 0;
}
```

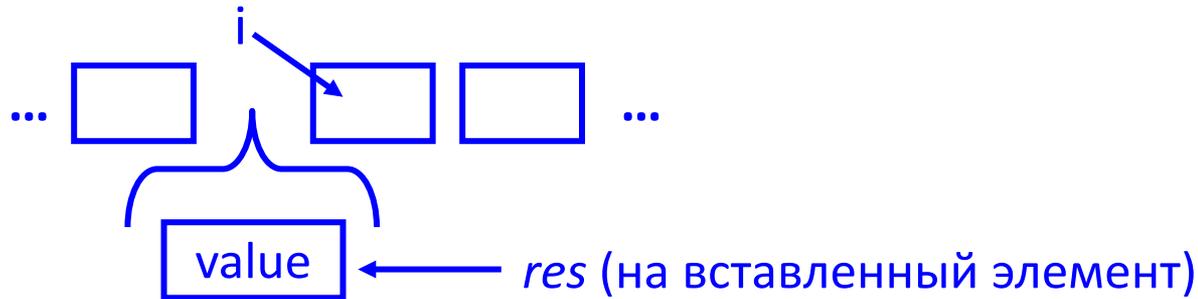
Контейнер векторов (vector)

- При создании вектора задаётся только начальный размер
- Операции *push_back ()* и *pop_back ()* изменяют количество элементов вектора: первая, чтобы добавить один или несколько элементов к концу вектора, а вторая, чтобы уничтожить один или несколько последних элементов
- Операции вставки и удаления элементов приводят к перемещению (копированию) некоторого числа элементов вектора на новое место: при вставке элементы копируются, чтобы освободить достаточно места для всех вставляемых элементов, при удалении элементы копируются, чтобы в векторе не оставалось несуществующих элементов

Контейнер векторов (vector)

- Вставка перед элементом:

`iterator insert (iterator i, const T& value) {...}`

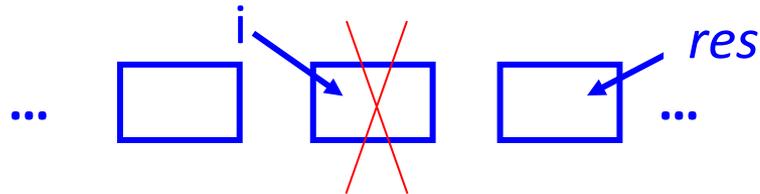


- Вставка нескольких одинаковых элементов перед элементом:
`iterator insert (iterator i, size_type number, const T & value){...}`
- Вставка в конец контейнера:
`void push_back (const T& value) { insert (end (), value); }`
- Уничтожение всех элементов вектора без вызова деструктора самого вектора:
`void clear () { erase (begin (), end ()); }`

Контейнер векторов (vector)

- Уничтожение заданного элемента и выдача итератора элемента, следующего за удалённым:

```
iterator erase (iterator i) { /* ... */ return (i); }
```

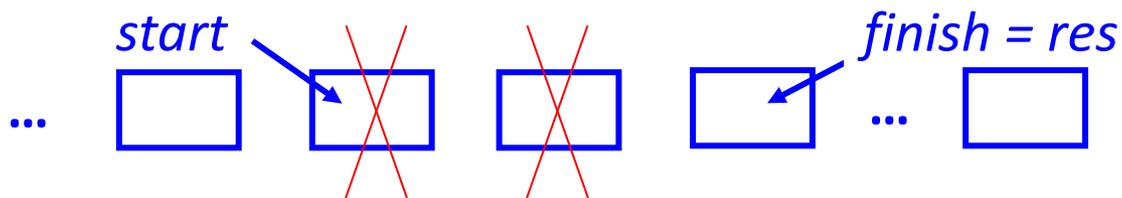


- Уничтожение последнего элемента:

```
void pop_back () { erase (end () - 1); }
```

- Уничтожение диапазона и выдача итератора элемента, следующего за удалённым:

```
iterator erase (iterator start, iterator finish) { /*...*/ return (finish); }
```



Контейнер векторов (vector)

- Операции *insert ()* и *erase ()* определены только для обычных итераторов, поэтому организуя циклы по обратным итераторам, эти обратные итераторы надо сначала преобразовывать к обычным, а лишь затем выполнять вставку или уничтожение элементов:

```
vector<int>::reverse_iterator ri = v.rbegin ();
while (ri != v.rend ())
    if (* ri ++ == Element)
        { vector<int>::iterator i = ri.base ();
          v.insert (i, - Element); // перед i-тым элементом вставить
          break;                // ещё один, с обратным знаком
        }
```

Контейнер векторов (vector)

- К контейнеру *vector* применима операция индексации *operator []*, обеспечивающая доступ к отдельным элементам вектора в произвольном порядке:

```
reference operator [] (size_type i) { return * (begin () + i); }
```

- Для доступа к элементам вектора используются итераторы с произвольным доступом (следствие возможности индексации)
- Операция индексации не проверяет выход за границы контейнера, однако, к контейнеру применима функция *at ()*, которая выполняет такой же индексированный доступ, но с проверкой диапазона индекса:

```
reference at (size_type i) { /* ... */ } // содержимое элемента  
// с номером i
```

Контейнер векторов (vector)

- При выходе за пределы разрешённого диапазона вектора (то есть при значении индекса меньше самого маленького разрешённого или больше самого большого) возбуждается исключительная ситуация *out_of_range*, поэтому при работе с функцией *at ()* используется перехватчик исключения:

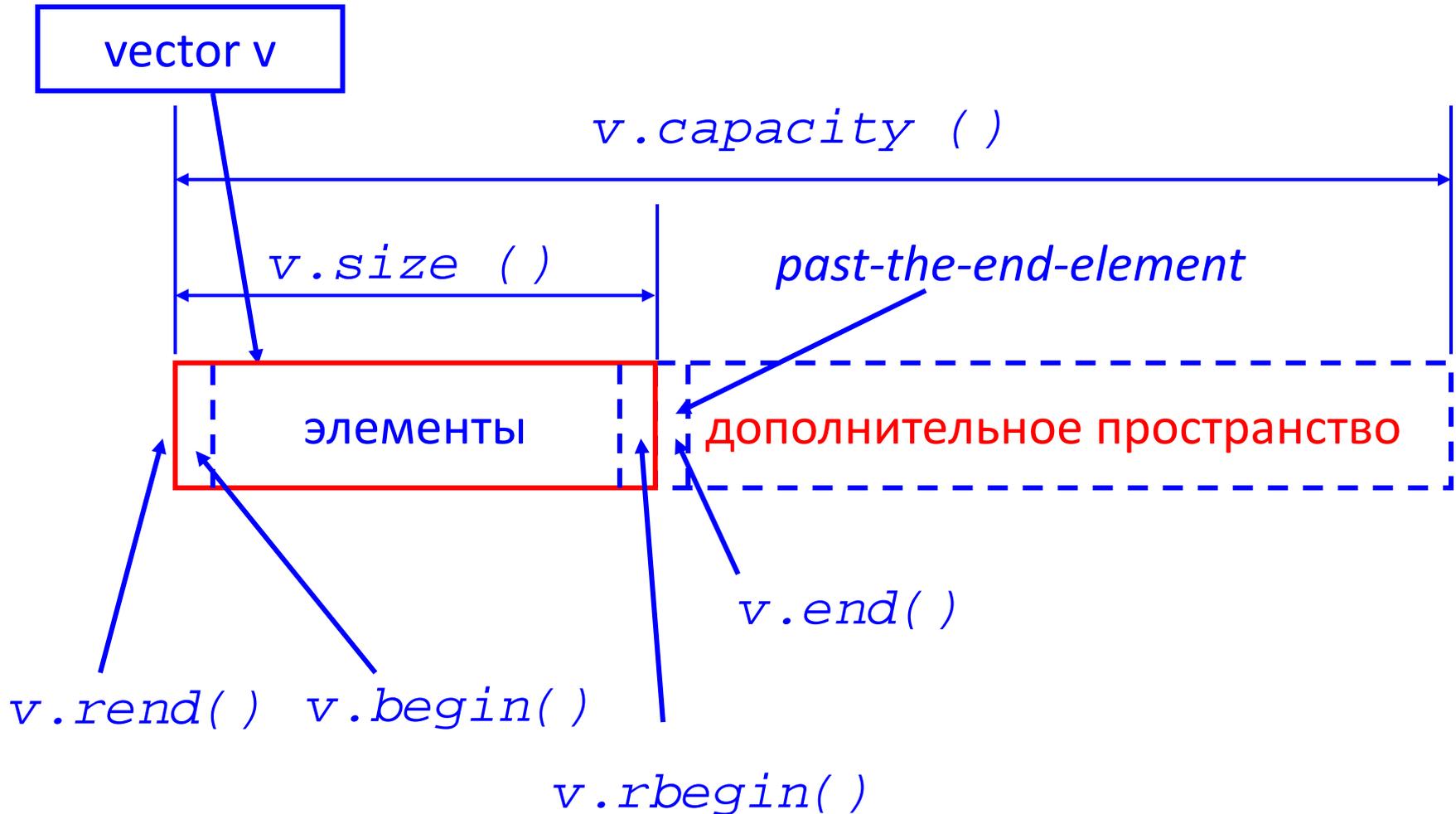
```
try          { /* ... */ v [5] = v.at (i); /* ... */ }  
catch (out_of_range) { /* ... */ /* ... */ }
```

- Нельзя выполнять продвижение итератора (*it = it + n*), если результат будет указывать за *past-the-end-element* (ошибка фиксируется самой операцией сложения, ещё до записи результата)

Контейнер векторов (vector)

- Функция *size ()*, не имеющая параметров, позволяет узнать число элементов вектора, функция *resize ()* меняет число элементов вектора на другое
- С помощью функции *reserve ()* можно выполнить предварительное резервирование памяти:
 1. В программах можно сразу выделить максимально необходимое пространство одной операцией вместо многократного выделения меньших фрагментов памяти
 2. Резервирование достаточного места позволяет гарантировать сохранность значений индексов и указателей на некоторый период обработки вектора
- Операция *reserve ()* отличается от операции *resize ()* тем, что число элементов вектора не меняется, функция *capacity ()* выполняется для определения размера зарезервированной памяти

Контейнер векторов (vector)



Контейнер векторов (vector)

```
#include <vector>
using namespace std;
typedef vector<int> Container;
typedef Container::size_type Cst;
void f (Container& v, int i1, int i2) {
    try { for (Cst i = 0; i < 10; i++) { v.push_back (i); } // Элементы: 0, 1, 2, ..., 9.
        v.at (i1) = v.at (i2);
        cout << v.size (); // Размер контейнера для данной точки
        Container::iterator p = v.begin ();
        p += 2; // Для векторов можно, для других – advance (p, 2)
        v.insert (p, 100); // Элементы: 0, 1, 100, 2, ..., 9. p теряет значение
        sort (v.begin (), v.end ()); // Сортировка диапазона
        for (Cst i = 0; i < v.size (); i ++) {cout << v [i]; }
    }
    catch (out_of_range) { /* ... */ // реакция на ошибочный индекс
}
int main () { Container v; f (v, 5, 12); }
```

Контейнер списков (list)

- Списки, строящиеся на основе стандартного контейнера *list*, позволяют осуществлять вставки и уничтожения любых элементов за постоянное время, произвольный доступ для списков применяться не может, то есть уровень разрешённого итератора – двунаправленный:

```
#include <list>
```

```
using namespace std;
```

```
template<class T, class A = allocator<T> > class list;
```

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

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

```
list (iterator first, iterator last, const A& = A ());
```

```
explicit list (const A& = A ());
```

```
explicit list (size_type size, const T& value = T (), const A& a = A ());
```

Контейнер списков (list)

- Размер списка не известен статически

```
bool empty    () const { /* ... */ } //истина, если контейнер пуст
size_type size () const { /* ... */ } //выдача текущего размера
```

- Методы, связанные с итераторами:

```
iterator begin ();          const_iterator begin () const;
Iterator end   ();          const_iterator end   () const;
```

- Операции, возвращающие не значения итераторов на элементы списка, а значения самих этих элементов:

```
reference front () { return * begin (); }
reference back  () { return *(end () - 1); }
```

Контейнер списков (list)

- Кроме операций *push_back ()* и *pop_back ()*, которые добавляют и удаляют элементы в конце списка, контейнер списков содержит операции *push_front ()* и *pop_front ()*, добавляющие и удаляющие элементы в начале списка:

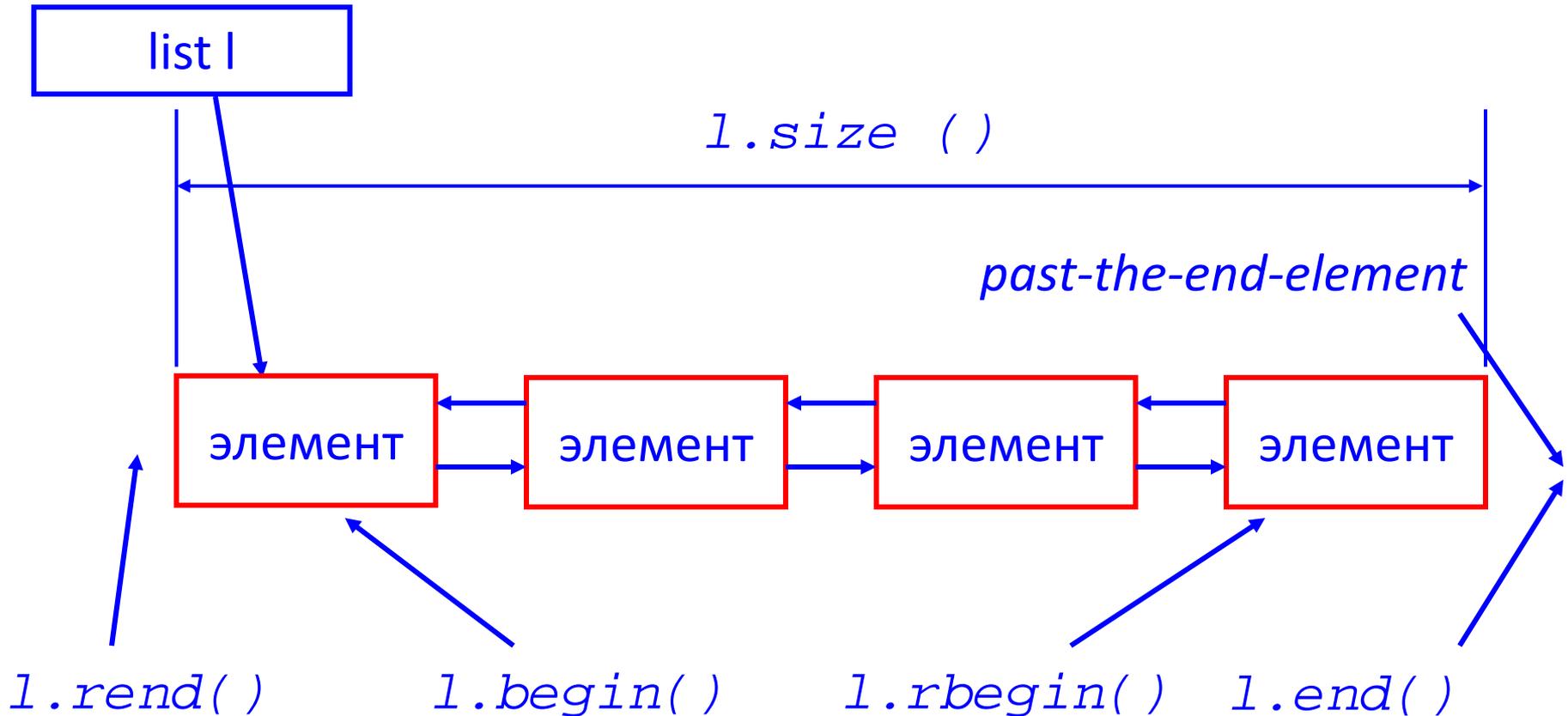
```
void push_front (const T& value) { insert (begin (), value); }  
void pop_front ()                { erase (begin ()); }
```

- Операция *insert ()* (добавление нового элемента к списку) может потребовать захвата дополнительной памяти

Контейнер списков (list)

- Вставка элементов не влияет на значимость итераторов (указателей) и ссылок на элементы
- Уничтожение элементов влияет на значимость только тех итераторов и ссылок, которые относятся к уничтожаемым элементам
- Для списков введены операции:
 - *remove ()* – уничтожение ненужных элементов списков
 - *splice ()* – перемещение элементов из одного списка в другой с уничтожением перемещённых элементов
 - *unique ()* – замена последовательности одинаковых элементов списка одним элементом
 - *merge ()* – слияние двух списков
 - *reverse ()* – реверсирование списка
 - *sort ()* – сортировка списка

Контейнер списков (list)



Контейнер векторов (vector)

```
#include <vector>
using namespace std;
typedef vector<int> Container;
typedef Container::size_type Cst;
void f (Container& v, int i1, int i2) {
    try { for (Cst i = 0; i < 10; i++) { v.push_back (i); } // Элементы: 0, 1, 2, ..., 9.
        v.at (i1) = v.at (i2);
        cout << v.size (); // Размер контейнера для данной точки
        Container::iterator p = v.begin ();
        p += 2; // Для векторов можно, для других – advance (p, 2)
        v.insert (p, 100); // Элементы: 0, 1, 100, 2, ..., 9. p теряет значение
        sort (v.begin (), v.end ()); // Сортировка диапазона
        for (Cst i = 0; i < v.size (); i ++) {cout << v [i]; }
    }
    catch (out_of_range) { /* ... */ // реакция на ошибочный индекс
}
int main () { Container v; f (v, 5, 12); }
```

Контейнер списков (list)

```
#include <list>
using namespace std;
typedef list<int> Container;
typedef Container::size_type Cst;
void f (Container& v, int i1, int i2) {
    try { for (Cst i = 0; i < 10; i++) { v.push_back (i); } // Элементы: 0, 1, 2, ..., 9.
        // v.at (i1) = v.at (i2); Требуется перепрограммирование!
        cout << v.size (); // Размер контейнера для данной точки
        Container::iterator p = v.begin ();
        advance (p, 2);
        v.insert (p, 100); // Элементы: 0, 1, 100, 2, ..., 9.
        sort (v.begin (), v.end ()); // Сортировка диапазона
        for (p = v.begin (); p != v.end (); ++ p) cout << * p;
    }
    catch (out_of_range) { /* ... */ // реакция на ошибочный индекс
}
int main () { Container v; f (v, 5, 12); }
```

Достоинства STL-подхода

- Все контейнеры обеспечивают стандартный интерфейс в виде набора операций, один контейнер может использоваться вместо другого, это не влечёт серьёзного изменения текста программы
- Дополнительная общность использования обеспечивается через стандартные итераторы
- Каждый контейнер связан с распределителем памяти, 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

Достоинства STL-подхода

- Контейнеры по определению однородны, то есть должны содержать элементы одного типа, но возможно создание разнородных контейнеров как контейнеров указателей на общий базовый класс
- Алгоритмы, входящие в состав STL, предназначены для работы с содержимым контейнеров, все алгоритмы представляют собой шаблонные функции, следовательно, их можно использовать для работы с любым контейнером

Недостатки STL-подхода

- Не являясь производными от некоторого базового класса, контейнеры не имеют фиксированного стандартного представления (это же верно и для итераторов), использование стандартных контейнеров и итераторов не подразумевает никакой явной или неявной проверки типов во время выполнения
- Каждый доступ к итератору приводит к вызову виртуальной функции, затраты по сравнению с вызовом обычной функции могут быть значительными
- Предотвращение выхода за пределы контейнера возлагается на программиста, при этом специальных средств контроля не предлагается

Новости стандарта Си++

- Полностью разрешено использование типа
`long long int`
- Введены типизированные перечисления
`enum class E { V1, V2, V3 = 100, V4 /*101*/ };`
`if (E::V4 == 101) ... /* Ошибка! */`
- Тип констант не обязательно должен быть `int`
`enum class E { V1, V2, V3 = 100, V4 /*101*/ };`
`if (E::V4 == 101) ... /* Ошибка! */`
- Тип констант перечисления можно задать явно:
`enum class E2 : unsigned int { V1, V2 };`
Здесь значение `E2::V1` определено, а `V1` – нет

Новости стандарта Си++

- Для перечисления, заданного не строго:

```
enum E3 : unsigned long { V1 = 1, V2 };
```

определены оба значения: и `E3::V1`, и значение `V1`

- Возможно предварительное объявление перечислений, но только с указанием размера:

```
enum E1; // Ошибка: низлежащий тип не определён
```

```
enum E2 : unsigned int; // ОК!
```

```
enum class E3; // ОК: низлежащий тип int
```

```
enum class E4 : unsigned long; // ОК!
```

```
enum E2 : unsigned short;
```

```
// Ошибка: E2 ранее объявлен с другим типом
```

Новости стандарта Си++

- Имя *NULL* в языке Си++ определяется как константа со значением *0*
- Перегрузка функций может приводить к неверному выбору функции *f ()* при вызове вида *f (NULL)*:

```
int f (char * p); int f (int i);    i = f (NULL); // int
```

- В новом стандарте введено новое ключевое слово *nullptr* для значения пустого указателя типа *std::nullptr_t*:

```
char * pc = nullptr;    int * pi = nullptr; // ОК!
```

```
bool b = nullptr; // ОК: b == false
```

```
int i = nullptr; // Ошибка!
```

```
i = f (0); /* int */    i = f (nullptr); // char*
```

Новости стандарта Си++

- Новым стандартом расширены возможности контейнеров, в частности, разрешена их инициализация, подобная инициализации встроенных массивов:

```
a [] = { 1, 2, 3, 4 };
```

```
vector<int> v [] = { 1, 2, 3, 4 };
```

- Для упорядоченных контейнеров введены методы, гарантированно дающие значения константных итераторов:

```
const_iterator cbegin ();    const_iterator cend ();
```

Новости стандарта Си++

- Новым стандартом введено ключевое слово ***constexpr***:

```
constexpr int give5 () { return 5; }
```

```
int mas [give5 () + 7];    // массив из 12 элементов
```

- Ограничения:
 - функция не может быть типа ***void***
 - тело функции должно иметь вид ***return выражение***
 - ***выражение*** может вызывать только те функции, что также обозначены ключевым словом ***constexpr***, или просто использовать обычные константы
 - функция, обозначенная с помощью ***constexpr***, не может вызываться до её определения

Новости стандарта Си++

- В константных выражениях можно использовать переменные любых числовых типов

```
constexpr double a = 9.8;
```

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

- Операцию определения размера объекта разрешено применять к членам-данным классов независимо от самих объектов классов

```
struct A { some_type a; }; /* ... */
```

```
sizeof (A::a) ... // ОК!
```

Новости стандарта Си++

- В старом стандарте требование указывать тип объекта в некоторых языковых конструкциях сильно их усложняло:

```
for (vector<double, My alloc<double> >::const iterator  
      p = v.begin (); p != v.end (); ++ p)  
    cout << *p << endl;
```

- Новый стандарт допускает автоматическое выведение типа объекта из типа иницирующего значения, что более экономно с точки зрения текста программы и более устойчиво к ошибкам:

```
for (auto p = v.cbegin (); p != v.cend (); ++ p)  
    cout << *p << endl;
```

Новости стандарта Си++

- Если *ft ()* – функция, возвращающая значение некоторого типа, переменная *var1* будет иметь соответствующий тип

```
auto var1 = ft ( /* ... */ );
```

- Возможно также:

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

- Допускаются также определения:

```
int v1;  
decltype (v1) v2 = 5;
```

Новости стандарта Си++

- Пересчётный цикл по коллекции для встроенных массивов и коллекций с функциями *begin ()* и *end ()*, возвращающими итераторы:

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

- Делегирующие конструкторы:

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

- Инициализация членов-данных класса в области их объявления в классе:

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

Новости стандарта Си++

```
struct B { virtual void some_func ();
           virtual void f (int);
           virtual void g ()           const;
};
struct D1 : public B {
           virtual void some_func ()           override; // Ошибка: нет такой функции в B
           virtual void f (int)               override; // ОК!
           virtual void f (long)             override; // Ошибка: несоответствие типа
           virtual void f (int)              const override; // Ошибка: несоответствие
           virtual int  f (int)               override; // Ошибка: несоответствие типа
           virtual void g ()                 const final;   // ОК!
           virtual void g (long);            // ОК: новая виртуальная функция
};
struct D2 : D1 { virtual void g ()           const;           // Ошибка: g () - финальная функция
};

struct F    final { int x,y; };
struct G : F {    int z;           // Ошибка: наследование от финального класса
};
```

Новости стандарта Си++

```
class Str { char * s;  
           int len;  
public:  
    Str (const char * sss = NULL); // традиционный конструктор неплоского класса  
    Str (const Str &);             // традиционный конструктор копирования  
    Str (Str && x)                 // конструктор переноса из временного объекта  
        { s = x.s;  
          x.s = NULL; // !!!  
          len = x.len;           }  
  
    Str & operator = (const Str & x); // традиционная перегруженная операция =  
    Str & operator = (Str && x)      // быстрый перенос временного объекта  
        { s = x.s;  
          x.s = NULL; // !!!  
          len = x.len;  
          return *this;         }  
  
    ~Str ();                       // традиционный деструктор неплоского класса  
    Str operator + (Str x);         // ...  
};
```