Гусин А.Н., Викентьева О.Л.

Проектирование программ и программирование на C++

Часть II: Объектно-ориентированное программирование

Объекты и классы

Объекты и классы

- Идея классов отражает строение объектов реального мира — каждый предмет или процесс обладает набором характеристик или отличительных черт, иными словами, свойствами и поведением.

- *Объект* является автономным действующим лицом в системе. Реальными кандидатами на роли объектов обычно выступают люди, места, вещи, организации, концепции и события.

- Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс.

- Класс является определяемым пользователем абстрактным типом данных, описывающим свойства и поведение какого-либо предмета или процесса посредством полей данных (аналогично структуре) и функций для работы с ними.

Основные принципы объектно-ориентированного программирования

Основные принципы ООП

- **Абстрагирование** концентрирует внимание на внешних особенностях объекта и позволяет отделить существенные особенности поведения от несущественных. Существуют различные типы абстракций, начиная с объектов, которые почти точно соответствуют физическим объектам предметной области, и, заканчивая объектами, которые не могут существовать в реальном мире.

Абстракция сущности	Объект представляет собой модель некоторого физического предмета (объекта) предметной области.
Абстракция поведения	Объект состоит из обобщенного множества операций.
Абстракция виртуальной машины	Объект группирует операции, которые либо используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня.
Произвольная абстракция	Объект включает в себя набор операций, не имеющих друг с другом ничего общего.

Основные принципы ООП

- Процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение, называется *инкапсуляцией.*

- Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

- Чаще всего инкапсуляция выполняется посредством скрытия всех внутренних деталей, не влияющих на внешнее поведение объекта. Обычно скрываются и внутренняя структура объекта, и реализация его методов. Это означает наличие двух частей в классе: интерфейса и реализации.

- Интерфейс отражает внешнее поведение всех объектов данного класса.

- Инкапсуляция позволяет изменить реализацию класса без изменения части программы, если интерфейс остался прежним.

- Такой подход приводит к контрактной модели программирования, которая заключается в разграничении внешнего облика, то есть интерфейса, и внутреннего устройства класса, реализации.

Основные принципы ООП

- *Иерархия* – это упорядочение абстракций, за счет чего достигается значительное упрощение понимания сложных задач.

- Основными видами иерархических структур применительно к сложным системам являются:
  - структура классов (иерархия "общее-частное");
  - структура объектов (иерархия "целое-часть").
- *Иерархия "общее-частное"* представлена концепцией *наследования*. Под этим подразумевается возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые.

- Свойства при наследовании повторно не описываются, что сокращает объем программы.

- Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные — на ветвях и листьях.

- В C++ каждый класс может иметь сколько угодно потомков и предков.

Основные принципы ООП

- *Иерархия "целое-часть"* вводит отношение *агрегации*. В качестве примера можно привести отношение "организация – сотрудники", когда целое является чем-то большим, чем просто набором своих частей.

- *Модульность* – это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.

- Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность. Кроме того, внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов, которые необходимы для исчерпывающего понимания программы в целом.

- При построении объектной модели рекомендуется разделять систему на отдельные компоненты, связанные с:
  - проблемной областью (классы бизнес-уровня),
  - взаимодействием с человеком (окна и отчеты),
  - управлением данными (работа с СУБД),
  - взаимодействием с другими системами.

Основные принципы ООП

- Под *типизацией* понимается способ защититься от использования объектов одного класса вместо другого, или, по крайней мере, управлять таким использованием. *Тип* определяют как точную характеристику свойств, включая структуру и поведение, относящуюся к некоторой совокупности объектов.

- Если стоит задача защититься от подмены понятий, например, когда одна функция в качестве параметра должна принимать значение целого типа, выражающее количество денежных средств, а ей передают значение целого типа, но выражающее количество человек в подразделении, следует вводить специальные типы вместо использования стандартного на все случаи жизни.
- Объявление, начинающееся с ключевого слова `typedef` в C++, вводит новое имя для типа, а не для переменной данного типа. Имена, вводимые `typedef`, являются синонимами, а не новыми типами.
- Преимущества сильной типизации:
  - отсутствие контроля типов может приводить к загадочным сбоям в программах во время их выполнения;
  - в большинстве систем процесс редактирование-компиляция-отладка утомителен, и раннее обнаружение ошибок просто незаменимо;
  - объявление типов улучшает документирование программ;
  - многие компиляторы генерируют более эффективный объектный код, если им явно известны типы.

Основные принципы ООП

- Также типизация определяет время, когда имена связываются с типами.
- Статическая связь означает, что типы всех переменных и выражений известны во время компиляции; *динамическое* (или позднее) *связывание* означает, что типы неизвестны до момента выполнения программы.

- С понятием динамического связывания тесно связана концепция *полиморфизма* – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы.

- *Параллелизм* позволяет различным объектам действовать одновременно. Это свойство, отличающее активные объекты от пассивных.

- **Активным** называется объект, который может представлять собой отдельный поток управления (абстракцию процесса).

- **Пассивный** объект, напротив, может изменять свое состояние только под воздействием других объектов. Для систем, построенных на основе ООП, мир может быть представлен, как совокупность взаимодействующих объектов.

- **Сохраняемость** – это способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

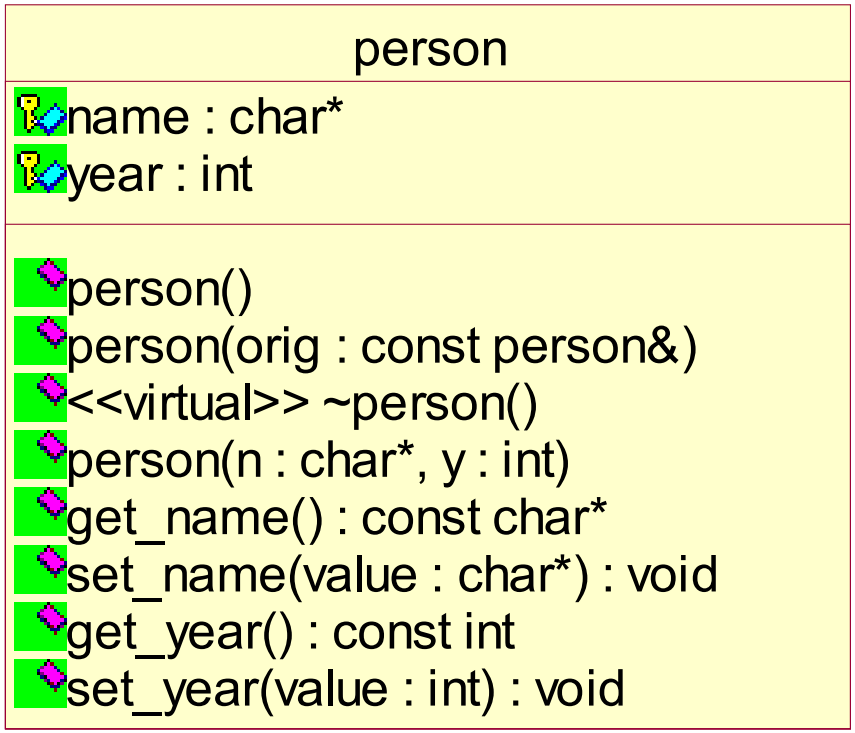
Контрольные вопросы

- 1) Что такое класс и объект?
- 2) Что такое абстрагирование? Виды абстракций.
- 3) Что такое инкапсуляция?
- 4) Что из себя представляет иерархия?

Описание классов

Описание классов

- В нотации UML класс обозначается в виде прямоугольника, разделенного на три части. В верхней содержится имя класса, в средней – его атрибуты (поля данных). В нижней части указываются методы класса, отражающие его поведение (то есть действия, выполняемые классом).



Описание классов

- Описание класса в общем виде выглядит так:

```
class <имя_класса>
{
      [private:]
      <описание скрытых элементов>
      public:
      <описание доступных элементов>
};
```

- ***Состояние объекта*** характеризуется перечнем (обычно неизменным) всех свойств данного объекта и текущими (обычно изменяемыми) значениями каждого из этих свойств. Тот факт, что всякий объект имеет состояние, означает, что всякий объект занимает определенное пространство (физически или в памяти компьютера).

- К числу свойств относятся присущие объекту или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Эти свойства принято называть *атрибутами класса*.

Описание классов

- Атрибуты содержатся внутри класса, поэтому они скрыты от других классов. В связи с этим иногда требуется указать, какие классы имеют право читать и изменять атрибуты. Это свойство называется *видимостью* атрибута.
- У атрибутов и операций, в зависимости от их назначения и требований доступности, определяют следующие значения этого параметра:
  - `public` (открытый). В этом разделе размещают атрибуты, доступные всем остальным классам. Любой класс может просмотреть или изменить их значением. В нотации UML такой атрибут обозначается знаком **"+"**.
  - `private` (закрытый). Такой атрибут не виден никаким другим классам, кроме дружественных. Закрытому атрибуту предшествует символ **"–"**.
  - `protected` (защищенный). Атрибуты этого раздела доступны только самому классу, его потомкам и друзьям (`friend`). Его признак – символ **"#"**.
- Можно задавать несколько секций `private` и `public`, порядок их следования значения не имеет.
- Видимостью элементов класса можно также управлять с помощью ключевых слов `struct` и `class`. Если при описании класса используется слово `struct`, то все поля и методы по умолчанию будут общедоступными (`public`). Если при описании класса используется слово `class`, то по умолчанию все методы и поля класса будут скрытыми (`private`).

Описание классов

- *Свойства атрибутов класса:*
  - могут иметь любой тип, кроме типа этого же класса (но могут быть указателями на этот класс);
  - могут быть описаны с модификатором `const`, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
  - Инициализация атрибутов при описании не допускается.
  - Если тело метода определено внутри класса, он является встроенным (`inline`). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости (`::`).

Описание классов

```
//person.h
//класс описывает свойства личности в информационной системе
class person
{
public:
   person();     //конструктор без параметров
   person(const person& orig); //конструктор копирования
   person(char* n, int y);      //конструктор с параметрами
   virtual ~person();   //деструктор

   const char* get_name() ;    //получение имени человека - селектор
   const int get_year();       //получение года рождения человека

   //изменение имени человека - модификатор
   void set_name(char* value);

   //изменение года рождения человека - модификатор
   void set_year(int value);

protected:
   char* name; //имя человека, указатель на строку символов
   int year;   //год рождения человека, выражаем целым числом
};
```

Описание классов

- *Поведение* – это то, как объект действует и реагирует на события, происходящие в системе. Поведение объекта определяется выполняемыми над ним операциями и его состоянием.

- Для того чтобы создать объекты класса `person`, напишем:

```
//main.cpp
#include "person.h"//подключаем заголовочный файл описания класса
void main()
{
     person A, B, C;
}
```

- В данном случае объявлено три различных объекта и каждый из них занимает определенный участок в памяти. Все объекты в системе имеют некоторое состояние, а состояние системы заключено в объектах.

Описание классов

- Объектно-ориентированный стиль программирования связан с воздействием на объекты путем передачи им сообщений (т.е. обращения к методам, описанным в классе объекта). Операция над объектом порождает некоторую реакцию этого объекта.
- Операция – это услуга, которую класс может предоставить своим клиентам. На практике над объектами можно совершать операции пяти видов:

Конструктор	Определяет способ создания объекта или его инициализации; имеет то же имя, что и класс.
Деструктор	Операция, выполняющая очистку памяти, когда объект класса выходит за пределы области видимости или он удаляется; имеет то же имя, что и класс со знаком "~" перед ним.
Модификатор	Операция, которая изменяет состояние объекта.
Селектор	Операция, считывающая состояние объекта, но не меняющая состояния.
Итератор	Операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности.

Описание классов

- Объекты взаимодействуют между собой, посылая и получая сообщения.

- **Сообщение** — это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью вызова соответствующих функций.

- Таким образом, с помощью ООП легко реализуется так называемая событийно-управляемая модель, когда данные активны и управляют вызовом того или иного фрагмента программного кода.

- Примером реализации событийно-управляемой модели может служить любая программа, управляемая с помощью меню. После запуска такая программа пассивно ожидает дейст-вий пользователя и должна уметь правильно отреагировать на любое из них. Событийная модель является противоположностью традиционной (директивной), когда код управляет данными: программа после старта предлагает пользователю выполнить некоторые дейст-вия (ввести данные, выбрать режим) в соответствии с жестко заданным алгоритмом.

Контрольные вопросы

- 1) Как в общем виде выглядит описание класса?
- 2) Что такое атрибуты класса?
- 3) Чем характеризуется состояние объекта?
- 4) Какие существуют спецификаторы доступа?
- 5) Какие свойства атрибутов вам известны?
- 6) Что такое поведение объекта?
- 7) Какие операции можно совершать над объектами?

Конструкторы

Конструкторы

- Для инициализации объектов класса не следует использовать функцию типа `init()`. Если нигде не сказано, что объект должен быть проинициализирован, то программист может забыть об этом, или сделать это дважды (часто с одинаково разрушительными последствиями).

- Для инициализации объекта следует использовать специальную функцию – конструктор, которая будет автоматически вызываться при определении каждого объекта класса или размещении его в памяти с помощью оператора `new`.

- Имя конструктора совпадает с именем класса.

- Таким образом, конструктор инициализирует компонентные данные объекта, выделяет для них память и другие необходимые ресурсы.

- В процессе своей жизни объект может сам создавать в памяти структуры данных вроде динамического массива или линейного списка. Связь самого объекта с такими структурами может осуществляться с помощью атрибута-указателя, хранящего адрес первого байта выделенной структуре области памяти. .

Конструкторы

- Существует три типа конструкторов:
  - конструктор без параметров, используется для создания "пустого" объекта;
  - конструктор с параметрами, используется для инициализации объекта требуемыми значениями;
  - конструктор копирования, используется для создания объекта, аналогичного тому, который уже существует.
  - конструкторы и деструкторы, создаваемые автоматически, не предполагают сколько-нибудь сложного поведения класса и оставляют реализацию особенностей поведения на усмотрение разработчика.
  - Если подобная ситуация разработчика не устраивает, им могут быть определены свои конструкторы и деструкторы, используемые при их наличии.

Конструкторы

- **Основные свойства конструкторов:**
  - Конструктор не возвращает значение, даже типа void. Также нельзя получить указатель на конструктор.
  - Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
  - Конструктор, вызываемый без параметров, называется конструктором по умолчанию.
  - Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию, но их может содержать только один из конструкторов.
  - Если программист не указал ни одного конструктора, компилятор создает его автоматически. В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, по-скольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.
  - Конструкторы не наследуются.
  - Конструкторы нельзя описывать с модификаторами `const`, `virtual` и `static`.
  - Конструкторы глобальных объектов вызываются до вызова функции `main`. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

Конструкторы

- Конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

```
имя_класса имя_объекта [(список параметров)];
//список параметров не должен быть пустым
имя класса (список параметров);
//создается объект без имени, список может быть пустым
имя_класса имя_объекта = выражение;
//создается объект без имени и копируется
```

- Напомним, что прежде, чем использовать классы для создания их экземпляров, мы должны обеспечить видимость их описаний для самой программы:

```
//подключаем часто используемые, но редко изменяемые файлы
#include "stdafx.h"
//подключаем заголовочный файл описания класса
#include "person.h"
```

Конструкторы

- Конструктор без параметров используется для создания "пустого" объекта.

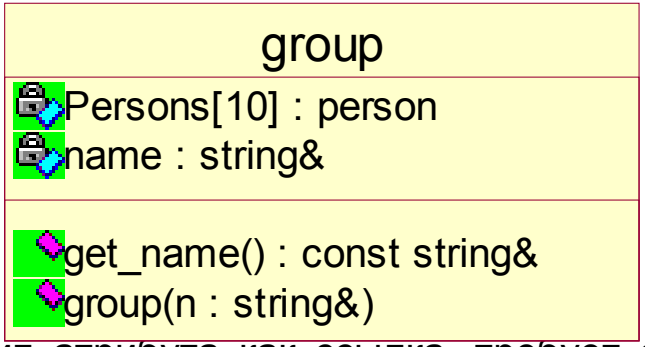
```
//реализация конструктора без параметров
person::person()
{
   name = new char[1];   //выделяем память под один символ
   name[0] = '\0';       //помещаем туда символ конца строки
   year = 0;             //"год рождения" приравниваем к нулю
}
```

- Конструктор с параметрами используется для создания объекта с определенным начальным состоянием.

```
//реализация конструктора с параметрами
person::person(char* n, int y)
{
   name = new char[strlen(n)+1];
   strcpy(name, n);
   year = y;
}
```

Конструкторы

- Существует еще один способ инициализации полей в конструкторе – с помощью списка инициализаторов, расположен-ных после двоеточия между заголовком и телом конструктора. Без этого способа не обойтись при инициализации полей-констант, полей-ссылок и полей-объектов. Проиллюстрируем его на примере объекта, описывающего некоторую группу лиц и хранящего ссылку name на строку с наименованием группы, которая должна быть определена вне области видимости нашего класса group.



- Заметим, что такой тип атрибута как ссылка, требует обязательной инициализации. Соответственно, наличие конструктора без параметров в такой ситуации неприемлемо и будет только вызывать сообщение компилятора об ошибке.

Конструкторы

```
class group
{
   public:
   group(string& n);
   const string& get_name();
   private:
   person Persons[10];
   string& name;
};
```

- Инициализаторы атрибутов перечисляются через запятую. Для каждого атрибута в скобках указывается начальное значение, которое может быть выражением:

```
group::group(string &n) : name(n)
{
}
```

Конструкторы

- Ниже представлен массив указателей на строку Names, значения которых мы определяем с помощью оператора new при создании самих строк. Далее определяется объект класса group, с которым связывается та строка, адрес которой содержит элемент `Names[0]`, для чего вызывается соответствующий конструктор.

```
string* Names[3];
Names[0] = new string("ASU-00-0");
Names[1] = new string("ASU-02-2");
Names[2] = new string("ASU-92-1");
group G(*Names[0]);
cout << G.get_name() << endl;
*Names[0] = *Names[2];
cout << G.get_name() << endl;
cout << *Names[1] << endl;
Names[0] = Names[1];
cout << G.get_name() << endl;
```

Конструкторы

- Последующее изменение значения строки приводит также и к изменению результата, возвращаемого функцией `get_name()`. Однако замещение адреса исходного элемента адресом строки `*Names[1]` не изменяет возвращаемого функцией значения:

```
ASU-00-0
ASU-92-1
ASU-02-2
ASU-92-1
```

- Причиной этого является то, что ссылка name лишь один раз связывается с объектом, на который первоначально указывал элемент `Names[0]`. Значение name не определяется при каждом обращении подобно разыменованию указателя (`*Names[0]`), почему изменение значения самого указателя и не дало предполагаемого результата.

Конструкторы

- *Конструктор копирования* – это специальный вид конструктора, получающий в качестве единственного параметра указатель на объект этого же класса (T — имя класса):

```
T::T(const T&)
{
...
/* тело конструктора */
}
```

- Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего:
  - при описании нового объекта с инициализацией другим объектом;
  - при передаче объекта в функцию по значению;
  - при возврате объекта из функции.

```
//копирующий конструктор, делаем новый объект по подобию исходного
person::person(const person& orig)
{
   name = new char[strlen(orig.name)+1];
   strcpy(name, orig.name);
   year = orig.year;
}
```

Конструкторы

```
//копирующий конструктор, делаем новый объект по подобию исходного
person::person(const person& orig)
{
   name = new char[strlen(orig.name)+1];
   strcpy(name, orig.name);
   year = orig.year;
}
```

- Если программист не указал ни одного конструктора копирования, компилятор создаст его автоматически. Такой конструктор выполняет поэлементное копирование полей. Если класс содержит указатели или ссылки, это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

Конструкторы

```
#include "stdafx.h"
#include "person.h"
void print(person x)   //объект передается по значению
{
   cout << x.get_name() << " / " << x.get_age() << endl;
}
person set_info()      //объект возвращается как значение функции
{
   char s[20]; int y;
   person p;
   cout > s;
   p.set_name(s);
   cout > y;
   p.set_year(y);
   return p;
}
```

Конструкторы

```
void main()
{  //вызов конструктора без параметров для создания объекта a:
   person a;
   //вызов конструктора с параметрами для создания объекта b:
   person b("Abram",1977);
   //вызов конструктора копирования для создания объекта c:
   person c = b;
   //вызов конструктора с параметрами для безымянного объекта
   //и вызов конструктора копирования для определения объекта d:
   person d = person("Gurgen",1942);
   //вызов конструктора без параметров для создания безымянного
   //объекта, размещаемому по адресу, хранящемуся в указателе pp1:
   person* pp1 = new person;
   //вызов конструктора с параметрами для создания безымянного
   //объекта, размещаемого по адресу, хранящемуся в указателе pp2:
   person* pp2 = new person("Ivan",1978);
   //вызов конструктора копирования для возврата значения функцией
   person e = set_info();
   //вызов конструктора копирования для создания объекта-параметра
   //функции:
   print(a); print(b); print(c); print(d);
   print(*pp1); print(*pp2); print(e);
}
```

Конструкторы

- Копирующий конструктор предназначен для создания нового объекта по подобию уже существующего, но это не то же самое, что и копирующий оператор присваивания. Тот должен правильно работать с уже созданным объектом.

Деструкторы

Деструкторы

- Если создание объекта подразумевает выделение каких-либо ресурсов, то таким классам требуется функция, которая освободит ресурсы, выделяемые конструктором. Такая функция называется *деструктором*, она будет автоматически вызвана при уничтожении объекта.

- Деструктор не имеет параметров и возвращаемого значения. Вызов деструктора выполняется неявно, при уничтожении объекта класса.
- Если в классе деструктор не определен явно, то компилятор генерирует деструктор по умолчанию, который просто освобождает память, занятую данными объекта. В тех случаях, когда требуется выполнить освобождение ресурсов, выделенных конструктором, необходимо определить деструктор явно.
- Деструктор вызывается автоматически, когда объект удаляется из памяти:
  - для локальных объектов это происходит при выходе из блока, в котором они объявлены;
  - для глобальных — как часть процедуры выхода из `main`;
  - для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции `delete`.
- Имя деструктора начинается с тильды (`~`), непосредственно за которой следует имя класса.

Деструкторы

- **Свойства деструктора:**
  - не имеет аргументов и возвращаемого значения;
  - не наследуется;
  - не может быть объявлен как `const` или `static` (см. далее);
  - может быть виртуальным (см. далее).
- Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.
- Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически — иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструкторы

- Деструктор для рассматриваемого примера будет выглядеть так:

```
//реализация деструктора для экземпляра класса
person::~person()
{
   delete [] name;
}
```

- Без необходимости явно вызывать деструктор объекта не рекомендуется.

Контрольные вопросы

- 1) Что такое конструктор и для чего он нужен?
- 2) Какие типы конструктора существуют?
- 3) Перечислите основные свойства конструкторов.
- 4) Когда вызывается конструктор?
- 5) Для чего используются конструктор с параметрами и конструктор без параметров?
- 6) Чем отличается конструктор копирования от операции присваивания?
- 7) Что такое деструктор и для чего он нужен?
- 8) Когда вызывается деструктор?
- 9) Какие свойства деструкторов вы знаете9

Методы классов

Методы классов

- Имя *метода-модификатора* принято предварять строкой "`set_`", указывающей проектировщику на то, что этот метод предназначен для изменения значения соответствующего атрибута объекта (например, метод `set_name()` для атрибута `name`). Обычно такие методы не возвращают никаких значений, или же возвращают значение кода, указывающее на успешность выполнения операции.

```
//реализация метода, изменяющего имя человека
void person::set_name(char* value)
{
   //выделяем память под новую строку:
   name = new char[strlen(value)+1];
   //копируем эту строку в область памяти
   strcpy(name, value);
}
```

- Задача этих методов состоит в том, чтобы переложить логику работы с атрибутами на класс. Именно класс должен обеспечивать выполнение правил работы со своими атрибутами.
- Так, приведенный выше код говорит о том, что изменение имени предполагает не побитовое копирование указателей на строку, а копирование самой строки, находящейся по соответствующему адресу. В случае же использования открытого атрибута любой клиент класса вправе делать со значением атрибута все, что угодно.

Методы классов

- Имена *методов-селекторов* предваряют строкой "`get_`", обозначающей получение ими значений атрибутов объекта. Эти методы используют для возвращения значений скрытых атрибутов объекта и вычисления связанных с ними показателей (см. пример выше).

```
//реализация метода, возвращающего имя человека
const char* person::get_name() const
{
   return name;
}
```

Методы классов

- *Методы-итераторы* позволяют выполнять некоторые действия для каждого элемента определенного набора данных. Эти действия должны быть независимы от структуры данных, и задаваться не одним из методов объекта, а произвольной функцией пользователя. Итератор можно реализовать, если эту функцию передавать ему через указатель на функцию.

- Обратим внимание на то, что мы не стали описывать в классе метод, отвечающий за ввод/вывод информации о человеке на консоль.

- Взаимодействие объекта с консолью лучше вынести либо в отдельный класс диалога с пользователем (еще один пример абстракции поведения), либо в независимые от объектов функции. Рассмотрим в этой связи работу с группой объектов.

Методы классов

- Создадим класс `group`, который будет включать в себя массив из 10 экземпляров класса `person` и метод `for_each()` – итератор для этой группы объектов. Отметим сразу, что означенный массив приведен в виде простого атрибута на диаграмме класса group только для наглядности, ибо подобную реализацию принято представлять на диаграмме как атрибут связи между экземплярами классов.
- Ради удобства записи дадим новое имя `FP` типу указателя на функцию, которая не возвращает значения и принимает в качестве аргумента объект типа `person`:

```
typedef void(*FP)(person&);
class group
{
   public:
   void for_each(FP);
   private:
   person Persons[10];
};
```

Методы классов

- Работа итератора заключается в последовательном переборе всех элементов группы и применении к ним любой функции, переданной посредством указателя:

```
void group::for_each(FP fp)
{
   for (int i = 0; i < 10; i++)
   {
        fp(Persons[i]);
   }
}
```

Методы классов

- Теперь опишем в основной программе пару независимых от объектов функций, которые требуется применять к объектам типа `person`, описанным в классе `group`:

```
void show_name(person& argname)
{
   //передача строки с именем в стандартный поток вывода cout:
   cout << argname.get_name() << endl;
}
void set_name(person& argname)
{
   char n[20];
   //получение строки с именем из стандартного поток ввода cin:
   cout << "Enter name: " << endl; cin >> n;
   argname.set_name(n);
}
```

- Такие функции называются свободными подпрограммами. Они исполняют роль операций высокого уровня над объектом или объектами одного или разных классов, и группируются в соответствии с классами, для которых они создаются. Это дает основание называть такие пакеты процедур утилитами класса.

Методы классов

- Нам же остается только создать объект-группу и применить к нему эти функции любым удобным способом:

```
void main()
{
   group G;     //создаем объект-группу из объектов типа person
   G.for_each(set_name);       //определяем для них имена
   cout << "Names:" << endl;
   FP pf = show_name;  //создаем указатель на функцию вывода
   G.for_each(pf);     //применяем этот указатель к группе
}
```

- Свободные подпрограммы часто возникают во время анализа и проектирования на границе объектно-ориентированной системы и ее процедурного интерфейса с внешним миром.

Методы классов

- Утилиты классов употребляются одним из двух способов. Во-первых, утилиты класса могут содержать одну или несколько свободных подпрограмм, и тогда следует просто перечислить логические группы (библиотеки) таких функций - нечленов. Во-вторых, утилита класса может обозначать класс, имеющий только переменные (и операции) класса (в C++ это означало бы класс только со статическими элементами).

```
class util
{
   public:
   static void show_name(person& argname);
   static void set_name(person& argname);
};
```

- В этом случае обращение к методу такого класса при отсутствии экземпляров у последнего должно осуществляться через оператор обращения к области видимости:

```
FP pf = util::show_name;
```

- Связь классов с утилитой может быть отношением использования, но не наследования или агрегирования. В свою очередь, утилита класса может вступать в отношение использования с другими классами и содержать их статические экземпляры, но не может от них наследовать.

Контрольные вопросы

- 1) В чем задача методов-модификаторов?
- 2) Для чего используются методы-селекторы7
- 3) Что позволяют делать методы-итераторы?
- 4) В чем заключается работа итератора?

Статические компоненты класса

Статические компоненты класса

- Спецификатор `static` описывает статические атрибуты и методы класса, которые можно рассматривать как глобальные переменные и функции, доступные в пределах класса.

- Статические атрибуты применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти атрибуты существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

- Особенности статических атрибутов:

  - Память под статический атрибут выделяется один раз при его инициализации, независимо от числа созданных объектов (и даже при их отсутствии), инициализируется с помощью операции доступа к области действия и не учитывается при определении размера объекта с помощью операции `sizeof`.

  - Статические атрибуты доступны как через имя класса, так и через имя объекта.

Статические компоненты класса

- Рассмотрим работу со статическими элементами класса в новой реализации контейнера `group`, пополняемого экземплярами класса `person`. Чтобы организовать такую динамическую структуру данных, каждый новый объект этого класса должен знать адрес созданного ранее объекта, который значение которого хранит статический атрибут `last`. Метод `add()` и метод-итератор `for_each()` объявлены статическими, поскольку производимые ими действия будут относиться к единственной динамической структуре класса `group`.

```
class group
{
   public:
   person* Person;
   group();
   static void add(person&);
   static void group::for_each(void (*)(person&));
   private:
   int number;
   group* next;
   static group* last;
};
```

Статические компоненты класса

- Прежде чем использовать статический атрибут, надо присвоить ему начальное значение (что следует делать в .cpp файле реализации класса, а не где-то ещё):

```
group* group::last = 0;
```

- На статические атрибуты распространяется действие спецификаторов доступа, поэтому статические поля, описанные как `private`, нельзя изменить с помощью операции доступа к области действия. Это делается с помощью статических методов.

- Эти методы могут обращаться непосредственно только к статическим атрибутам и вызывать только другие статические методы класса, потому что им не передается скрытый указатель `this`.

- ***Указатель* this** хранит адрес того объекта, который вызвал функцию и неявно используется внутри метода для ссылок на элементы объекта.

- В явном виде этот указатель применяется в основном для возвращения из метода указателя (`return this;`) или ссылки (`return *this;`) на вызванный объект.

Статические компоненты класса

```
group::group()
{  number = 0;
   next = last;
   last = this;
}
void group::add(person& p)
{  static int z = 0;
   group* x = new group;
   x->Person = &p;
   z++;
   x->number = z;
}
void group::for_each(void (*fp)(person&))
{  group* pz = last;
   while (pz)
   {
   fp(*(pz->Person));
   pz = pz->next;
   }
}
```

Статические компоненты класса

- Обращение к статическим методам производится так же, как к статическим атрибутам, – через имя класса:

```
void main()
{
   person A, B, C;
   A.set_name("Gogi");
   B.set_name("Gagik");
   C.set_name("Ara");
   group::add(A);
   group::add(B);
   group::add(C);
   group::for_each(show_name);
}
```

- … либо, если создан хотя бы один объект класса, через имя объекта.

Контрольные вопросы

- 1) Что описывает спецификатор static?
- 2) Для чего применяются статические атрибуты?
- 3) Что хранит указатель this?

Дружественные функции и классы

Дружественные функции и классы

- Иногда скрытые атрибуты класса требуется открыть для ограниченного доступа извне, что достигается использованием дружественных функций и дружественных классов.

- Дружественные функции применяются для доступа к скрытым атрибутам класса и представляют собой альтернативу методам. Методы, как правило, связаны с атрибутами объекта.

- Правила описания и особенности дружественных функций:
  - Дружественная функция объявляется *внутри класса,* к элементам которого ей нужен доступ, с ключевым словом `friend`. В качестве параметра ей должен передаваться объект или ссылка на объект класса, поскольку указатель `this` данной функции не передается.
  - Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие спецификаторов доступа, место размещения ее объявления в классе безразлично.
  - Одна функция может быть дружественной сразу нескольким классам.

Дружественные функции и классы

- Допустим, мы хотим ограничить возможности по изменению состояния некоторых объектов предметной области таким образом, чтобы делать это могла только специальная функция.

- Это может делаться исходя из того, что реализация последней должна соответствовать определенному набору требований разработчика по организации взаимодействия между пользователем и объектами бизнеслогики системы.

- Такой подход может быть актуальным, когда предполагается возможность создания новой надстройки к базовой архитектуре объектов системы и стоит задача защититься от некорректных изменений объектов предметной области, которые новый модуль предполагает получать из общей базы данных.

Дружественные функции и классы

- С этой целью мы можем объявить методы-модификаторы класса person как закрытые (или защищенные, если в будущем предполагается наследование), а необходимую для обеспечения интерфейса с пользователем функцию `set_name()` как дружественную:

```
class person
{
   friend void set_name(person& argname);
   private:
   //...
}
```

Дружественные функции и классы

- Если все методы какого-либо класса должны иметь доступ к скрытым атрибутам другого, весь класс объявляется дружественным с помощью ключевого слова friend. Так, мы можем объявить класс dialog в качестве дружественного для person, если хотим обеспечить изменение внутреннего состояния объектов этого класса средствами спроектированного диалога.

```
class person
{
    friend class dialog;
    //...
}
```

- Естественно, перечень друзей не должен ограничиваться одним классом или функцией, а определяется лишь соображениями разумной достаточности. Использования дружественных функций нужно по возможности избегать, поскольку они нарушают принцип инкапсуляции и, таким образом, затрудняют отладку и модификацию программы.

Перегрузка операций

Перегрузка операций

- C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Эта дает возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в C++, за исключением следующих:

```
.*        ?:        ::        #        ##        sizeof
```

- Перегрузка операций осуществляется с помощью методов специального вида (*функций-операций)* и подчиняется следующим правилам:
  - при перегрузке операций сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
  - для стандартных типов данных переопределять операции нельзя;
  - функции-операции не могут иметь аргументов по умолчанию;
  - функции-операции наследуются (за исключением =);
  - функции-операции не могут определяться как `static`.

Перегрузка операций

- Функция-операция должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс.

- Функция-операция содержит ключевое слово `operator`, за которым следует знак переопределяемой операции:

```
тип operator операция(список параметров){ тело функции }
```

Перегрузка унарных операций

Перегрузка унарных операций

- Унарная (т.е. имеющая) функция-операция, определяемая *внутри класса,* должна быть пред-ставлена с помощью нестатического метода *без параметров*, при этом операндом является вызвавший ее объект.

```
class person
{
   public:
   …
   person& operator++();
};
person& person::operator++()
{
   ++weight; return *this;
}
person x;
++x;
cout << x.get_weight();
```

- Запись `++x;` интерпретируется компилятором как вызов компонентной функции `x.++()`.

Перегрузка унарных операций

- Если функция определяется *вне класса,* она должна иметь *один* параметр типа класса.

```
class person
{
   …
   friend person& operator--(person &rhs);
};
person& operator--(person &rhs)
{
   --rhs.weight; return rhs;
}
person x;
--X;
cout << x.get_age();
```

- В этом случае запись `--x;` интерпретируется компилятором как вызов глобальной функции `--(x)`.

Перегрузка унарных операций

- Операции постфиксного инкремента и декремента должны иметь фиктивный параметр типа `int`. Он используется только для того, чтобы отличить их от префиксной формы.

```
class person
{
   …
   person& operator ++(int);
};
```

- Если не описывать функцию внутри класса как дружественную, нужно учитывать доступность изменяемых полей. В данном случае поле `weight` недоступно извне, так как описано со спецификатором `private`, поэтому для его изменения требуется использование метода-модификатора `set_weight()`. В таком случае можно перегрузить операцию инкремента с помощью обычной функции, описанной вне класса:

```
person& operator ++(person& p, int)
{
   double w = p.get_weight(); w++; p.set_weight(w);
   return p;
}
```

Перегрузка бинарных операций

Перегрузка бинарных операций

- Бинарная функция-операция, определяемая *внутри класса,* должна быть представлена с помощью нестатического метода с параметрами, при этом вызвавший ее объект считается первым операндом:

```
class person
{
   …
   bool operator<(person& rhs) const;
   friend bool operator>(const person& p1, const person& p2);
};

bool person::operator<(person& rhs) const
{
   return (weight < rhs.weight);
}
```

Перегрузка бинарных операций

- Запись `X<Y`, где `X` и `Y` объекты класса `person` будет интерпретироваться компилятором как вызов компонентной функции `X.<(Y)`.

- Если функция определяется *вне класса,* она должна иметь два параметра типа класса.

```
bool operator>(const person& p1, const person& p2)
{
   return (p1.weight > p2.weight);
}
```

- Запись `X>Y`, где `X` и `Y` объекты класса `person` будет интерпретироваться компилятором как вызов глобальной функции `>(X,Y)`.

Перегрузка операции присваивания

Перегрузка операции присваивания

- Операция присваивания определена в любом классе по умолчанию как поэлементное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. Если класс содержит поля, память под которые выделяется динамически, необходимо определить собственную операцию присваивания. Чтобы сохранить семантику присваивания, операция-функция должна возвращать ссылку на объект, для которого она вызвана, и принимать в качестве параметра единственный аргумент – ссылку на присваиваемый объект.

```
person& person::operator=(person& rhs)
{
   //проверка на самоприсваивание:
   if (&rhs == this) return *this;
   name = new char[strlen(rhs.name)+1];
   strcpy(name, rhs.name);
   year = rhs.year;
   weight = rhs.weight;
   return *this;
}
```

Перегрузка операции присваивания

- Возврат функцией ссылки на объект делает возможной цепочку операций присваивания:

```
person A("Vovan", 1960, 120), B, C;
C = B = A;
```

- Операцию присваивания можно определять только как метод класса. Она не наследуется.

Перегрузка операции вызова функции

Перегрузка операции вызова функции

- Класс, в котором определена операция вызова функции, называется функциональным. От такого класса не требуется наличия других полей и методов. Объект такого класса называется объект-функция.

```
class greater
{
   public:
   int operator()(int a, int b)
   {
        return a > b;
   }
};
void main()
{
   greater x;
   cout << x(1, 5);              // Результат - 0
   cout << greater()(5, 1);      // Результат - 1
}
```

Перегрузка операции вызова функции

- Поскольку в классе `greater` определена операция вызова функции с двумя параметрами, выражение `x(1,5)` является допустимым и может быть записать в виде `x.operator(1,5)`. Как видно из примера, объект функционального класса используется так, как если бы он был функцией.

- Во втором операторе вывода выражение `greater()` используется для вызова конструктора по умолчанию класса `greater`. Результатом выполнения этого выражения является объект класса `greater`. Далее, как и в предыдущем случае, для этого объекта вызывается функция с двумя аргументами, записанными в круглых скобках.

- Операцию `()` (а также операцию `[]`) можно определять только как метод класса. Можно определить перегруженные операции вызова функции с различным количеством аргументов.

Контрольные вопросы

- 1) Для чего нужна перегрузка операций?
- 2) Какие существуют правила работы с перегрузками?
- 3) Как описывается перегрузка внутри класса и вне его?
- 4) Как операция присваивания определена по умолчанию?
- 5) Какие операции не наследуются?
- 6) Что такое объект-функция?

Идентичность объектов

Идентичность объектов

- *Идентичность* – это такое свойство объекта, которое отличает его от всех других объектов. Однако следует отличать идентичность от адресуемости объекта (конкретного адреса объекта в памяти).

```
struct point
{
   //определяем точку на плоскости с координатами x и y
   int x, y;
   point() : x(0), y(0) {}
   point(int xv, int yv) : x(xv), y(yv) {}
};
class item
{
   //экранный объект
   …
   public:
   item(const point& location);
   item();
   //…
};
```

Идентичность объектов

```
//объявление объектов класса item
item A;
item* B = new item(point(75, 75));
item* C = new item(point(100, 100));
item* D = 0;
item& E = A;
```

- При выполнении этих операторов возникают пять имен, но только три разных объекта. Конкретно, в памяти будут отведены четыре ячейки под имена `A`, `B`, `C` и `D`. При этом `A` будет именем объекта, `B`, `C` и `D` будут указателями, а `E` является другим именем для объекта `A`. Кроме того, лишь `B` и `C` будут на самом деле указывать на объекты класса `item`. У объектов, на которые указывают `B` и `C` нет имен, хотя на них можно ссылаться, "разыменовывая" соответствующие указатели: например, `*B`. Поэтому можно сказать, что B указывает на отдельный объект класса `item`, на имя которого можно косвенно ссылаться через `*B`. Уникальная идентичность (но не обязательно имя) каждого объекта сохраняется на все время его существования, даже если его внутреннее состояние изменилось.

Обращение к элементам классов

Обращение к элементам классов

- К объектам программы можно обращаться не только по их имени, но и путём разыменования указателей. Атрибуты и методы классов также подразумевают возможность подобного обращения к себе.
- К элементам классов можно обращаться с помощью указателей. Для этого определены операции `.*` и `->*`. Указатели на поля и методы класса определяются по-разному.
- Формат указателя на поле класса:

```
тип_данных(имя_класса::*имя_указателя);
```

- В определение указателя можно включить его инициализацию в форме:

```
&имя_класса::*имя_поля;    // Поле должно быть public
```

- Если бы поле `age` класса `person` (см. выше) было объявлено как `public`, определение указателя на него имело бы вид:

```
int (person::*page) = person::age;
person a("Vasia",15);
person* pa=&a;
cout << a.*page;    // Обращение через операцию .*
cout << pa->*page; // Обращение через операцию ->*
```

Обращение к элементам классов

- Указатели на поля классов не являются обычными указателями т. к. при присваивании им значений они не ссылаются на конкретный адрес памяти, поскольку память выделяется не под классы, а под объекты классов.

- Формат указателя на метод класса:

```
возвр_тип (имя_класса::*имя_указателя)(параметры);
```

- Например, описание указателя на метод класса `person`

```
int person::get_age() {return age;}
```

- а также на другие методы этого класса с такой же сигнатурой будет иметь вид:

```
int (person::*pget)();//указатель на get_age()
```

- Такой указатель можно задавать в качестве параметра функции. Это дает возможность передавать в функцию имя метода:

```
void fun(int (person:: *pget)())
{
   (*this.*pget)(); // Вызов функции через операцию .*
   (this->*pget)(); // Вызов функции через операцию ->*
}
```

Обращение к элементам классов

- Можно настроить указатель на конкретный метод с помощью операции взятия адреса:

```
//присваивание значения указателю:
pget = & person::get_age;
person a,*pa;
pa = new person;
// Вызов функции через операцию .* :
int a_age= (a.*pget)();
// Вызов функции через операцию ->* :
int pa_age = (p->*pget)();
```

- Правила использования указателей на методы классов:
  - Указателю на метод можно присваивать только адреса методов, имеющих соответствующий заголовок.
  - Нельзя определить указатель на статический метод класса.
  - Нельзя преобразовать указатель на метод в указатель на обычную функцию, не являющуюся элементом класса.

- Однако в отличие от указателя на переменную или обычную функцию, указатель на метод не ссылается на определенный адрес памяти. Он больше похож на индекс в массиве, поскольку задает смещение. Конкретный адрес в памяти получается путем сочетания указателя на метод с указателем на определенный объект.

Контрольные вопросы

- 1) Что такое идентичность?
- 2) Какими способами можно обращаться к объектам?
- 3) Какие операции определены для обращения к элементам класса?
- 4) Какие существуют правила использования указателей на методы классов?

Отношения между классами и объектами

Отношения между классами и объектами

- Отношения двух любых объектов основываются на предположениях, которыми один объект обладает относительно другого: об операциях, которые можно выполнять над объектом и об его ожидаемом поведении.

- Если объект предоставляет свои ресурсы другим объектам, то он называется сервером, а если он их потребляет – клиентом.

- **Связь** – это специфическое сопоставление, через которое объект-клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.

- Объект может являться сервером по отношению к одним объектам и клиентом по отношению к другим.

Отношения между классами и объектами

- Поведение объекта характеризуется услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами.

- Внешнее проявление объекта рассматривается с точки зрения его взаимодействия с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство. Каждая операция, предусмотренная этим взаимодействием, однозначно определяется ее формальными параметрами и типом возвращаемого значения.

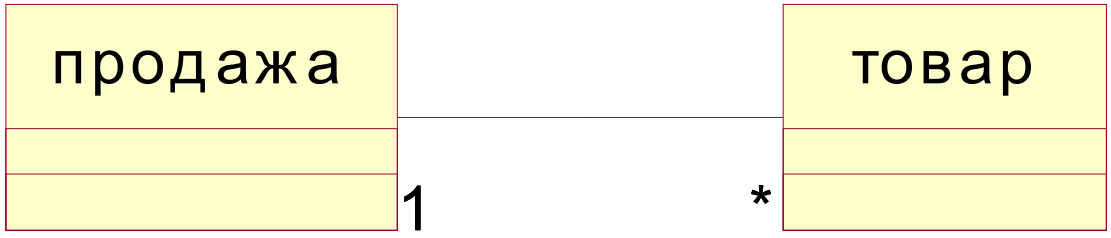
- Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется *протоколом*.

Отношения между классами и объектами

- Связь дает классу возможность узнавать об атрибутах, операциях и связях другого класса. В нотации языка UML взаимодействие между классами отражают связывающими их линиями.

- Между объектами могут существовать следующие виды отношений:

- **Ассоциация** – это смысловая связь, которая не имеет направления и не объясняет, как классы общаются друг с другом. Однако именно это требуется на ранней стадии анализа, поэтому мы фиксируем только участников, их роли и мощность отношения. На диаграммах UML эту связь отображают обыкновенной линией, связывающей классы



Отношения между классами и объектами

- Ассоциация является наиболее общим и неопределенным видом отношений экземпляров классов. Обычно аналитик констатирует наличие ассоциации и, постепенно уточняя объект, превращает ее в какую-то более специализированную связь.

- Ассоциации могут быть двунаправленными или однонаправленными. На языке UML двунаправленные ассоциации рисуют в виде простой линии без стрелок или со стрелками с обеих ее сторон. На однонаправленной ассоциации изображают только одну стрелку, показывающую ее направление.

Отношения между классами и объектами

- Рассмотрим в качестве примера торговую точку. В этом примере можно выделить две абстракции – товары и продажи. Класс `product` – это то, что мы продали в некоторой сделке, а класс `sale` – сама сделка, в которой продано несколько товаров. Ассоциация является двунаправленной, т.к. зная товар, можно получить информацию о сделке, в которой он был продан, а, имея информацию о сделке, можно узнать, что было продано.

- Исходя из вышеозначенной диаграммы, для классического C++ мы получим два заголовочных файла с определениями используемых классов.

Отношения между классами и объектами

```
//файл product.h
#include "sale.h"
class product
{
   private:
   sale *LastSale;
};
//файл sale.h
#include "product.h"
class sale
{
   private:
   //указатель на массив, содержащий совокупность товаров,
   //проданных в сделке
   product* Products
};
```

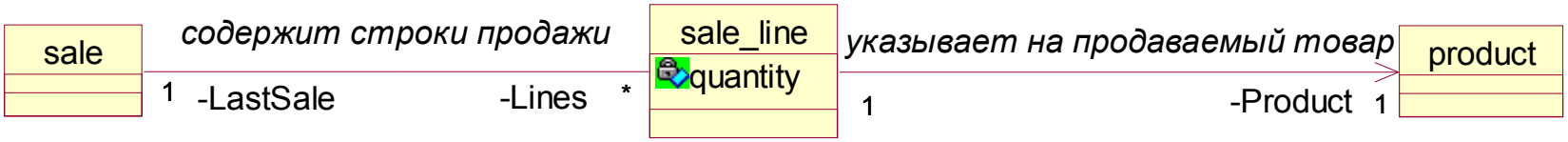
- Это ассоциация представляет собой связь "один-ко-многим": каждый экземпляр товара относится только к одной последней продаже, в то время как каждый экземпляр `sale` может указывать на совокупность товаров.

Отношения между классами и объектами

- *Мощность отношения* показывает, сколько экземпляров одного класса взаимодействуют с помощью этой связи с одним экземпляром другого класса в данный момент времени. Количество экземпляров указывается на линии связи со стороны каждого класса–участника цифрой или символом "*", который имеет значение "много". Это говорит о том, что число экземпляров такого класса неизвестно и может быть любым.

- Чтобы проиллюстрировать отношение ассоциации, рассмотрим приведенный выше пример подробнее. Исходный пример показывал нам только то, что продажа каким-то образом связана с продаваемыми товарами. Реально же во время продажи могут быть реализованы одинаковые товары, которые учитывают не по конкретным экземплярам, а просто по количеству.

Отношения между классами и объектами



- Объект класса `sale` ("продажа") содержит множество строк (объекты `sale_line`), каждая из которых указывает на описываемый ей товар типа `product` и на количество этого товара – `quantity`. Между строкой продажи и товаром мы имеем однонаправленную ассоциацию с мощностью связи "один-к-одному", поскольку классу `product`, описывающему товар, ничего не требуется знать о строке продажи, а та описывает потребность только в одном товаре.

Отношения между классами и объектами

- Если все сообщения отправляются только одним классом и прини-маются только другим классом, а не наоборот, между этими классами имеет место однонаправленная связь. Если хотя бы одно сообщение отправляется в обратную сторону, ассоциация должна быть двунаправленной.

- Связь вида "многие-ко-многим" обычно присутствует в качестве ассоциации только на ранних стадиях анализа.

- Допускается также указывать рядом с классами конкретные значения количества их экземпляров, если они заранее известны. Это непосредственно влияет на создаваемый по диаграммам код языка.

- Связи можно уточнить с помощью имен связей или ролевых имен. *Имя связи* – это обычно глагол или глагольная фраза, описывающая, зачем она нужна. В рассмотренном примере между классом `sale` (продажа) и классом `sale_line` (строка продажи) существует ассоциация. В связи с этим возникает вопрос понимания, чем является объект класса `sale` по отношению к объекту класса `sale_line`? Для того чтобы указать на это, ассоциацию можно обозначить "содержит строки продажи", т. е. класс `sale` "содержит" объекты класса `sale_line`.

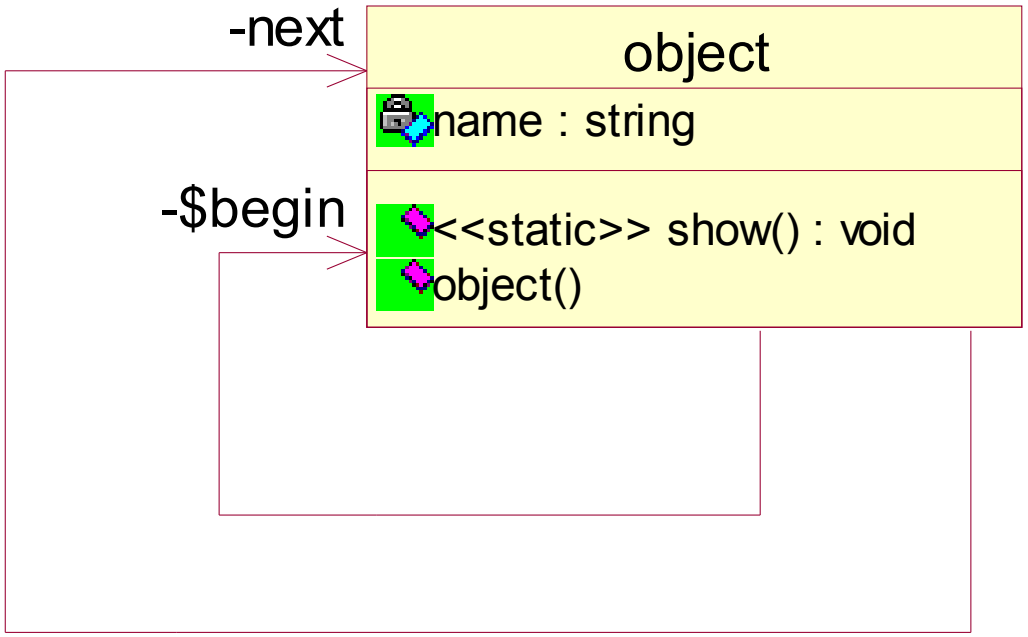
Отношения между классами и объектами

- Имена у связей определять необязательно. На создаваемый по диаграммам код влияния они не оказывают. Обычно это делают, если причина создания связи не очевидна. Кроме того, проектировщику легче различать именованные связи, чем угадывать смысл отношений между классами и их экземплярами. Имя показывают около линии соответствующей связи.

- Для описания того, зачем нужны связи в ассоциации, применяют *ролевые имена*. Возвращаясь к примеру с торговой точкой, можно сказать, что объекты класса `sale_line` по отношению к объекту `sale` играют роль строк – `Lines` (Строки). Ролевые имена – это обычно имена существительные, их показывают на диаграмме рядом с классом, играющим соответствующую роль. Как правило, пользуются или ролевым именем, или именем связи, но не обоими сразу. Как и имена связей, ролевые имена не обязательны, их дают, только если цель связи не очевидна. На готовый код имена ролей не влияют, однако делают текст программы более удобным для восприятия, поскольку используются в качестве имен атрибутов соответствующих классов.

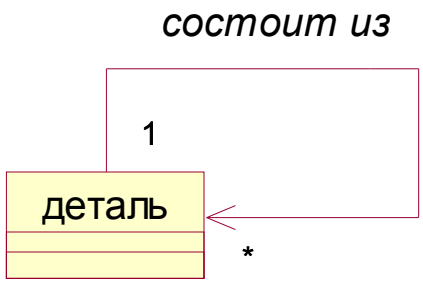
Отношения между классами и объектами

- Ассоциации могут быть рефлексивными. *Рефлексивная ассоциация* предполагает, что один экземпляр класса взаимодействует с другими экземплярами этого же класса. Примером может служить рассмотренный выше контейнерный класс group, экземпляры которого связаны друг с другом:

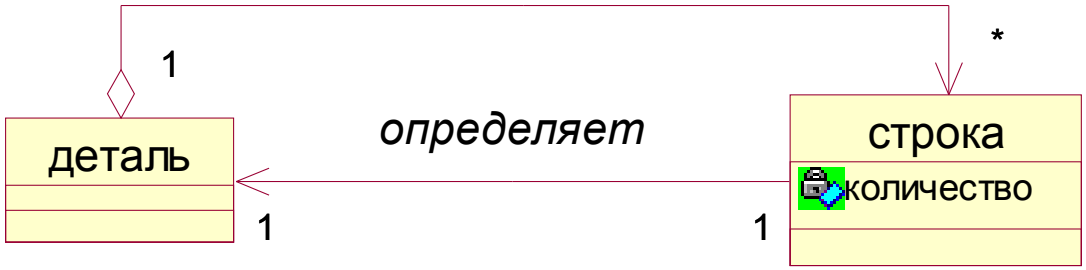


Отношения между классами и объектами

- На ранних этапах проектирования ассоциация обычно говорит лишь о наличии связи, реализация которой будет определена позднее. Так, мы можем вначале сказать только то, что детали могут собираться из других деталей:

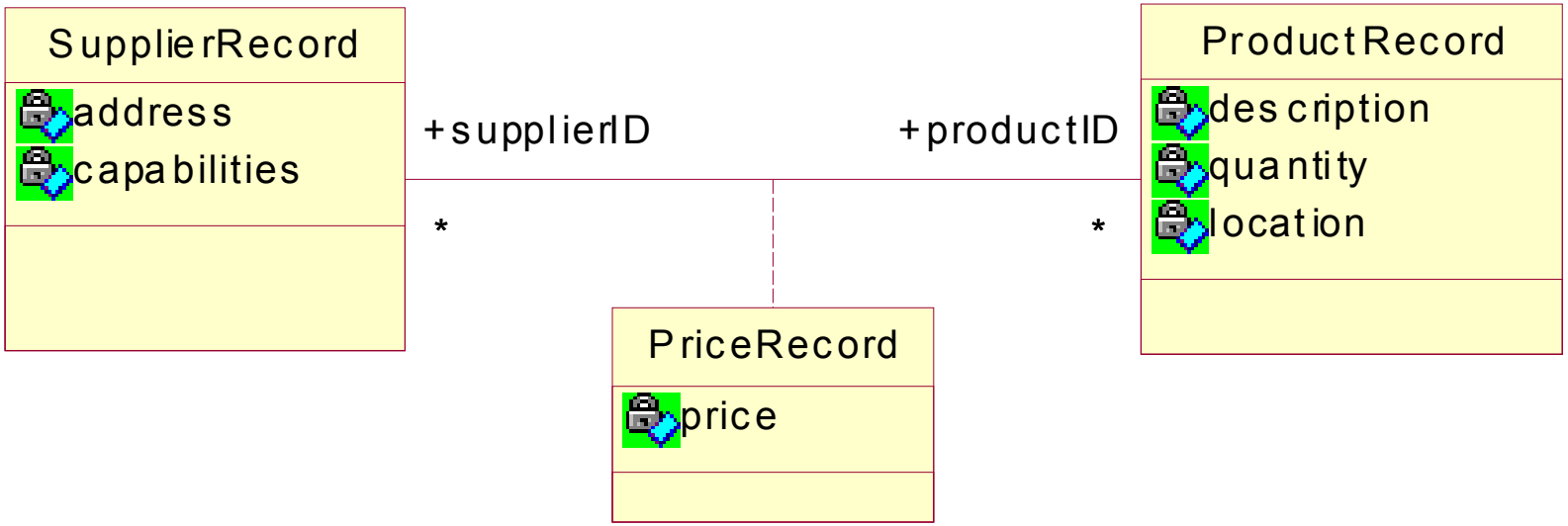


- … а в дальнейшем уточнить это отношение:



Отношения между классами и объектами

- Отношение ассоциации может также иметь структуру и поведение. Это происходит в том случае, когда информация обращена к связи между объектами, а не к самому объекту. Такой вариант называется *ассоциацией с примечанием* и обозначается пунктирной линией, соединяющий связь между двумя классами и ассоциированный класс. В этом случае мы имеем дело с информацией, относящейся к паре объектов, а не к каждому отдельному объекту.



Отношения между классами и объектами

- В случае, когда может возникнуть необходимость в подобном отношении. Нередко встречается ситуация, когда одинаковый товар заказчику поступает от разных поставщиков, соответственно, и цены на него могут отличаться. С одной стороны, по одному товару можно выйти на всех его поставщиков, с другой – от поставщика определить все поставляемые им товары. Однако по такой связи нельзя однозначно определить цену товара. Однозначно ее определяет пара ключей – код поставщика и код товара. Поскольку сведения о цене определяются их связью, запись о цене выражают в виде ассоциированного с ее линией класса.

- В нашем случае объявления классов `SupplierRecord` (сведения о поставщиках) и `ProductRecord` (сведения о продуктах) не будут содержать никаких ссылок друг на друга и, в то же время, оба будут включать атрибут, содержащий указатели на объекты `PriceRecord`, тем самым, выражая отношение "один-ко-многим".

Отношения между классами и объектами

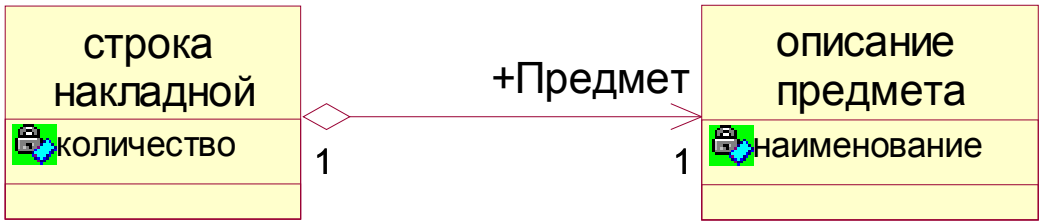
```
//сведения о поставщиках:
class SupplierRecord
{
   private:
   …
   PriceRecord* the_PriceRecord;
   …
}
```

- В свою очередь класс `PriceRecord` представляет собой составной ключ, однозначно определяющий связь цены, выражаемой атрибутом `price`, с товаром и его поставщиком.

```
//цена на конкретный продукт от конкретного поставщика:
class PriceRecord
{
   private:
   double price;
   ProductRecord *productID;
   SupplierRecord *supplierID;
   …
}
```

Отношения между классами и объектами

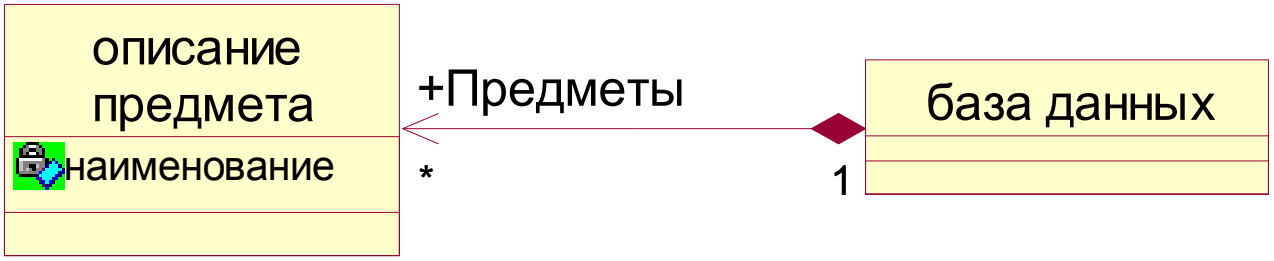
- *Агрегация* описывает отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата) мы можем придти к его частям (атрибутам). В этом смысле агрегация – специализированный частный случай ассоциации. Объект, являющийся атрибутом другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения (см. "Использование"). Агрегацию отображают в виде линии с ромбиком у класса, являющегося целым.



- Приведенный выше вариант агрегации (с незакрашенным ромбиком) называется также *агрегацией по ссылке*.

Отношения между классами и объектами

- В дополнение к простой агрегации UML вводит более сильную разновидность агрегации, называемую *композицией* или *агрегацией по значению*. Согласно композиции объект-часть может принадлежать только единственному целому, и, кроме того, зачастую жизненный цикл час-тей совпадает с циклом целого: они живут и умирают вместе с ним. Любое удаление целого распространяется на его части. Такой вид агрегации отображается с помощью закрашенного ромбика со стороны целого



- Отношение агрегации между классами имеет непосредственное отношение к агрегации между их экземплярами. Агрегация по ссылке означает включение в класс агрегата атрибута-указателя или атрибута-ссылки на экземпляр связанного с ним класса. В этом случае уничтожение целого (агрегата) не приводит к уничтожению его частей и эти части, таким образом, могут принадлежать различным объектам. Агрегация по значению предполагает включение в класс агрегата собственно объекта-части.

Отношения между классами и объектами

- Рассмотрим пример из области торговли. Товары принимаются согласно документам строгой отчетности:

Счет АП-0001684 от 23 Июня 2004 г.

ПОСТАВЩИК: ООО "РОСЭК-Пермь"
614058, Пермская область, г. Пермь, ул. Деревообделочная, дом 3

ИНН 5903042804 КПП 590301001
р/с 40702810149490151935 в Дзержинском ОСБ №6984 Западно-Уральского Банка СБ РФ г. Перми к/с 30101810900000000603, БИК 045773603

Web: www.roselektro.ru, e-mail: perm@roselektro.ru

Тел/факс: (3422) 38-54-64, 38-54-60
ПЛАТЕЛЬЩИК: Пермский государственный технический университет
ИНН

№	Артикул	Товар	Ед.	Кол.	Цена	Сумма
1	2	3	4	5	6	7
1	leg 30038	Кабель-канал пластик с/крышкой 50x100 L=2м (упак. 16 шт.)	м	20	275,00	5500,00
2	leg 30547	Суппорт Mosaic д/к-к ...x100 и крышки 75мм 6М (упак. 10 шт.)	шт.	10	110,50	1105,00
3	leg 74131	Розетка Mosaic -2М 2Р+Е нем.стд. (упак. 10 шт.)	шт.	30	92,00	2760,00
4	sch 11203	Авт. выкл. ВА63 1п 16А С	шт.	2	96,00	192,00

Итого: **9 557,00 руб**
В том числе НДС 18% **1 457,85 руб**

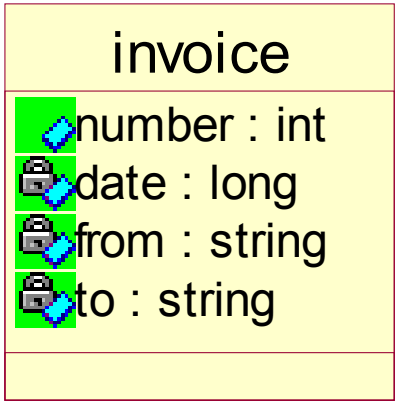
Всего к оплате **9 557,00 руб**

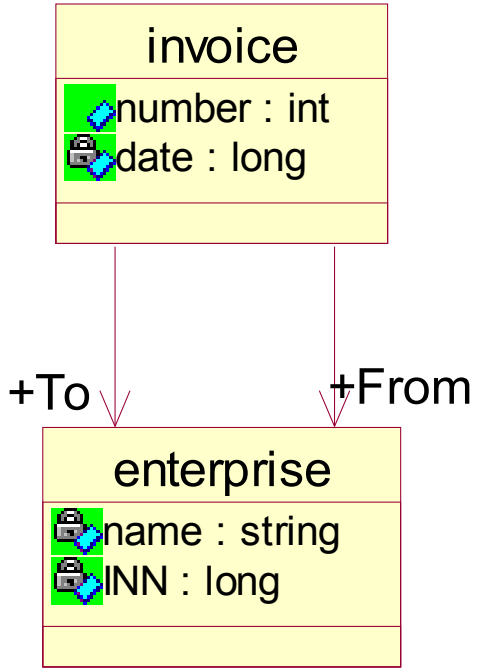
Всего наименований 4, на сумму 9557 руб
Сумма: Девять тысяч пятьсот пятьдесят семь рублей 00 копеек.
Руководитель _____***/Закурдаев И.А./***

Бухгалтер _____ ***/Тагирова З.З./***

Отношения между классами и объектами

- Эти документы необходимо создавать, обрабатывать и хранить. Важен не внешний вид, а логическое представление.
- Начнем с документа «счет». В исходном документе имеется информация о номере счета, дате выписки, поставщике и плательщике и т.д. Представим документ «счет» как класс invonce. Номер счета и дата являются его непосредственными свойствами.
- Поставщик с плательщиком в этом плане не столь однозначны: если нет необходимости в отдельном учете предприятий-клиентов, то можно обойтись простыми строками для этих атрибутов (`from` и `to`), в противном случае лучше выделить понятие "предприятие" в отдельный класс (`enterprise`), с которым и связать класс, описывающий счет.





Отношения между классами и объектами

- Для предприятия, выписывающего счет, вряд ли будет представлять ценность поле поставщика, поэтому данный атрибут вообще можно исключить из логического представления документа, отнеся собственные реквизиты лишь к внешнему виду счета.

- Текст счета формулируется из множества строк. Каждая строка содержит номер строки, артикул, наименование товара, единицу измерения, количество, цену и сумму. Номер строки не определяет никаких свойств строки, поскольку говорит только о порядке следования строки.

- После номера строки находится артикул – код товара, и далее его наименование. Если предполагается хранить где-то информацию о товаре и выбирать ее, а не заполнять каждый раз текстовое поле, то стоит вынести понятие товара в отдельный класс. К нему отнесем атрибуты "артикул", "наименование товара", "цена" и "единица измерения".

- Атрибут "количество" выражает мощность множества товаров данного наименования, а не описывает свойство единицы товара. Поэтому введем класс `line`, который и будет содержать указатель на объект, связанный с поставляемым товаром (`item`) и количество поставляемых товаров. Промежуточная сумма по строке представляет собой произведение цены товара на его количество, а итог складывается из промежуточных сумм.

Отношения между классами и объектами

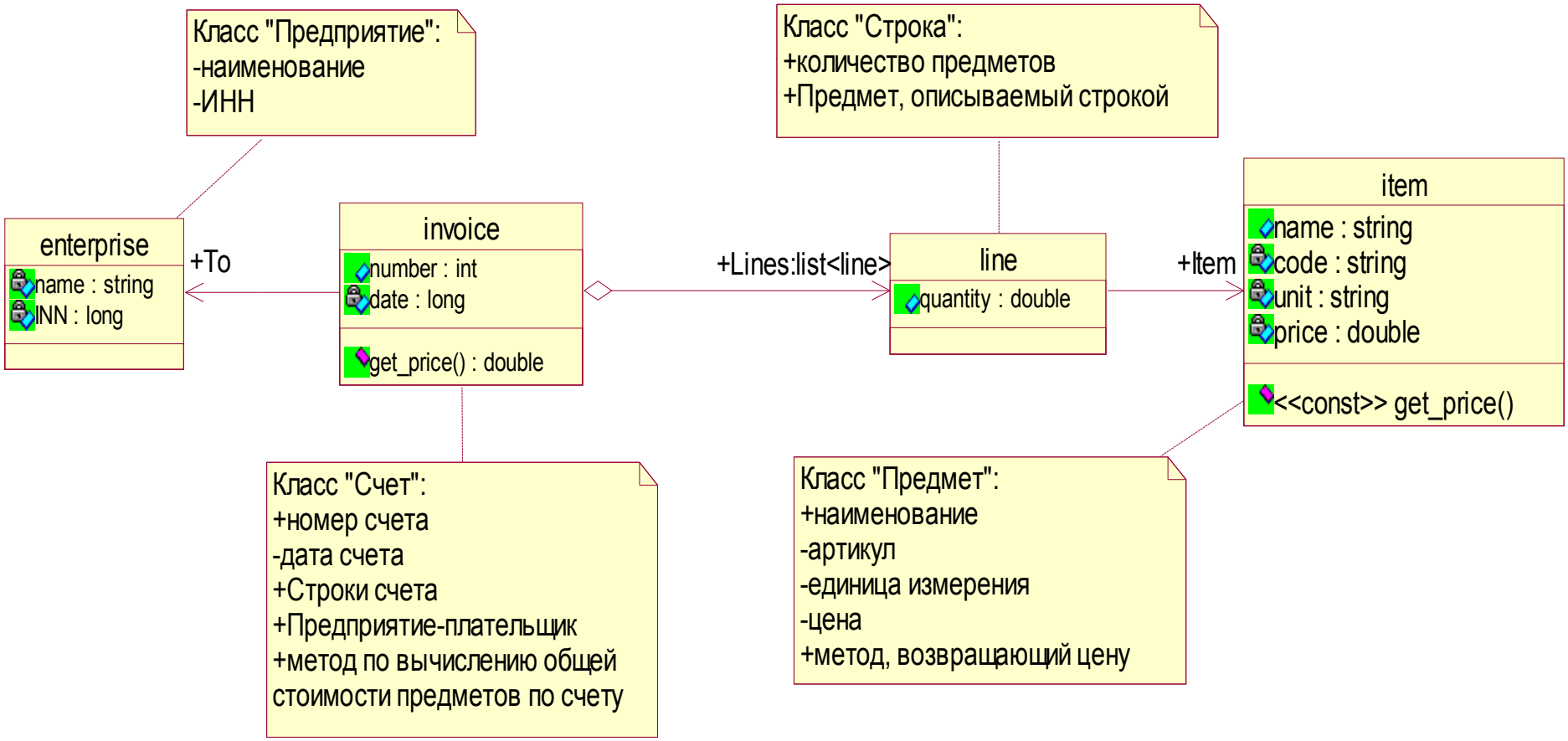
- В результате мы получим следующую номенклатуру классов:

`invoice`	свойства счета, счет содержит множество строк
line	строка в счете, указывает количество некоторых предметов
item	свойства предмета, на который выписывается документ
enterprise	реквизиты предприятия, на которое выписывается счет

- Заметим, что каждому счету соответствует множество строк, причем заранее их количество неизвестно. По этой причине следует использовать специальный контейнерный класс вместо объявления статического массива из объектов.
- Мы остановимся на контейнере `list`, реализующем связный список.

Отношения между классами и объектами

- Отношения между экземплярами рассмотренных классов будет выражены следующей диаграммой:



Отношения между классами и объектами

- Код, полученный на её основе, будет выглядеть так:
-

```
//Класс "Предприятие"
class enterprise
{  private:
   //наименование предприятия
   string name;
   //ИНН предприятия
   long INN;
};
//Класс "Предмет"
class item
{  public:
   //наименование
   string name;
   //метод, возвращающий цену предмета
   const double get_price() const;
   private:
   //артикул
   string code;
   //единица измерения
   string unit;
   //цена
   double price;
};
```

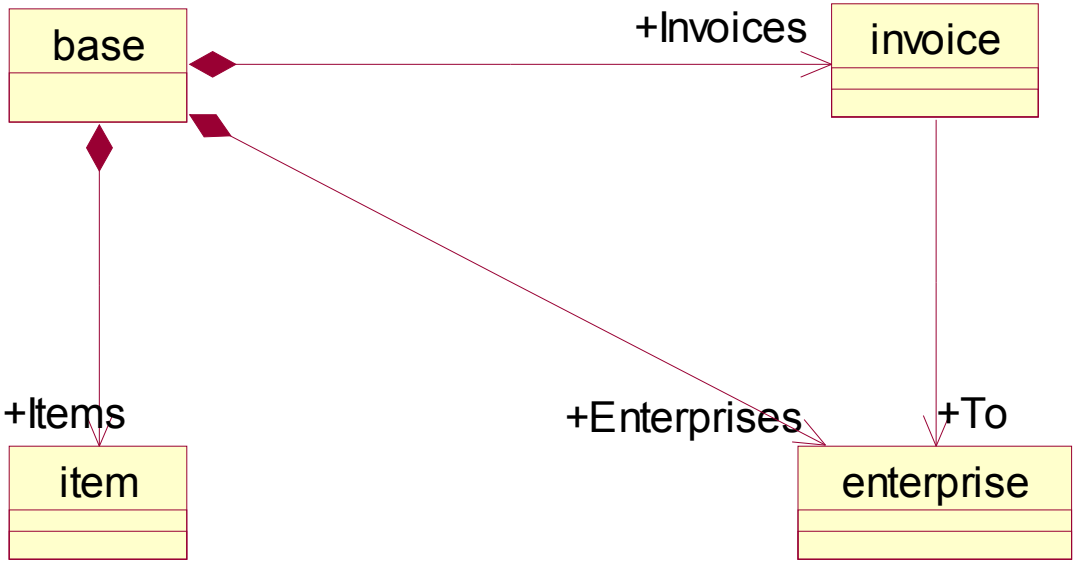
Отношения между классами и объектами

```
//Класс "Строка"
class line
{
   public:
   //количество
   double quantity;
   //Предмет, на который указывает строка
   item* Item;
};
//Класс "Счет"
class invoice
{
   public:
   //номер счета
   int number;
   //Строки счета
   list<line> Lines;
   //Предприятие-плательщик
   enterprise* To;
   //подсчет стоимости счета
   double get_price();
   private:
   //дата
   long date;
};
```

Отношения между классами и объектами

- Из приведенного кода видно, что ассоциативные связи реализованы посредством указателей на экземпляры классов (например, атрибут `Item` класса `line` хранит адрес объекта типа `item`). В классе `invoice` мы имеем агрегацию экземпляров класса "Строка" по значению посредством атрибута `Lines`, представляющего собой контейнер "список" для экземпляров класса `line`.

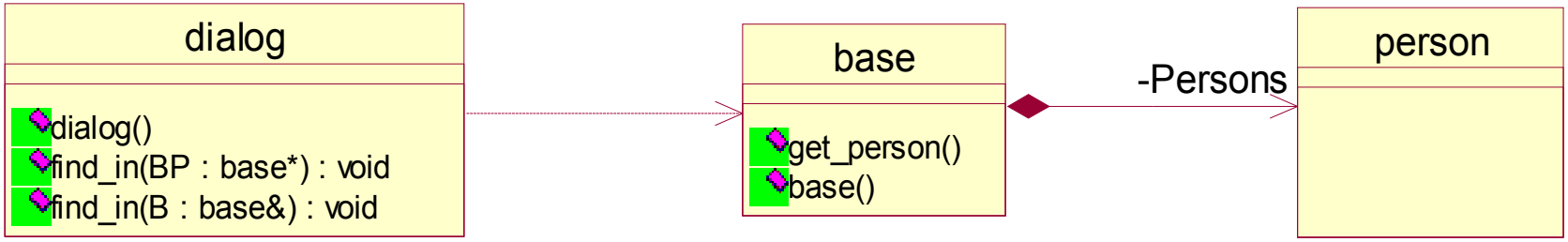
- Логично будет предположить, что те экземпляры классов, с которыми другие связаны ассоциацией, тоже должны где-то храниться. По этой причине будет разумно добавить еще одну абстракцию – класс `base`, описывающий базу данных, которая будет хранить набор ключевых объектов предметной области. Здесь мы специально делаем акцент на физическом включении таким объектов других:



Отношения между классами и объектами

- **Отношение использования** между классами соответствует равноправной связи между их экземплярами. Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера). В UML эти отношения изображают в виде пунктирной линии со стрелкой.

- Когда дело доходит до реализации системы, необходимо обеспечить *видимость* связанных объектов. Есть четыре способа обеспечить видимость:
  - сервер глобален по отношению к клиенту;
  - сервер (или указатель на него) передан клиенту в качестве параметра операции;
  - сервер является частью клиента;
  - сервер локально порождается клиентом в ходе выполнения какой-либо операции.



Отношения между классами и объектами

- Приведенный пример показывает обращение клиента, объекта некоего диалогового класса (`dialog`), к серверу: объекту класса base, играющему роль базы данных, хранящей объекты класса dialog. Здесь использование объекта класса base подразумевает вызов его метода, в частности `get_person()`, для поиска с его помощью объекта класса person по значению его атрибута name.

```
class dialog
{
   public:
   dialog();
   void find_in(base& B);
   void find_in(base* BP);
};
dialog::dialog()
{
   string x; person* pp;
   while (x != "0")
   {
        cout << "Search name: "; cin >> x;
        pp = DB.get_person(x); //DB – глобальный объект.
        if (pp) cout << "Address of object is " << pp << endl;
        else cout << "Object not found!" << endl;
   }
}
```

Отношения между классами и объектами

- В данном случае создаваемый конструктором объект диалога обращается к объекту DB, который является глобальным в программе. Если же мы пожелаем производить поиск в какой-то другой базе данных, можно добавить такую возможность путем передачи классу диалога ссылки или указателя на один из таких объектов:

```
void dialog::find_in(base &B)          //передача сервера по ссылке
{
   string x; cout << "Search name: "; cin >> x;
   cout << B.get_person(x);
}
void dialog::find_in(base *BP)      //передача сервера по указателю
{
   string x; cout << "Search name: "; cin >> x;
   cout << BP->get_person(x);
}
```

- Любой из этих вариантов соответствует приведенной выше диаграмме, иллюстрируя только способы обеспечения видимости сервера для клиента. Связь использования на диаграмме лишь объявляет о наличии соответствующего отношения между объектами классов, оставляя реализацию на откуп разработчику.

Контрольные вопросы

- 1) Чем характеризуется поведение объекта?
- 2) Что такое протокол?
- 3) Что дает связь классу?
- 4) Какие виды отношений существуют между объектами?
- 5) Что такое ассоциация и что она представляет?
- 6) Что показывает мощность отношения?
- 7) Что такое агрегация?
- 8) Какие существуют способы обеспечить видимость связанных объектов?

Наследование

Наследование

- **Наследование** – это такое отношение между классами, когда один класс частично или полностью повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Наследование устанавливает между классами иерархию "общее-частное".

- Связи наследования также называют обобщениями (`generalization`) и изображают в виде больших белых стрелок от класса-потомка к классу-предку

- Рассмотрим учет сотрудников организации. Каждый сотрудник – это, прежде всего, человек. Студент, работник, руководитель – частные случаи объекта "человек", определяющего общие характеристики для всех своих разновидностей. Организация этих понятий в иерархию позволяет избежать повторения кода и обращаться с объектами производных классов как с объектами базового.

- Возьмем за основу иерархии класс `person`, немного изменив его структуру ради соответствия концепции наследования. Поскольку для всех объектов кадровой структуры требуется знать имя, фамилию, а также год рождения для определения возраста, объявим данные атрибуты как `protected`, заменив заодно тип представления имени и отчества на более удобный в работе тип `string`.

Наследование

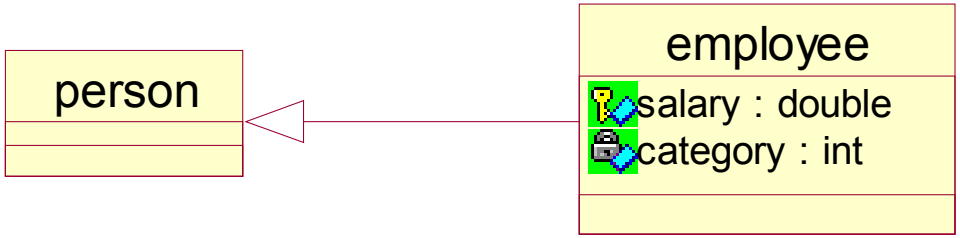
```
class person
{
   public:
   void set_year(int value);
   const int get_year() const;
   void set_sirname(string& value);
   const string& get_sirname() const;
   void set_name(string& value);
   const string& get_name() const;
   protected:
   string name;
   string sirname;
   int year;
};
```

- Работник (`employee`) отличается от просто человека (`person`) тем, что работает в некотором подразделении (`department`) и получает определенную заработную плату. Руководитель (`manager`), являясь в свою очередь работником предприятия, отвечает за определенную группу подчиненных и может получать заработную плату, складывающуюся из основной ставки и надбавки в виде процента от заработной платы своих работников.

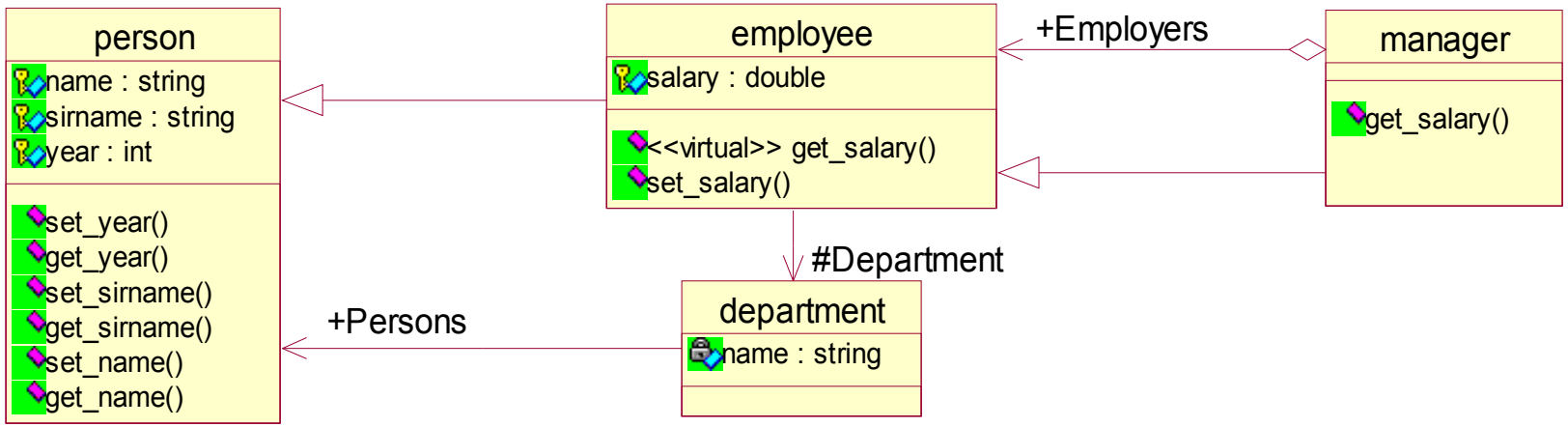
Наследование

- Выделять новый класс из существующего стоит лишь тогда, когда он знает или делает то, чего не знает и не делает объект базового класса. Это означает присутствие в производном классе атрибута или метода, неприменимого для объектов класса-предка.

- Когда же требуется только выразить различие между объектами по категориям, достаточно сделать это просто по значению атрибута, выражающего место конкретного объекта среди прочих. В подобном случае наша диаграмма классов могла бы выглядеть так:



- Нам же требуется не просто отличать руководителя от просто работника, но и учитывать его связь с множеством подчиненных. Эта связь, в том числе, может использоваться для расчета заработной платы.



Наследование

- Отношение наследования между классами вовсе не исключает различных видов отношений между их экземплярами. Так, мы вправе связать экземпляр класса manager с множеством объектов, описывающих подчиненных ему работников employee, а каждого из работников – с тем подразделением department, в котором тот работает. Разумно предположить, что и от подразделения можно будет перейти к списку связанных с ним лиц (по атрибуту `persons`).

```
class employee : public person
{
public:
     virtual const double get_salary() const;
     void set_salary(double value);
protected:
     double salary;
     department* Department;
};
```

- Связи между экземплярами классов реализуются посредством атрибутов, для которых также следует указывать область видимости в зависимости от того, кому они должны быть доступны.

Наследование

- Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства.
- Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов.
- По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт:

```
class manager : public employee
{
public:
     list<employee*> Employers;
     virtual const double get_salary();
};
```

Наследование

- **Правила наследования различных методов:**

- Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы.

- Порядок вызова конструкторов:
  - Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (без параметров).
  - Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.
  - В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.

Наследование

- Операция присваивания не наследуется и, поэтому ее также требуется явно определить.

- Деструкторы не наследуются, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.

- В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, это будет сделано автоматически.

- Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем — деструкторы элементов класса, а потом деструктор базового класса.

Наследование

- Теперь попробуем учесть отличия руководителя от просто работника, заключающиеся в особенности вычисления заработной платы. Метод `get_salary()` в таком случае должен возвращать не просто значение атрибута `salary`, определяющего тариф по ставке работника…

```
const double employee::get_salary()
{
   return salary;
}
```

- …, а ещё и учитывать величину надбавки:

```
const double manager::get_salary()
{
   double add = 0; list<employee*>::iterator i;
   for (i = Employers.begin(); i != Employers.end(); i++)
   {
        add += 0.03*(*i)->get_salary();
   }
   return salary+add;
}
```

Наследование

- Таким образом, следует вызывать нужный метод в зависимости от того, объект какого класса нам встретился в ходе работы программы. Такое поведение, называемое полиморфным, реализуется с помощью механизма виртуальных методов.

- Для того чтобы иметь возможность переопределить поведение метода `get_salary()` в подклассе `manager`, мы объявили его в классе `employee` как виртуальный.

Контрольные вопросы

- 1) Что такое наследование?
- 2) Что позволяет строить механизм наследования?
- 3) Конструкторы наследуются?
- 4) Что наследует производный класс?
- 5) Что представляет множественное наследование?
- 6) Если иерархия состоит из нескольких уровней, то в каком порядке вызываются деструкторы?

Виртуальные методы

Виртуальные методы

- Доступ к объектам иерархии классов лучше всего осуществляется через указатели на базовый тип. При наследовании со спецификатором `public` такому указателю можно присваивать адрес любого производного класса.

- Проблема заключается в том, что компилятор не будет знать, с каким именно классом связан указатель.

Виртуальные методы

- Например, мы имеем в наличии трех работников: двух подчиненных (A и B) и их руководителя C, составляющих множество E. Им назначена зарплата в 3300, 4400 и 5500 рублей соответственно. У нас стоит задача представить фактические доходы работников, которые будут определяться не только окладом.

```
void main()
{
   employee A, B;       manager C;
   employee* E[3];
   E[0] = &A; E[1] = &B; E[2] = &C;
   C.Employers.push_back(&A); C.Employers.push_back(&B);
   E[0]->set_salary(3300);
   E[1]->set_salary(4400);
   E[2]->set_salary(5500);

   for (i = 0; i < 3; i++)
   {
   cout << E[i]->get_salary() << endl;
   }
}
```

Виртуальные методы

- Отметим, что во время компиляции не будет известно, на какой объект указывает `E[i]`, следовательно, класс будет определяться по типу указателя. Соответственно будет выбран и правильный метод для вычисления заработной платы.
- В C++ реализован механизм позднего связывания, когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью **виртуальных методов** и рассмотрен в следующем разделе.
  Для определения виртуального метода используется спецификатор **`virtual`**:
- **Правила описания и использования виртуальных методов:**
- Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а с отличающимся набором параметров – обычным.
- Виртуальные методы наследуются, т.е. переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя.
- Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости.
- Виртуальный метод не может объявляться с модификатором static, но может быть объявлен как дружественный.

Виртуальные методы

Механизм позднего связывания

- Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает *таблицу виртуальных методов* (vtbl), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в vtbl одно и то же смещение для каждого класса в пределах иерархии.

- Каждый объект содержит скрытое дополнительное *поле ссылки* на vtbl, называемое vptr. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).

- На этапе компиляции ссылки на виртуальные методы заменяются на обращения к vtbl через vptr объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы.

Виртуальные методы

- Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта.
- Четкого правила, по которому метод следует делать виртуальным, не существует. Можно дать рекомендацию объявлять виртуальными методы, для которых есть вероятность, что они будут переопределены в производных классах.
- Методы, 1) которые во всей иерархии останутся неизменными 2) которыми производные классы пользоваться не будут, делать виртуальными нет смысла.
- С другой стороны, при проектировании иерархии не всегда можно предсказать, каким образом будут расширяться базовые классы, а объявление метода *виртуальным* обеспечивает гибкость и возможность расширения.
- Виртуальный механизм работает только при использовании указателей или ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется *полиморфным.*
  - В данном случае полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

Виртуальные методы

Абстрактные классы

- **Абстрактным классом** называется класс, в котором есть хотя бы одна чистая виртуальная функция. **Чистой виртуальной функцией** называется функция, которая имеет определение:
- `virtual тип имя_функции (список_формальных_параметров) = 0;`
- Чисто виртуальный метод должен переопределяться в производном классе (возможно опять как чисто виртуальный).
- Класс, содержащий хотя бы один чисто виртуальный метод, называется абстрактным. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для других классов. Объекты абстрактного класса создавать нельзя
- При определении абстрактного класса необходимо иметь в виду следующее:
  - абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;
  - допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;
  - если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Виртуальные методы

Ключи доступа при наследовании

- При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью ключей доступа *private, protected u public:*

```
class имя : [private | protected | public] базовый_класс
 {тело класса};
```

- Если базовых классов несколько, они перечисляются через запятую. Ключ доступа может стоять перед каждым классом.

```
class A {...};
class B {...};
class C {...};
class D: A, protected B, public C { ... };
```

- По умолчанию для классов используется ключ доступа private, а для структур — public.
- Кроме этого, доступность в производном классе регулируется ключом доступа к базовому классу. Этот ключ указывается в объявлении производного класса и определяет вид наследования: public, private или protected.

Виртуальные методы

- *Открытое* наследование сохраняет статус доступа для всех элементов базового класса
- *Защищенное* – понижает статус доступа всех элементов базового класса со статусом `public` до `protected`
- Закрытое – понижает статусы доступа всех элементов базового класса со статусом `public` и `protected` до `private`.

- Производные классы наследуют элементы базового класса `person`, но некоторые функции будут выполняться в каждом классе по-своему. Например, функция `show()` будет для каждого класса выводить его собственные поля. Для вызова метода предка из потомка используется операция доступа к области видимости "`::`".

Виртуальные методы

Множественное наследование

- Множественное наследование означает, что класс имеет несколько базовых классов. Если в базовых классах есть одноименные методы, то может произойти конфликт идентификаторов, который устраняется при помощи операции доступа к области видимости `::`. В иерархию, представленную на рис. 12, добавим класс `employee_student` (сотрудники, которые учатся).

```
class employee_student: public employee, public student
{...};
employee_student A;
cout<<A.student::get_age(); //метод класса student
cout<<A.employee::get_age();//метод класса employee
```

- Если у базовых классов есть общий предок (`person` в примере), то производный класс унаследует два экземпляра поля предка, а это чаще всего нежелательно. Для вызова метода `get_age()` требуется явно указывать классы, в которых эти методы описаны, иначе компилятор не сможет разобраться к методу какого из базовых классов надо обратиться.

Виртуальные методы

- Чтобы избежать двойного наследования полей, при наследовании общего предка, его определяют с ключевым словом `virtual` (виртуальный класс).

```
class person
{...};
class student: virtual person
{...};
class employee: virtual person
{...};
class employee_student: public employee, public student
{...};
```

Класс `employee_student` будет содержать только один экземпляр полей класса `person`.

Контрольные вопросы

- 1) Что такое механизм позднего связывания и с помощью чего он реализован?
- 2) Правила использования виртуальных методов.
- 3) Что такое полиморфизм?
- 4) Что называется абстрактным классом?
- 5) Что нужно иметь в виду при определении абстрактного класса?
- 6) Что означает множественное наследование?

Шаблоны классов

Шаблоны классов

Шаблоны функций

- Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные (см. часть I). При перегрузке функции для каждого используемого типа определяется своя функция. Шаблон функции определяется один раз, но определение параметризируется, т. е. тип данных передается как параметр шаблона.

- *Формат шаблона:*

```
template <параметры_шаблона>
заголовок_функции
{тело функции}
```

- Таким образом, шаблон семейства функций состоит из 2 частей – заголовка шаблона: `template <список параметров шаблона>` и обыкновенного определения функции, в котором вместо типа возвращаемого значения и/или типа параметров, записывается имя типа, определенное в заголовке шаблона.

Шаблоны классов

```
template <class T> set_value(T& value)
{
   cout << "Enter value: " << endl; cin >> value;
}
```

- Подобные функции часто имеет смысл группировать в параметризованные утилиты классов, имеющие практически то же предназначение, что и обычные утилиты классов:

- Естественно, в некоторых случаях необходимо учитывать и особенности типов, указываемых в качестве параметров шаблона. Так, операции ввода и вывода потоковых классов определены по умолчанию только для простых типов. Соответственно, если мы желаем использовать такие операции для определяемых нами классов, то их требуется переопределить для работы с объектами этих классов.

Шаблоны классов

```
template <class T> class util
{
public:
   static void set_value(T&);
   static void show_value(T&);
};

template <class T> void util<T>::set_value(T& value)
{
   cout << "Enter value: " << endl; cin >> value;
};

template <class T> void util<T>::show_value(T& value)
{
   cout << "Value is: " << value << endl;
};
```

- Следует отметить, что объявление класса шаблона и реализация его функций должны размещаться в одной единице трансляции, под которой обычно подразумевается заголовочный файл, например "`util.h`".
- Обращение к таким функциям будут иметь свои особенности, поскольку потребуется указывать тип – параметр шаблона:

Шаблоны классов

Шаблоны классов

- Шаблоны классов так же, как и шаблоны функций поддерживают механизм обобщенного программирования, т. е. программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает применение абстрактного типа в качестве параметра при определении класса. Такой класс называется *параметризованным*. После того как шаблон класса определен, он может использоваться для определения конкретных классов. Процесс генерации компилятором определения конкретного класса по шаблону класса и аргументам шаблона называется *инстанцированием* шаблона.

- Определение параметризованного (шаблонного, обобщенного) класса имеет вид:

```
template <параметры шаблона>
class имя_класса
{
      ...
};
```

Шаблоны классов

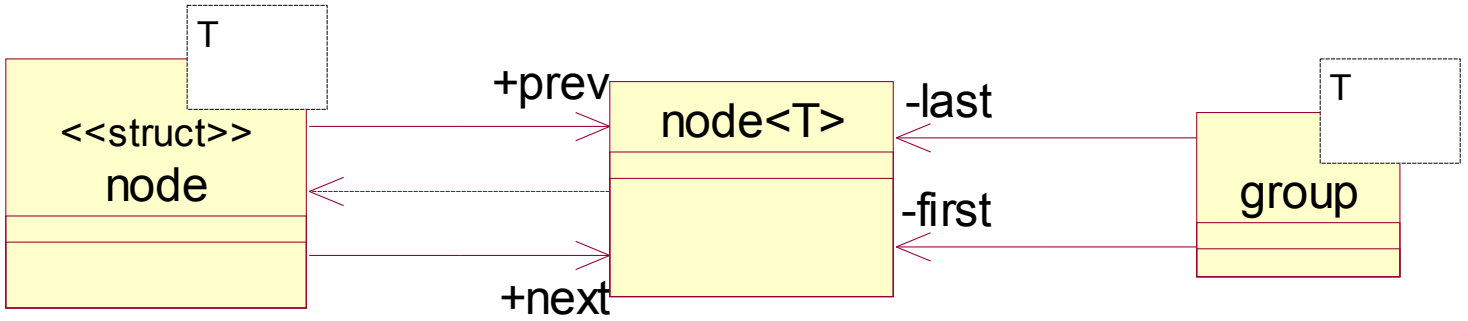
- В качестве примера для шаблона продолжим рассмотрение нашего класса-группы для объектов, поскольку самым распространенным способом использования шаблонов классов является реализация различных видов контейнеров.
- Ниже рассмотрен шаблон класса group, предназначенного для определения контейнерных типов с элементами любых типов (как стандартных, так и определённых программистом).

```
template <typename T> class group
//T – параметр шаблона
{
public:
   group();
   void push_back(T);
   void for_each(void (*)(T&));
protected:
   node<T>* last;
   node<T>* first;
};
```

Шаблоны классов

- Наш объект-группа представляет собой заготовку двунаправленного списка, реализуемого посредством включения указателей на первый и последний элемент типа node. Каждый из них реализуется шаблоном структуры, включающим в себя помимо указателей на соседние элементы (`prev` и `next`) и атрибут data хранимого элементом значения типа `T`.

```
template <class T> struct node
{
public:
   node<T>* prev;
   T data;
   node<T>* next;
};
```



Шаблоны классов

- В проекте, состоящем из нескольких файлов, определение шаблона класса обычно выносится в отдельный файл. Но для того, чтобы инстанцировался конкретный экземпляр шаблона класса необходимо, чтобы определение шаблона находилось в одной единице трансляции с этим экземпляром. Поэтому все определение шаблонного класса размещается в заголовочном файле, а затем этот файл подключается к нужным файлам с помощью директивы `#include`. Чтобы этот файл не включался повторно, используется директива `#ifndef`.

- По этой причине реализацию методов шаблона мы размещаем в том же заголовочном файле, где находится и определение класса шаблона – в "`group.h`".

```
template <typename T> group<T>::group()
{
   first = 0; last = 0;
};
```

Шаблоны классов

```
template <typename T> void group<T>::push_back(T newdata)
{
   if (last)
   {
        last->next = new node<T>;
        last->next->prev = last;
        last = last->next;
        last->next = 0;
   }
   else
   {
        first = new node<T>;
        first->prev = 0;
        first->next = 0;
        last = first;
   }
   last->data = newdata;
};
```

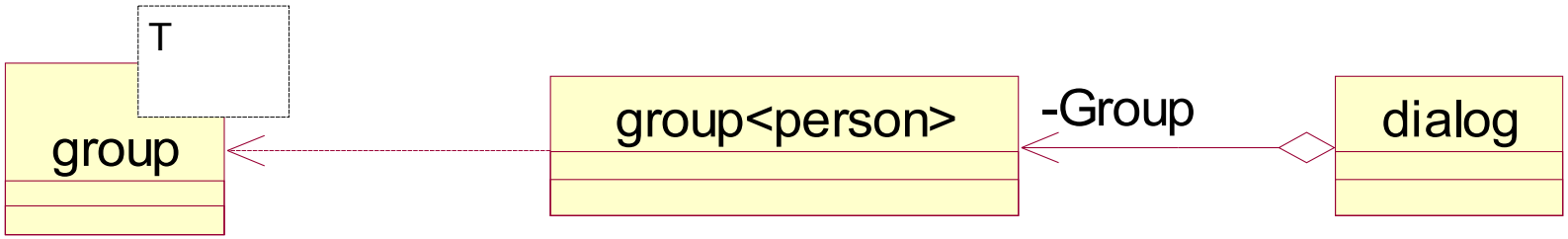
Шаблоны классов

```
template <typename T> void group<T>::for_each(void(*fp)(T&))
{
   node<T>* nx = first;
   while (nx)
   {
        fp(nx->data);
        nx = nx->next;
   }
}
```

- При включении шаблона класса в программу никакие классы не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо параметра шаблона указывается конкретный тип. Экземпляр создается либо объявлением объекта, либо объявлением указателя на инстанцированный шаблонный тип с присваиванием ему адреса с помощью операции `new`. Встретив такие объявления, компилятор создает код исходного класса.

```
class dialog
{
public:
   dialog();
   virtual ~dialog();
private:
   group<person> Group;
};
```

Шаблоны классов



- Приведённая выше диаграмма обозначает, что экземпляр класса `dialog` содержит атрибут `Group` типа `group<person>`, который, в свою очередь, определяется посредством использования шаблона group с формальным параметром `T`, вместо которого подставляется фактический `person` или любой другой необходимый тип.

Шаблоны классов

- Пример ниже показывает работу с созданным контейнером `Group` посредством объекта диалога dialog.

```
dialog::dialog()
{
   person x; int q;
   cout << "How many instances?"; cin >> q;

   for (int i = 0; i < q; i++)
   {
        Group.push_back(x);
   }
   Group.for_each(util<person>::set_value);
   Group.for_each(util<person>::show_value);
}
```

- Создаваемый диалог предоставляет механизмы работы клиента с локально создаваемым сервером – контейнером `Group`, посредством запроса количества `q` экземпляров класса `person` у пользователя и размещением оных в контейнере путём вызова его метода `push_back()`.
- Последующее определение состояния каждого объекта группы и вывод сведений о них выполняется с помощью передаваемых методу-итератору `for_each()` функций, применяемых к последовательности элементов контейнера.

Шаблоны классов

- Отметим, что обеспечить последовательный доступ ко всем частям объекта можно не только определяя итерации как часть протокола объекта, но и создавая отдельные объекты, ответственные за итеративный опрос других структур. Рассмотрим теперь и второй подход по тем причинам, что наличие выделенного итератора позволяет одновременно проводить несколько просмотров одного и того же объекта и, выделение итератора в качестве отдельного механизма поведения способствует большей ясности в описании класса.

- *Пассивный итератор* (рассмотренный ранее), применяет функцию, предоставляемую клиентом, ко всем элементам последовательности за одну операцию. *Активный итератор* (который мы сейчас и рассмотрим), требует каждый раз от клиента явного обращения к себе для перехода к следующему элементу.

Шаблоны классов

- В таком случае каждому объекту-итератору должен быть поставлен в соответствие определенный элемент объекта-контейнера. Играя роль указателя, активный итератор на самом деле таковым не является, поэтому попытка передачи такого объекта вместо адреса вызовет ошибку. Его работа заключается в том, чтобы переходить к следующему (префиксный оператор ++) или предыдущему элементу в списке (метод end() служит для определения нахождения итератора в его границах). Доступ к самим хранимым элементами данным осуществляется с помощью операторов -> и *. Метод reset() здесь присутствует для "сброса" состояния итератора к началу последовательности.

```
template <typename T> class group_iterator
{
public:
   group_iterator(group<T>&);
   void reset();
   group_iterator<T>& operator++();
   bool end();
   T* operator->();
   T& operator*();
protected:
   group<T>& G;
   node<T>** current;
};
```

Шаблоны классов

- В первую очередь наш итератор требуется связать со структурой, элементы которой он должен перебирать, что мы делаем при создании объекта класса `group_iterator`, и связать его с элементом последовательности. Для этого мы инициализируем атрибут-ссылку `G` и атрибут `current`, хранящий адрес указателя на объект, описывающий элемент последовательности – структуру типа `node`.

- Поясним, почему нам потребовалось вводить `current` как "указатель на указатель". Дело в том, что, создавая объект-итератор, мы связываем его с первым элементом последовательности, адрес которого хранится в атрибуте `first` объекта-группы. Если последовательность ещё не содержит элементов, то значение `first` будет нулевым, до тех пор, пока элемент не будет добавлен. Создав атрибут `current` как копию `first`, а не как его адрес (который остаётся неизменным), мы бы остались в неведении относительно фактического значения указателя на первый элемент списка, изменяющегося в случае добавления или удаления данного элемента. В нашем же случае значения получаются путем выполнения операции разыменования.

Шаблоны классов

- Рассмотрим, как можно реализовать шаблон функции для поиска элементов в массиве, который хранит объекты типа `Data`:

```
template <class Data>
Data* Find(Data*mas, int n, const Data& key)
{
   for(int i=0;i<n;i++)
   if (*(mas + i) == key)
        return mas + i;
   return 0;
}
```

- Функция возвращает адрес найденного элемента или 0, если элемент с заданным значением не найден.

- Эту функцию можно использовать для поиска элементов в массиве любого типа, но использовать ее для списка нельзя, поэтому авторы STL ввели понятие итератора. *Итератор* более общее понятие, чем указатель. Тип `iterator` определен для всех контейнерных классов STL, однако, реализация его в разных классах разная.

Шаблоны классов

- К основным операциям, выполняемым с любыми итераторами, относятся:
  - Разыменование итератора: если `p` – итератор, то `*p` – значение объекта, на ко-торый он ссылается.
  - Присваивание одного итератора другому.
  - Сравнение итераторов на равенство и неравенство (`==` и `!=`).
  - Перемещение его по всем элементам контейнера с помощью префиксного (`++p`) или постфиксного (`p++`) инкремента.

- Реализация итератора специфична для каждого класса, поэтому при объявлении всегда указывается область видимости:

```
vector<int>::iterator iter1;
list<person>::iterator iter2;
```

Шаблоны классов

- Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику. Вместо привычной формы

  `for (i =0; i < n; ++i)` используется следующая:

  `for (i = first; i != last; ++i),`

  где `first` - значение итератора, указывающее на первый элемент в контейнере, а `last` – значение итератора, указывающее на воображаемый элемент *за последним* элементом контейнера.
- Операция сравнения `<` заменена на операцию `!=`, т. к. операции `<` и `>` для итераторов в общем случае не поддерживаются.
- Для всех контейнерных классов определены унифицированные методы begin() и `end()`, возвращающие адреса `first` и `last` соответственно.

Шаблоны классов

- **Правила описания шаблонов:**
  - шаблоны методов (функций) не могут быть виртуальными;
  - шаблоны классов могут содержать статические элементы, дружественные функции и классы;
  - шаблоны могут быть производными, как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.

Шаблоны классов

Специализация

- Иногда возникает необходимость определить специализированную версию шаблона для какого-то конкретного типа его параметра (или одного из параметров).
- Рассмотрим отношение `T a<T b`, где `T` – параметр шаблона. Строки `char*` сравниваются не так как числа, поэтому имеет смысл применить специализацию шаблона.

```
template <class T>
class Sample
{
   bool Less(T) const;
   ...
};
//специализация для char*
template < >
class Sample<char*>
{
   bool Less(T) const;
   ...
};
```

Шаблоны классов

Использование классов функциональных объектов для настройки шаблонных классов

- **Функциональным объектом** называется объект, для которого перегружена операция вызова функции `operator()`. Класс, экземпляром которого является этот объект, называется **функциональным классом**. В таком классе не нужны другие поля и методы. Использование такого класса имеет специфический синтаксис.

```
struct LessThan
{
   bool operator()(const int x, const int y)
   {
        return x<y;
   }
};
int main()
{
   LessThan x;  //объект функционального класса
   int a = 5,b = 10;
   if( x(a,b)) cout<<a<<"<"<<b;
}
```

Шаблоны классов

- Пример шаблонного класса для выбора одного из двух значений, хранящихся в этом классе. Параметрами шаблона являются тип значений и критерий выбора.

```
template <class T>
struct LessThan
{
   bool operator()(const T&x,const T&y)
   {
   return x<y;
   }
};
struct GreaterThan
{
   bool operator()(const T&x,const T&y)
   {
   return x>y;
   }
};
```

Шаблоны классов

```
template <class T, class Compare>
class PairSelect
{
   T a, b;
   public:
   PairSelect(const T &x, const T & y):a(x),b(y){} //конструктор
   void Show()const
   {
        if(Compare()(a,b)) cout<<a; else cout<<b;
   }
};
void main()
{
   PairSelect<int,LessThan<int> >p1(15,10);
   p1.Show();  //вывод 10
   PairSelect<double,GreaterThan<double>>p2(15.5,10.1);
   p2.Show();  //вывод 15.5
}
```

Шаблоны классов

Средства стандартной библиотеки шаблонов (STL)

- Стандартная библиотека шаблонов STL (Standard Template Library), входящая в набор стандартных средств языка C++, предназначена для облегчения жизни разработчиков, дабы избавить их от самостоятельного программирования наиболее часто используемых архитектурных решений. В неё включены такие фундаментальные механизмы, как:
  - потоки;
  - строки;
  - контейнерные классы;
  - обобщённые алгоритмы;
  - итераторы;
  - численные методы.

Шаблоны классов

Ответственность классов и объектов

- Усложнение информационной системы за счет универсальных объектов, нагруженных всевозможными функциями, неизбежно приводит к потере качества и трудностям в совершенствовании программы.

- Предметная область определяется классами, моделирующими реальные сущности ключевых задач системы. Под этими сущностями обычно подразумеваются люди, места, вещи, организации, концепции и события.

- Соответствующие им классы должны описывать только тот минимум свойств и поведения, который характерен для реального прототипа объекта. Например, класс person не несет никакой ответственности за непосредственное взаимодействие с пользователем или базой данных, а используются другими объектами и функциями. Такими клиентами могут быть и средства диалога с человеком.

- В большинстве случаев именно человек определяет характеристики создаваемых объектов предметной области, но делая это с помощью элементов диалога. Сам диалог с пользователем чаще всего реализован с помощью форм графического или текстового интерфейса.

Контрольные вопросы

- 1) Что такое шаблоны и для чего они вводятся?
- 2) Какой класс называется параметризированным?
- 3) Что такое пассивный и активный итераторы и чем они отличаются?
- 4) Какие операции выполняются с любыми итераторами?
- 5) Перечислите основные правила описания шаблонов.
- 6) Что называется функциональным объектом?
- 7) Что такое библиотека STL? Что в нее включено?

Потоковые классы

Потоковые классы

- **Потоковые классы C++**
- Наиболее частой задачей является получение от пользователя определённых данных или наоборот передача данных пользователю. Её решение можно рассматривать и как часть реализации некоторой функции, так и в виде самостоятельного объекта, предназначенного для получения данных от человека.

- Для примера можно взять поле ввода или меню различных видов: и то, и другое – объекты экранного диалога.

- Обычно ввод-вывод выполняется при помощи библиотеки потоков, системы графического пользовательского интерфейса или функциями ввода-вывода языка C. Эти интерфейсы ввода-вывода в первую очередь направлены на считывание и запись отдельных значений разнообразных типов.

Потоковые классы

- Мы будем рассматривать потоки как наиболее простой вариант для реализации интерфейса с пользователем путём текстового представления экранного диалога.

- *Поток* — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику.

- Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специальную область оперативной памяти – *буфер*.

- Буфер накапливает байты, и фактическая передача данных выполняется после заполнения буфера. При вводе это дает возможность исправить ошибки, если данные из буфера еще не отправлены в программу.

- При неформатированном вводе-выводе передача информации осуществляется блоками байтов данных без какого-либо преобразования. При форматированном байты группируются таким образом, чтобы их можно было воспринимать как типизированные данные (целые числа, строки символов, числа с плавающей запятой и т. п.)

Потоковые классы

- По направлению обмена потоки можно разделить на:
  - *входные* (данные вводятся в память);
  - *выходные* (данные выводятся из памяти);
  - *двунаправленные* (допускающие как извлечение, так и включение).

- По виду устройств, с которыми идёт работа, потоки делят на *стандартные*, *файловые* и *строковые*.
  - **стандартные потоки** предназначены для передачи данных от клавиатуры и на экран;
  - **файловые потоки** — для обмена информацией с файлами на внешних носителях данных (например, на магнитном диске);
  - **строковые потоки** — для работы с массивами символов в оперативной памяти.

Потоковые классы

Стандартные потоки

- Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов – `ios` и `streambuf`.
  - класс `ios` содержит общие для ввода и вывода поля и методы;
  - класс `streambuf` обеспечивает буферизацию потоков и их взаимодействие с физическими устройствами.

- Между этими классами существует связь: в классе `ios` есть поле `bp`, которое является указателем на `streambuf`.

- Стандартному потоку ввода соответствует класс `istream`, стандартному потоку вывода – класс `ostream`. Эти классы являются производными от класса `ios`.

- Класс `iostream` является наследником классов `istream` и `ostream`, это класс двунаправленных потоков, он обеспечивает общие средства потокового ввода/вывода. В библиотеке `iostream` автоматически доступны объекты: `cin`, соответствующий стандартному потоку ввода, и `cout` для потока вывода. Оба эти потока являются буферизированными.

Потоковые классы

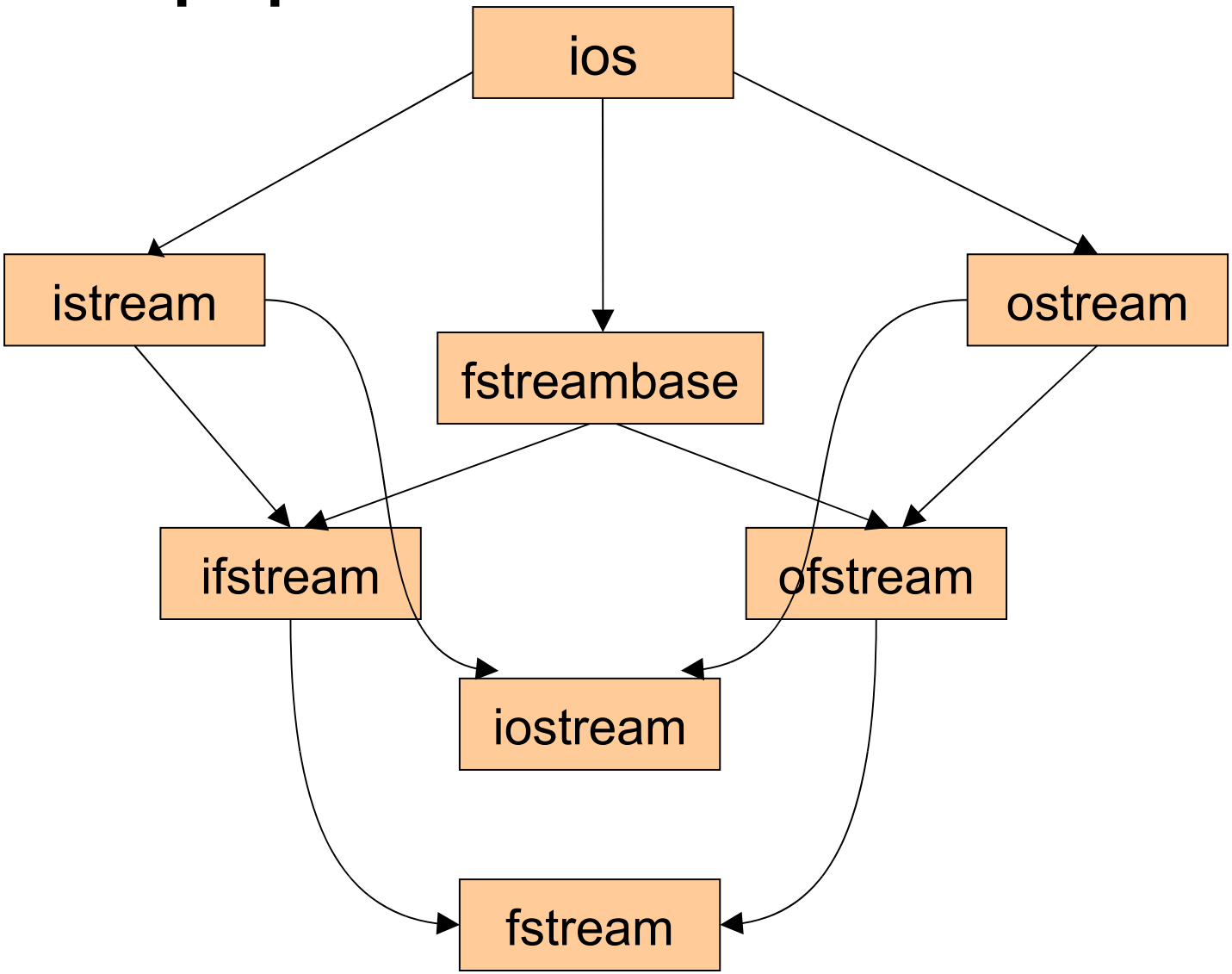
- `ostream` определяет оператор `<<` ("записать в") для управления выводом встроенных типов. Класс `istream` обеспечивает оператор ввода `>>` ("прочесть из") для небольшого набора стандартных типов.

- Означенные операторы может быть переопределены разработчиком для других вводимых типов.

- Эти функции должны возвращать ссылку на экземпляры классов потоков для того, чтобы к ней можно было применить другой `operator<<()` или `operator>>()` в зависимости от используемого потока.

- Оператор `>>` пропускает символы-разделители (пробел, табуляция, новая строка, новая страница и возврат каретки).

иерархия потоковых классов



Потоковые классы

- Если компилятор встретит в программе операторы:

```
int x = 5;
cout<<x;
```

  он определит, что левый операнд имеет тип `ostream`, а правый – `int`.

- В заголовочном файле `ostream` он найдет прототип метода ostream& `operator<<(int)` и вызовет метод `cout.operator<<(x)`, где `x` имеет тип `int`.

- В результате работы этого метода на экране будет выведено число 5.

- В случае вывода на экран значений встроенных типов (`int, float, double` и др.) класс `ostream` обеспечивает их преобразование из внутреннего двоичного формата к изображению в виде строки символов. При этом выполняется форматирование, которое установлено по умолчанию полями базового класса `ios`.

- Аналогично работает поток `cin`.

Потоковые классы

- Чаще всего нам приходится иметь дело с получением текстовой информации.
- Простейший объект диалога можно представить посредством класса (`edit`), у которого есть поле с заголовком (name) для ввода строки, отображаемое в некоторых координатах (`start_x` и `start_y`).
- Свойства элемента задаются конструктором с параметрами, а его отображение на экране и последующее получение строки обеспечивается методом `show()`.

```
class edit
{
   int start_x, start_y;
   string name;
public:
   edit(int, int, string);
   string show();
};

edit::edit(int sx, int sy, string n)
{
   start_x = sx;
   start_y = sy;
   name = n;
}
```

Потоковые классы

- Реализация метода `show()` требует определения *дескриптора устройства вывода* (`hStdout`), представленного текстовой консолью. Так же мы используем и функцию `SetConsoleCursorPosition()` для перемещения курсора в экранные координаты, определяемые структурой типа `COORD`.
- Эти механизмы относятся к библиотеке "`windows.h`".

```
string edit::show()
{
   HANDLE = GetStdHandle(STD_OUTPUT_HANDLE);
   COORD c; c.X = start_x; c.Y = start_y;
   SetConsoleCursorPosition(hStdout, c);
   cout << name; c.Y++; string s;
   SetConsoleCursorPosition(hStdout, c);
   getline(cin, s);
   return s;
}
```

- Для получения строки `s` из потока ввода `cin` мы задействуем функцию `getline()`, которая возвращает последовательность символов вместе с разделителями.

Потоковые классы

Потоки и типы, определяемые пользователем

- Для ввода и вывода в потоках используются перегруженные для всех стандартных типов операции чтения и извлечения << и >>. При этом выбор конкретной операции определяется типом фактических параметров.

- Иногда разработчику требуется единый способ определения состояния объектов. Под этим подразумевается ввод и вывод объектов с помощью соответствующих потоков.

- Для организации подобного интерфейса с пользователем необходимо определить операции чтения и извлечения в том классе, объекты которого будут передаваться потокам.

Потоковые классы

```
class person
{
public:
...
   friend istream& operator>>(istream& i, person& rhs);
   friend ostream& operator<<(ostream& o, const person& rhs);
...
private:
   string name;
   string sirname;
   int year;
   int weight;
};
istream& operator>>(istream& i, person& rhs)
{
   return i >> rhs.name >> rhs.sirname;
}
ostream &operator<<(ostream& o, const person& rhs)
{
   return o << rhs.name << " " << rhs.sirname << endl;
}
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

- **Форматирование при вводе/выводе**
- В потоковых классах форматирование выполняется тремя способами:
  - с помощью флагов;
  - манипуляторов;
  - форматирующих методов.
- Флаги представляют собой отдельные биты, объединенные в поле `x_flags` класса `ios`.

Ввод/вывод в языке C++ средствами стандартной библиотеки

- **Таблица 3. Флаги форматирования потоковых классов**

флаг	описание действия при установленном бите
`skipws`	при извлечении пробельные символы игнорируются
`left`	выравнивание по левому краю поля
`right`	выравнивание по правому краю поля
`internal`	знак числа выводится по левому краю, число – по правому; промежуток заполняется символами x_fill
`dec`	десятичная система счисления
`oct`	восьмеричная система счисления
`hex`	шестнадцатеричная система счисления
`showbase`	выводится основание системы счисления (0x для шестнадцатеричных чисел и 0 для восьмеричных)
`showpoint`	при выводе вещественных чисел печатать десятичную точку и дробную часть
`uppercase`	при выводе чисел использовать символы верхнего регистра для букв
`showpos`	печатать знак при выводе положительных чисел
`scientific`	печатать вещественные числа в форме мантиссы с порядком
`fixed`	печатать вещественные числа в форме с фиксированной точкой

Ввод/вывод в языке C++ средствами стандартной библиотеки

- Флаги [`left`, `right` и `internal`], [`dec`, `oct` и `hex`], а также [`scientific` и `fixed`] взаимно исключают друг друга, т.е. в каждый момент может быть установлен только один флаг из каждой группы.

- **Методы управления флагами класса ios:**

`long flags()`	возвращает текущие флаги потока;
`long flags(long)`	присваивает флагам значение параметра;
`long setf(long)`	устанавливает флаги, биты которых установлены в параметре;
`long setf(long,long)`	присваивает флагам, биты которых установлены в первом параметре, значение соответствующих битов второго параметра.

- Все функции возвращают прежние флаги потока

Ввод/вывод в языке C++ средствами стандартной библиотеки

- **В классе ios определены методы форматирования:**

`int width()`	возвращает значение ширины поля вывода;
`int width(int)`	устанавливает ширину поля вывода в соответствии со значением параметра;
`int precision()`	возвращает значение точности представления при выводе вещественных чисел;
`int precision(int)`	устанавливает значение точности представления при выводе вещественных чисел, возвращает старое значение точности;
`char fill()`	возвращает текущий символ заполнения;
`char fill(char)`	устанавливает значение текущего символа заполнения, возвращает старое значение символа.

- **… которые используют следующие защищенные атрибуты целого типа:**

`x_width`	минимальная ширина поля вывода;
`x_precision`	количество цифр в дробной части при выводе вещественных чисел
`x_fill`	символ заполнения поля вывода

Ввод/вывод в языке C++ средствами стандартной библиотеки

- Манипуляторы – это функции, которые можно включать в цепочку операций помещения и извлечения для форматирования данных, т. е. их можно включать прямо в поток. Манипуляторы делятся на простые, не требующие указания аргументов, и параметризованные.

- **Простые манипуляторы:**

`dec`	устанавливает при вводе и выводе флаг десятичной системы счисления;
`oct`	устанавливает при вводе и выводе флаг восьмеричной системы счисления;
`hex`	устанавливает при вводе и выводе флаг шестнадцатеричной системы счисления;
`ws`	устанавливает при вводе извлечение пробельных символов;
`endl`	при выводе включает в поток символ новой строки и выгружает буфер;
`ends`	при выводе включает в поток нуль-терминатор;
`flush`	при выводе выгружает буфер.

Ввод/вывод в языке C++ средствами стандартной библиотеки

Параметризированные манипуляторы требуют указания аргумента. Для их использования к программе надо подключить заголовочный файл `<iomanip>`.

`setbase(int n)`	задает основание системы счисления (n=10|8|16|0, 0 – по умолчанию, десятичное число, кроме случаев, когда вводятся восьмеричные или шестнадцатеричные числа)
`resetiosflags(long)`	сбрасывает флаги состояния потока, биты которых установлены в параметре;
`setiosflags(long)`	устанавливает флаги состояния потока, биты которых в параметре равны 1.
`setfill(int)`	устанавливает символ-заполнитель с кодом, равным значению параметра;
`setprecision(int)`	устанавливает максимальное количество цифр в дробной части для вещественных чисел;
`setw(int)`	устанавливает минимальную ширину поля вывода.

Ввод/вывод в языке C++ средствами стандартной библиотеки

В качестве ещё одного варианта реализации интерфейса с человеком приведём такой элемент диалога как таблица.

```
class person_util
{
public:
   static int start_x;

   static void make_title();
   static void make_table(person&);
};
```

- Метод `make_table()`, кроме возможностей библиотеки "`windows.h`", иллюстрирует механизмы использования потоков и их манипуляторов. Задача этого метода заключается в последовательном выводе колонок с информацией о значении атрибутов объекта типа `person`. Для этого из структуры `CSBI` извлекаются текущие координаты курсора, и в соответствующей горизонтальной позиции выводится строка необходимой ширины, имеющая нужный цвет и выравнивание. Атрибут цвета текста (`attribute`) определяется побитовым сложением числовых констант: `FOREGROUND_RED | FOREGROUND_GREEN| FOREGROUND_INTENSITY| BACKGROUND_BLUE`, что, в итоге, представляется десятичным числом 30 и желтыми символами на синем фоне.

Ввод/вывод в языке C++ средствами стандартной библиотеки

```
void person_util::make_table(person& arg)
{
   HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
   CONSOLE_SCREEN_BUFFER_INFO CSBI; COORD c;
   GetConsoleScreenBufferInfo(hStdout, &CSBI);
   c = CSBI.dwCursorPosition; c.X = start_x;
   WORD attribute = 30;
   SetConsoleCursorPosition(hStdout, c);
   SetConsoleTextAttribute(hStdout, attribute);
   cout.setf(ios::left);
   cout.width(20); cout << arg.get_name();
   SetConsoleTextAttribute(hStdout, 7);
   cout << " ";
   SetConsoleTextAttribute(hStdout, attribute);
   cout << setw(20) << arg.get_sirname();
   SetConsoleTextAttribute(hStdout, 7);
   cout << endl;
}
```

ставим флаг "выравнивание налево":

используем метод форматирования ширины поля вывода:

функция-манипулятор тоже определяет ширину:

Создание заголовка таблицы методом `make_title()` мы не[вы]ше представляет собой упрощенный вариант рассмотренного `make_table()`. Напомним лишь, что нашему статическому атрибуту надо присвоить начальное значение:

```
int person_util::start_x = 0;
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

Пользователь может создать свой собственный манипулятор и определить в нем правила, по которым будет осуществляться вывод информации в поток

```
typedef ostream&(*point)(ostream&, int, char);//point - тип-указатель
                                              //на функцию-манипулятор:
class mnp       //класс для манипулятора
{
   int width;   //ширина поля вывода
   char fill;   //символ-заполнитель
   point fun;   //функция, определяющая ширину и заполнитель

public:
   mnp(int w, char c, point f) : width(w), fill(c), fun(f) {}
   //перегруженная функция-операция для вывода объекта в поток:
   friend ostream& operator<<(ostream&, mnp);
};

ostream& operator<<(ostream& out, mnp m)
{
   return m.fun(out, m.width, m.fill);//возвращаем поток как результат
                                      //работы функции-манипулятора:
}
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

```
ostream& format(ostream& out, int w, char f)
{ //выполняем форматирование для выводимых данных:
   out.width(w);     //ширина
   out.fill(f);      //заполнитель
   //правое выравнивание для всех данных,
   //вывод целых - в шестнадцатиричной СС и верхнем регистре:
   out.flags(ios::right|ios::uppercase|ios::hex);
   return out;
}

void main()
{ //выводим A на 15 позициях, с заполнением пустоты точками:
   int A = 987; cout << mnp(15, '.', format) << A << endl;
}
```

В данном случае для объекта класса `mnp` вызывается перегруженный оператор `<<`, который обращается к функции `format()` через атрибут-указатель `fun` объекта `m`, передавая ей основные параметры формата вывода. С тем же успехом, вместо конструктора класса, можно было бы вызвать функцию вида:

```
mnp set_stream(int w, char f)
{
   return mnp(w, f, format);
}
```

…подменяющую явное обращение к объекту класса `mnp`. В любом случае, пользователь увидел бы что-нибудь наподобие "`..........3DB`".

Ввод/вывод в языке C++ средствами стандартной библиотеки

- **Методы обмена с потоками**

Для работы с объектами потоковых классов наряду с операциями (>> и <<) определены ещё и методы неформатированного чтения и записи в поток. Преобразование данных при этом не выполняется.

`get()`	возвращает код извлеченного из потока символа или `EOF`, если достигнут конец файла;
`get(c)`	присваивает код извлеченного из потока символа аргументу `c`;
`get(buf, num, lim='\n')`	читает из потока символы, пока не встретится символ `lim` (по умолчанию `'\n'`) или пока не будет прочитано `num-1` символов, прочитанные символы помещаются в массив `buf` и к ним добавляется `'\0'`. Символ `lim` остается в потоке.
`getline(buf, num, lim='\n')`	аналогично предыдущему методу, но `lim` удаляется из потока (в массив `buf` он все равно не записывается);
`peek()`	возвращает код следующего символа в потоке, но не извлекает его из потока;
`ignore(num= 1, lim = EOF)`	считывает и пропускает символы до тех пор, пока не будет прочитано `num` символов или не встретится разделитель, заданный параметром `lim`. Возвращает ссылку на текущий поток;
`read(buf, num)`	считывает из потока `num` символов в массив `buf`.

Ввод/вывод в языке C++ средствами стандартной библиотеки

`gcount()`	возвращает количество символов, прочитанных последним вызовом функции неформатированного ввода;
`rdbuf()`	возвращает указатель на буфер типа `streambuf`, связанный с данным потоком.
`put(c)`	выводит в поток символ `c`;
`write(buf,num)`	выводит в поток num символов из массива `buf`.
`seekg(pos)`	устанавливает текущую позицию чтения в значение `pos`;
`seekg(offs, org)`	перемещает текущую позицию чтения на `offs` байтов, считая от одной из трех позиций, определяемых параметром `org`: `ios::beg` (от начала файла), `ios::cur` (от текущей позиции) или `ios::end` (от конца файла);
`tellg()`	возвращает текущую позицию чтения потока;
`flush()`	записывает содержимое потока вывода на физическое устройство;
`seekp(pos)`	устанавливает текущую позицию записи в значение `pos`;
`seekp (offs, org)`	перемещает текущую позицию записи на `offs` байтов, считая от одной из трех позиций, определяемых параметром `org`: `ios::beg` (от начала файла), `ios::cur` (от текущей позиции) или `ios::end` (от конца файла);
`tellp()`	возвращает текущую позицию записи потока;

Ввод/вывод в языке C++ средствами стандартной библиотеки

При организации диалогов с пользователем программы при помощи потоков необходимо учитывать буферизацию. Например, при выводе приглашения к вводу мы не можем гарантировать, что оно появится раньше, чем будут считаны данные из входного потока, поскольку приглашение появится на экране только при заполнении буфера вывода.

Для решения этой проблемы определена функция `tie()`, которая связывает потоки `istream` и `ostream` с помощью вызова вида `cin.tie(&cout)`. После этого вывод очищается (т. е. выполняется функция `cout.flush()`) каждый раз, когда требуется новый символ из потока ввода.

Приведем пример использования некоторых методов:

```
cin.ignore(7, ',');
streambuf SB = *cin.rdbuf();
cout << &SB;
```

Очистив поток ввода, мы получаем от пользователя символы, которые заносятся в поток при определенных условиях: когда пропущено 7 символов, или после того, как была встречена запятая. Далее инициализируем собственный буфер буфером потока `cin`, и передаем его потоку `cout`. Таким образом, для строки "123,4567890" получим результат "4567890", а для "12345678,90" уже "8,90"!

Ввод/вывод в языке C++ средствами стандартной библиотеки

Ошибки потоков

Наиболее распространенная ошибка при использовании потоков – не заметить, что введено не то, что предполагалось, поскольку на входе был не тот формат. Нужно или проверять состояние потока ввода перед использованием предположительно считанных данных, или задействовать исключения.

- Для отслеживания ошибок потоков в базовом классе `ios` определено поле `state`, отдельные биты которого отображают состояние потока.

`ios::goodbit`	нет ошибок;
`ios::eofbit`	конец файла;
`ios::failbit`	ошибка форматирования или преобразования;
`ios::badbit`	серьезная ошибка, пользоваться потоком нельзя;
`ios::hardfail`	неисправность оборудования.

Ввод/вывод в языке C++ средствами стандартной библиотеки

- Получить текущее состояние потока можно с помощью метода `int rdstate()`, но существуют и другие более удобные методы.

`int eof()`	возвращает ненулевое значение, если установлен флаг `eofbit`;
`int fail()`	возвращает ненулевое значение, если установлен один из флагов `failbit`, `badbit` или `hardfail`;
`int bad()`	возвращает ненулевое значение, если установлен флаг `badbit`;
`int good()`	возвращает ненулевое значение, если сброшены все флаги.
`void clear (int = 0)`	параметр принимается в качестве состояния ошибки, при отсутствии параметра состояние ошибки устанавливается 0;

Если произошла ошибка ввода, в результате которой установлен флаг только `failbit`, то работу с потоком можно продолжить, но сначала надо сбросить флаги ошибок с помощью метода `clear()`. Этот же метод используется и для установки флагов, если передать ему не нулевой аргумент.

`operator void*()`	возвращает нулевой указатель, если установлен хотя бы один бит ошибки, эта операция вызывается каждый раз, когда поток сравнивается с 0;
`operator ! ()`	возвращает ненулевой указатель, если установлен хотя бы один бит ошибки.

Ввод/вывод в языке C++ средствами стандартной библиотеки

```
Stream.clear();    //сбросить все флаги
Stream.clear(rdstate() | ios::eofbit);// установить флаг конец файла
Stream.clear(ios::eofbit);      // установить флаг конец файла.
                                //и сбросить все остальные

while(Stream)
{
   //ввод или вывод …
};
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

Файловые потоки

Файл – именованная информация на внешнем носителе, т. е. логически файл – это конечное множество последовательных байтов, следовательно, устройства, такие как принтер, клавиатура и монитор, являются частным случаем файлов.

- По способу доступа файлы можно разделить на
  - последовательные, чтение и запись в которые производится с начала байт за байтом;
  - файлы с произвольным доступом, допускающие чтение и запись в указанную позицию.
- Стандартная библиотека содержит три класса для работы с файлами:
  - `ifstream` – класс входных файловых потоков;
  - `ofstream` – класс выходных файловых потоков;
  - `fstream` – класс двунаправленных файловых потоков.

Эти классы являются производными от классов `istream`, `ostream` и `iostream` соответственно, поэтому они наследуют перегруженные операции `<<` и `>>`, флаги форматирования, манипуляторы, методы, состояние потоков и т. д.

- Использование файлов в программе предполагает следующие операции:
  - создание потока;
  - открытие потока и связывание его с файлом;
  - обмен (ввод/вывод);
  - уничтожение потока;
  - закрытие файла.

Ввод/вывод в языке C++ средствами стандартной библиотеки

Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом. Конструкторы с параметрами создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

```
ifstream(const char *name, int mode = ios::in);
ofstream(const char *name, int mode = ios::out | ios::trunc);
fstream(const char *name, int mode = ios::in | ios::out);
```

- Вторым параметром является режим открытия файла. Вместо значения по умолчанию можно указать одно из следующих значений, определенных в классе `ios`:

  По умолчанию файлы открываются в текстовом режиме. Это означает, что на вводе последовательность возврата каретки/перевода строки преобразуется в символ `'\n'`. На выводе символ `'\n'` преобразуется в последовательность возврат каретки/перевод строки. В двоичном режиме такие преобразования не выполняются.

- Функция компонента `ofstream::open` объявляется следующим образом:

```
void open(char * name, int=ios::out, int prot=filybuf::openprot);
```

- Аналогично, объявление `ifstream::open` имеет вид:

```
void open(char * name, int=ios::in, int prot=filybuf::openprot);
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

- Второй аргумент, называемый режимом открытия, имеет показанные умолчания. Аргумент режима открытия (возможно, связанный операцией ИЛИ с несколькими битами режима) можно явно задать в следующей форме:
  - `ios::in` – открыть файл для чтения;
  - `ios::out` – открыть файл для записи;
  - `ios::ate` – установить указатель на конец файла, читать нельзя, можно только записывать данные в конец файла;
  - `ios::app` – открыть файл для добавления;
  - `ios::trunc` – если файл существует, то создать новый;
  - `ios::binary` – открыть в двоичном режиме;
  - `ios::nocreate` – если файл не существует, выдать ошибку, новый файл не открывать;
  - `ios::noreplace` – если файл существует, выдать ошибку, существующий файл не открывать;

Открыть файл в программе можно с использованием либо конструкторов, либо метода `open`, имеющего такие же параметры, как и в соответствующем конструкторе.

```
fstream stream;        //конструктор без параметров
//открывается файл, который связывается через fbuf с потоком
stream.open("..\\f.dat",ios::in);
//конструктор с параметрами, создает и открывает для чтения файл
fstream stream("..\\f.dat",ios::in);
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

После того как файловый поток открыт, работать с ним можно также как и со стандартными потоками `cin` и `cout`. При чтении данных из входного файла надо контролировать был ли достигнут конец файла после очередной операции вывода. Это можно делать с помощью метода `eof()`.

Если в процессе работы возникнет ошибочная ситуация, то потоковый объект принимает значение равное 0.

Когда программа покидает область видимости потокового объекта, то он уничтожается, при этом перестает существовать связь между потоковым объектом и физическим файлом, а сам файл закрывается. Если файл требуется закрыть раньше, то используется метод `close()`.

Ввод/вывод в языке C++ средствами стандартной библиотеки

```
//вспомогательная функция
int GetInt(istream&in)  //ввод целого числа из потока in
{
   int value;
   while(1)  //бесконечный цикл
   {
      in>>value; //чтение целого числа
      if(in.peek=='\n')//если в буфере типа streambuf, связанном с //потоком
   in находится изображение целого числа, заканчивающегося //'\n', то после
   извлечения этого числа в буфере останется только //'\n'. Выполняется
   проверка этого условия.
      {
         in.get(); //буфер очищается
         break; //выход, т. к. число введено верно
      }
      else
      {
          cout<<"\nError in data; Repeat, please:"<<endl;
          in.clear();//сбрасываются флаги ошибок потока in
          while(in.get()!='\n'{};    //из буфера извлекаются все
      }                              //символы до символа '\n'
   }
   return value;
}
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

```
//функция чтения объекта из потока
istream& operator >>(istream& in, person& S)
{
   in.getline(S.name,20); // читает из потока символы, пока не
   //встретится символ '\n' или пока не будет прочитано 19 символов,
   //прочитанные символы помещаются  в массив S.name и к ним
   //добавляется '\0'.
   S.age=GetInt(in);  //вызов функции для ввода целого числа
   return in;
}
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

- Одним из типовых средств работы пользователя с данными программы является **меню**, под которым, кроме экранной формы, подразумевается и некоторый объект системы. Класс для него должен описывать характеристики экранного объекта и методы для работы с ним.

```
class menu
{
public:
   void add_string(string s); //добавить строку в меню
   int show();     //показать меню
   menu(int, int, int, int);  //конструктор с параметрами
   void redraw(); //перерисовка экранной формы
private:
   int pos;       //начальная позиция отображения
   int curr_pos; //текущая позиция маркера меню
   int start_y;  //начальная координата Y
   int start_x;  //начальная координата X
   unsigned int height;     //высота в строках
   unsigned int width;      //ширина в символах
   vector<string> Strings; //массив строк меню
   HANDLE hStdout;
};
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

Исходя из назначения данного класса, единственным конструктором, имеющим смысл, является конструктор с параметрами, задающий значения экранных координат, ширину и высоту меню, а также определяющий контекст. `menu::menu(int sx, int sy, int w, int h)`

```
{
   start_x = sx; start_y = sy; width = w; height = h;
   pos = 0; curr_pos = 0;
   hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
}
```

Поскольку пользователь осуществляет взаимодействие с меню на основе символьной информации, меню должно содержать массив составляющих его строк, что реализуется посредством включения в класс меню атрибута `Strings` контейнерного типа `vector<string>`. Добавление строк туда должно учитывать правила работы с экранным объектом, для чего имеется метод `add_string()`. Он определяет разницу между шириной меню в символах и длиной строки, после чего либо дополняет её пробелами, либо отсекает лишние символы, и только после этого добавляет измененную строку в контейнер `Strings`.

```
void menu::add_string(string s)
{
    int count = width - s.size();
    if (count) s.append(count, ' ');
    else s = s.substr(0, width);
    Strings.push_back(s);
}
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

- Метод `redraw()` предназначен для перерисовки меню при изменении его состояния, которое определяется номером `pos` первой отображаемой строки и позицией `curr_pos` маркера внутри строк, выводимых на экран. Эти значения непосредственно зависят от действий пользователя, обрабатываемых методом `show()`.

```
void menu::redraw()
{
    COORD c; c.X = start_x; c.Y = start_y; int y = 0;
    WORD attribute;
    vector<string>::iterator si = Strings.begin();
    for(int i = curr_pos; i < curr_pos+height &&
    si != Strings.end(); i++, si++, y++)
    {
    c.Y = start_y + y; SetConsoleCursorPosition(hStdout, c);
    if(i == pos)
        {
        attribute = FOREGROUND_RED|FOREGROUND_GREEN |

        FOREGROUND_INTENSITY|BACKGROUND_BLUE|BACKGROUND_INTENSITY;
        }
    else
        {
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

```
        attribute = FOREGROUND_RED|FOREGROUND_GREEN|
        FOREGROUND_INTENSITY|BACKGROUND_BLUE;
        };
   SetConsoleTextAttribute(hStdout,attribute);cout << Strings[i];
   SetConsoleTextAttribute(hStdout, 0);       cout << endl;
    }
}
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

Метод `show()` непосредственно реализует интерфейс с пользователем, изменяя состояние меню в зависимости от нажатия клавиш человеком, и возвращая по нажатию клавиши Enter номер выбранной строки.

```
int menu::show()
{
   int z;
   if (Strings.begin() != Strings.end())
   {
       redraw();
       do
       {
           z = _getch(); //получаем код нажатой клавиши
           switch (z)
           {
               case 72:  //нажата клавиша "стрелка вверх"
               {
                   SetConsoleTextAttribute(hStdout, 0);
                   if (pos == curr_pos && pos > 0)
                   --curr_pos;
                   if (pos > curr_pos) --pos;
                   break;
               }
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

```
                case 80: //нажата клавиша "стрелка вниз"
                        {
                                SetConsoleTextAttribute(hStdout, 120);
                                if (pos < Strings.size()-1) ++pos;
                                if (pos == curr_pos+height &&
                                    pos < Strings.size()) ++curr_pos;
                                break;
                        }
                }
                redraw();
        }
        while (z != 13);        //выполняем, пока не нажали Enter
        SetConsoleTextAttribute(hStdout, 7);
        return pos;
    }
    else
    {
        SetConsoleTextAttribute(hStdout, 7);
        return -1;
    }
}
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

Нижеследующий пример показывает создание меню на основе массива объектов класса person, имена и фамилии которых передаются объекту-меню M в качестве строк.

```
void main()
{
   person x; string s; int index;
   vector<person> Persons;
   vector<person>::iterator pi;
   for (int i=0; i < 2; i++)
   {
   cout << "Enter name: "; cin >> s;
   x.set_name(s);
   cout << "Enter sirname: "; cin >> s;
   x.set_sirname(s);
   Persons.push_back(x);
   }
   menu M(10, 10, 15, 4);
   for (pi = Persons.begin(); pi != Persons.end(); pi++)
   {
   M.add_string(pi->get_name()+" "+pi->get_sirname());
   }
   index = M.show();
   cout << Persons[index].get_sirname() << endl;
   cout << "Address of object is " << &Persons[index] << endl;
}
```

Ввод/вывод в языке C++ средствами стандартной библиотеки

- Поскольку метод `show()` возвращает индекс элемента меню, можно узнать, и какой объект был выбран, и какой у этого объекта адрес, о чём и говорят две последние строки кода. Теперь мы можем связать объект или указатель на него с любым другим объектом, передавая их как параметры соответствующим методам этих объектов. Отметим, что использование меню, естественно, не ограничивается выбором единственного объекта, и может быть продолжено путём очередных обращений к методу `show()`.

Контрольные вопросы

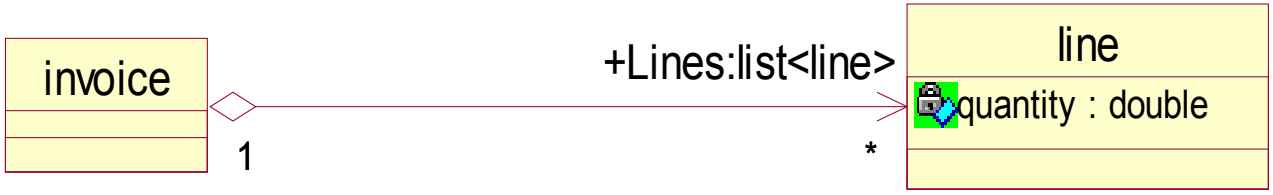
- 1) Что такое поток?
- 2) Какими бывают потоки по направлению обмена?
- 3) Какие бывают потоки по виду устройств, с которыми идет работа?
- 4) Стандартные потоки.
- 5) Какими способами в потоковых классах выполняется форматирование?
- 6) Что такое флаги и как они работают?
- 7) Что такое манипулятор и как они работают7
- 8) Оштбки потоков.Поиск и устранение.
- 9) Что такое файл? Какие классы существуют для работы с файлами?
- 10) Какие операции предполагает использование файлов в программе?

Контейнерные классы

Контейнерные классы

- **Контейнеры** — это объекты, содержащие другие однотипные объекты. Контейнерные классы являются шаблонными, поэтому хранимые в них объекты могут быть как встроенных, так и пользовательских типов. Эти объекты должны допускать копирование и присваивание. Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваивания не объявлены в них закрытыми или защищенными. Контейнеры STL реализуют основные структуры данных, используемые при написании программ.

- При описании диаграмм классов на UML, объявления атрибутов-контейнеров могут использоваться в качестве спецификаторов ролей экземпляров классов в отношениях ассоциации или агрегации:



- Так, приведенный выше пример описывает, что накладная (экземпляр класса `invoice`) включает множество строк – экземпляров класса `line` посредством описанного в классе `invoice` атрибута `Lines` типа `list<line>`, что в коде будет выглядеть следующим образом:

```
class invoice
{
public:
list<line> Lines;
};
```

Контейнерные классы

- **Обобщенные алгоритмы** реализуют большое количество процедур, применимых к контейнерам: поиск, сортировку, слияние и т. п. Однако они не являются методами контейнерных классов. Алгоритмы представлены в `STL` в форме глобальных шаблонных функций. Благодаря этому достигается универсальность: эти функции можно применять не только к объектам различных контейнерных классов, но также и к массивам. Независимость от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара указателей (`first` и `last`), задающая диапазон обрабатываемых элементов.

- Реализация указанного механизма взаимодействия базируется на использовании итераторов. Итераторы представляют собой обобщение концепции указателей и служат, как мы рассмотрели ранее, для обращения к элементам контейнера и последовательного продвижения по нему.

Контейнерные классы

Строки

- **Строка** является классом, который хранит символы и предоставляет операции, такие как доступ по индексу, конкатенацию и сравнение, которые мы обычно связываем с понятием "строка". Стандартная библиотека для реализации строк предоставляет шаблон класса `basic_string`, с помощью которого определён тип `string`, представляющий собой синоним для `basic_string<char>`:

```
typedef basic_string<char> string;
```

- Тип `basic_string` похож на `vector`, который мы рассмотрим далее, за исключением того, что `basic_string` вместо набора операций со списками, которые предлагает `vector`, обеспечивает некоторые типовые операции со строками, такие как поиск подстрок.
- В качестве типа своих элементов строка может использовать любой тип, причём эффективность кода может улучшиться при отсутствии операции копирования, определяемой пользователем, в типе символов.

```
basic_string<unsigned int> int_str1, int_str2;
int_str1 = 9876;
basic_string<unsigned int>::iterator si = int_str1.begin();
int_str1+=5432;
int_str1+=1098;
int_str2=7654;
int_str1.append(int_str2);
while (si != int_str1.end())
     cout << *si << endl; si++;
```

Контейнерные классы

- **Контейнеры**
- **Последовательные контейнеры** обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся
  - векторы (`vector`);
  - списки (`list`);
  - двусторонние очереди (`deque`).
- Кроме них, существуют специализированные контейнеры (или адаптеры контейнеров), реализованные на основе базовых :
  - стеки (`stack`);
  - очереди (`queue`) ;
  - очереди с приоритетами (`priority_queue`).
- **Ассоциативные контейнеры** обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров:
  - словари (`map`);
  - словари с дубликатами (`multimap`);
  - множества (`set`);
  - множества с дубликатами (`multiset`);
  - битовые множества (`bitset`).

Контейнерные классы

```
#include <map>
#include <list>
#include <vector>
#include "item.h"
#include "person.h"
void main()
{
   std::vector<int> V;          //вектор с элементами типа int

   std::list<person> L;         //список с элементами типа person

   //словарь для доступа к целым числам по адресу объекта типа item:
   std::map<item*, int, std::Less<item*> > M;
}
```

Контейнерные классы

- **Итераторы**
- В STL существуют следующие типы итераторов:
  - **Входные итераторы** (InputIterator) используются алгоритмами STL для чтения значений из контейнера, аналогично тому, как программа может вводить данные из потока cin.
  - **Выходные итераторы** (OutputIterator) используются алгоритмами для записи значений в контейнер, аналогично тому, как программа может выводить данные в поток cout.
  - **Прямые итераторы** (ForwardIterator) используются алгоритмами для навигации по контейнеру только в прямом направлении, причем, они позволяют и читать, и изменять данные в контейнере.
  - **Двунаправленные итераторы** (BidirectionalIterator) имеют все свойства прямых итераторов, но позволяют осуществлять навигацию по контейнеру и в прямом, и в обратном направлениях (для них дополнительно реализованы операции префиксного и постфиксного декремента).
  - **Итераторы произвольного доступа** (RandomAccessIterator) имеют все свойства двунаправленных итераторов плюс операции (наподобие сложения указателей) для доступа к произвольному элементу контейнера.

Контейнерные классы

- Для двунаправленных итераторов и итераторов произвольного доступа определены разновидности, называемые адаптерами итераторов. Обратный итератор (`reverse_iterator`) – это адаптер, просматривающий последовательность в обратном направлении. Итераторы вставки предназначены для добавления элементов в
  - начало (`front_iterator`);
  - конец (`back_iterator`) ;
  - произвольное место (`insert_iterator`) контейнера.

Контейнерные классы

- **Общие свойства контейнеров**
- Для удобства использования в большинстве контейнерных классов с помощью `typedef` определены синонимы типов, которые можно использовать при написании шаблонов функций, ничего не зная о настоящих типах контейнера:

- **Унифицированные типы, определенные в STL**

Тип	Пояснение
`value_type`	тип элемента контейнера
`size_type`	тип индексов, счетчиков элементов и т. д.
`iterator`	итератор
`const_iterator`	константный итератор (значения элементов изменять запрещено)
`reference`	ссылка на элемент константная
`const_reference`	ссылка на элемент (значение элемента изменять запрещено)
`key_type`	тип ключа для ассоциативных контейнеров
`key_compare`	тип критерия сравнения для ассоциативных контейнеров

Контейнерные классы

- Возьмем для примера функцию вычисления суммы всех элементов контейнера:

```
template <class C> typename C::value_type sum(const C& c)
{
     typename C::value_type s = 0;
     typename C::const_iterator p = c.begin();
     while (p!=c.end())
     {
          s+=*p;
          ++p;
     }
     return s;
}
```

- Теперь её можно использовать для работы с любым контейнером:

```
vector<int> v1;
int m[5]={1,2,3,4,5};
v1.insert(v1.begin(), m, m+5);
cout << sum(v1) << endl;
```

Контейнерные классы

операция или метод	пояснение
операции равенства (==) и неравенства (!=)	возвращают значение `true` или `false`
операция присваивания (=)	копирует один контейнер в другой
`clear()`	удаляет все элементы
`insert()`	добавляет один элемент или диапазон элементов
`erase()`	удаляет один элемент или диапазон элементов
`size_type size() const`	возвращает число элементов
`size_type max_size() const`	возвращает максимально допустимый размер контейнера
`bool empty() const`	возвращает `true`, если контейнер пуст
`iterator begin()`	возвращают итератор на начало контейнера (итерации будут выполняться в прямом направлении)
`iterator end()`	возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены)
`reverse_iterator begin()`	возвращают реверсивный итератор на конец контейнера (итерации будут выполняться в обратном направлении)
`reverse_iterator end()`	возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены)

Контейнерные классы

- **Использование последовательных контейнеров**
- К основным последовательным контейнерам относятся:
- Контейнер **вектор** (`vector`), является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации `[]` или метода `at`. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

- Контейнер **список** (`list`) организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки n-го элемента нужно последовательно выбрать предыдущие `n-1` элементов.

- Контейнер **двусторонняя очередь** (`deque`) во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

Контейнерные классы

- Существует пять способов определить объект для последовательного контейнера.
  - Создать пустой контейнер:

```
vector<int> VI;      list<double> LD;
```

  - Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:

```
vector<int> VI(100); //создаем вектор из 100 элементов
list<double> LD(20); //создаем список из 20 элементов
```

  - Создать контейнер заданного размера и инициализировать его элементы указанным значением:

```
//создаем вектор из 100 элементов и инициализируем их нулями:
vector<int> VI(100, 0);
//создаем очередь из 300 элементов и инициализируем их нулями:
deque<float> DF(300, 0.0);
```

Контейнерные классы

- Создать контейнер и инициализировать его элементы значениями диапазона `(first, last)` элементов другого контейнера:

```
//создаем и инициализируем массив из 7 элементов
int array[7] = (15, 2, 19, -3, 28, 6, 8);
//создаем вектор и инициализируем его элементами массива,
//array – указатель на первый элемент массива
vector<int> VI(array, array + 7);
//создаем список и инициализируем его элементами вектора,
//VI.begin() – метод, который возвращает указатель на
//первый элемент массива, VI.end() – возвращает указатель
//на фиктивный элемент, следующий за последним
list<int> LI(VI.begin() + 2, VI.end());
```

- Создать контейнер и инициализировать его элементы значениями элементов другого однотипного контейнера:

```
//создаем и инициализируем элементы нулями:
vector<int> V1(100,0);
//создаем и инициализируем элементами вектора v1:
vector<int> V2(V1);
```

Контейнерные классы

- **ВЕКТОР для `item`! И работа с ним.**
- Метод `insert` имеет следующие реализации:
  - `iterator insert(iterator pos, const T& key);` – вставляет элемент `key` в позицию, на которую указывает итератор pos, возвращает итератор на вставленный элемент;
  - `void insert(iterator pos, size_type n, const T& key);` – вставляет `n` элементов `key`, начиная с позиции, на которую указывает итератор `pos`;
  - `template <class InputIter>`
    `void insert(iterator pos, InputIter first, InputIter last);` вставляет элементы диапазона `first…last`, начиная с позиции, накоторую указывает итератор `pos`.

Контейнерные классы

```
void main()
{
   vector<int> v1(5, 0);
   int m[5] = {1, 2, 3, 4, 5};
   //добавили один элемент со значением 100 в начало вектора:
   v1.insert(v1.begin(), 100);
   // добавили два элемента со значениями 200 в начало вектора:
   v1.insert(v1.begin()+1, 2, 200);
   // добавили пять элементов со значениями из массива m в
   вектор,  // начиная с        третьего:
   v1.insert(v1.begin()+3, m, m+5);
   //добавили один элемент со значением 100 в конец вектора:
   v1.insert(v1.end(), 100);
   //печать вектора
   for(int i=0; i<v1.size(); i++)
        cout << v1[i] << " ";
}
```

- **Результат выполнения программы:**

  100 200 200 1 2 3 4 5 0 0 0 0 0 100

Контейнерные классы

- Метод `erase` имеет следующие реализации:
  - `iterator erase(iterator pos);` – удаляет элемент, на который указывает итератор `pos`.
  - `iterator erase(iterator first, iterator last);` – удаляет диапазон элементов `first…last`.

```
void main()
{
   vector<int> v1;     //создание пустого вектора
   int n, a;
   cout << "\nn?";
   cin >> n;
   for(int i=0; i<n; i++) //заполнение вектора:
   {
        cout << "?";
        cin >> a;
        v1.push_back(a);  //добавление в конец
   }
   for(i=0; i<v1.size(); i++) cout << v1[i] << " "; //печать
   вектора:
        cout << "\n";
```

Контейнерные классы

```
    //удаление первого элемента:
    vector<int>::iterator iv = v1.erase(v1.begin());
    cout << (*iv) << "\n";
    for(i=0; i<v1.size(); i++)
         cout << v1[i] << " ";
    //удаление элементов со 2 по 5:
    vector<int>::iterator v =
    v1.erase(v1.begin()+1, v1.begin()+5);
    //удаление последнего элемента:
    vector<int>::iterator iv = v1.erase(v1.end()-1);
    v1.pop_back();
}
```

Контейнерные классы

- Метод `swap(v)` служит для обмена элементов одного типа, но не обязательно одного размера: `v1.swap(v2)`. К векторам также применимы операции сравнения (`==, !=, <, <=`):

```
// пример работы с векторами, демонстрирующий использование
// методов swap(), empty (), back(), pop_back()
void main()
{
double arr[] = {1.1, 2.2, 3.3, 4.4};
//количество элементов в массиве:
int n = sizeof(arr)/sizeof(double);
//инициализация вектора массивом:
vector<double> v1(arr, arr + n);
vector<double> v2;  //пустой вектор
v1.swap(v2);        //обменять содержимое v1 и v2
while (!v2.empty())
{
    cout << v2.back() <<" ";       //вывести последний элемент
    v2.pop_back();          //и удалить его
     }
}
```

Контейнерные классы

- Рассмотрим теперь основные операции для очередей и списков.

```
//Пример работы  с очередью
typedef deque<int> tdeque;
void main()
{
    //создать объект-контейнер типа deque
    tdeque<long> q;
    cout << "\nThe first queue\nn?";
    int n;  //количество объектов в queue
    long a;
    cin >> n;
    for(int i=0; i<n; i++)
    {
        cout << "?";
        cin >> a;
        q.push(a);  //добавление в очередь
    }
    print("The first queue:\n", q);
    cout << "\nTwo first elements are deleting...\n";
```

Контейнерные классы

```
    //итератор поставили на начало очереди temp:
    tdeque::iterator p = q.begin();
    q.erase(p, p+2);    //удалили два элемента из начала
    p = q.begin();
    q.insert(p,100);    //добавили элемент в начало
    print("The first queue:\n", q);
}
```

Контейнерные классы

- Пример работы со списком :

```
void main()
{
    list<int> l;        //создание пустого списка
    list<int>::iterator i;    //итератор для работы со списком
    int x=1;
    while(x!=0)         //заполнение списка
    {
        cout<<"?";
        cin>>x;
        l.push_back(x);         //добавление в конец списка
    }

    //для печати используем функцию print(), определенную выше:
    print("list:",l);
    i = l.begin();     //итератор поставили на начало списка
    //добавление элемента со значением 100:
    l.insert(++i, 100);
    print("list:", l); //печать
    l.pop_back();      //удаление из конца
    l.pop_front();     //удаление из начала
    print("list:", l); //печать
}
```

Контейнерные классы

- **Адаптеры контейнеров**
- Специализированные последовательные контейнеры:
  - стек;
  - очередь;
  - очередь с приоритетами.

  не являются самостоятельными контейнерными классами, а реализованы на основе рассмотренных выше классов, поэтому они называются адаптерами контейнеров.

- По умолчанию для стека (заголовочный файл <stack>) прототипом является класс deque.

- Объявление stack<int> s создает стек на базе двусторонней очереди (по умолчанию). Если по каким-то причинам нас это не устраивает, и мы хотим создать стек на базе списка, то объявление будет выглядеть следующим образом:

```
stack<int, list<int> > s;
```

- Смысл такой реализации заключается в том, что специализированный класс просто переопределяет интерфейс класса-прототипа, ограничивая его только теми методами, которые нужны новому классу. Стек не позволяет выполнить произвольный доступ к своим элементам, а также не дает возможности пошагового перемещения, т. е. итераторы в стеке не поддерживаются.

Контейнерные классы

- Методы класса stack:
  - `push ()` – добавление в вершину стека;
  - `pop ()` - удаление из вершины;
  - `top ()` - просмотр элемента, находящегося в вершине стека;
  - `empty()` – проверка пустой стек или нет;
  - `size ()` – возвращает количество элементов в стеке.

- Шаблонный класс `queue` (заголовочный файл <`queue`>) является адаптером, который может быть реализован на основе двусторонней очереди (реализация по умолчанию) или списка. Класс `vector` в качестве класса-прототипа не подходит, поскольку в нем нет выборки из начала контейнера. Очередь использует для проталкивания данных один конец, а для выталкивания – другой
- Методы класса queue:
  - `push ()` – добавление в конец очереди;
  - `pop ()` – удаление из начала очереди;
  - `front ()` – просмотр элемента, находящегося в начале очереди;
  - `back()` – просмотр элемента, находящегося в конце очереди;
  - `empty ()` – проверка пустая очередь или нет;
  - `size()` – возвращает количество элементов в очереди

Контейнерные классы

- Шаблонный класс `priority_queue` (заголовочный файл <queue>) поддерживает такие же операции, как и класс queue, но реализация класса возможна либо на основе вектора (реализация по умолчанию), либо на основе списка. Очередь с приоритетами отличается от обычной очереди тем, что для извлечения выбирается максимальный элемент из хранимых в контейнере. Поэтому после каждого изменения состояния очереди максимальный элемент из оставшихся сдвигается в начало контейнера.
- Методы класса `priority_queue: push(); pop(); front(); back(); empty(); size().`

```
//создание пустой очереди с приоритетами
priority_queue<int> p;
p.push(17);
P.push(5);
P.push(400);
P.push(2500);
P.push(1);
while (!P.empty())
{ cout << P.top() << ' '; //печать максимального элемента очереди
P.pop();  }          //удаление максимального элемента
```

добавление элементов в очередь

- Результат работы программы:

  2500 400 17 5 1

Контейнерные классы

- **Алгоритмы**

- Алгоритм – это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Чтобы использовать обобщенные алгоритмы, нужно подключить к программе заголовочный файл <algorithm>.

- В табл. приведены наиболее популярные алгоритмы STL.

алгоритм	назначение
accumulate	вычисление суммы элементов в заданном диапазоне
copy	копирование последовательности, начиная с первого элемента
count	подсчет количества вхождений значения в последовательность
count_if	подсчет количества выполнений условия в последовательности
equal	попарное равенство элементов двух последовательностей
fill	замена всех элементов заданным значением
find	нахождение первого вхождения значения в последовательность
find_first_of	нахождение первого значения из одной последовательности в другой
find_if	нахождение первого соответствия условию в последовательности
for_each	вызов функции для каждого элемента последовательности

Контейнерные классы

алгоритм	назначение
`merge`	слияние отсортированных последовательностей
`remove`	перемещение элементов с заданным значением
`replace`	замена элементов с заданным значением
`search`	нахождение первого вхождения в первую последовательность второй последовательности
`sort`	сортировка
`swap`	обмен двух элементов
`transform`	выполнение заданной операции над каждым элементом последовательности

- В списках параметров всех алгоритмов первые два параметра задают диапазон обрабатываемых элементов в виде полуинтервала `[first, last)`, где `first` – итератор, указывающий на начало диапазона, а `last` - итератор, указывающий на выход за границы диапазона.

Контейнерные классы

```
// программа считывает числа в вектор, сортирует по возрастанию и
// выводит на экран:
void main()
{
   int n;
   cout << "n?";
   cin >> n;
   vector<int> v;
   int x;
   //заполнение вектора
   for(int i=0; i<n; i++)
   {
        cout << "?"; cin >> x;
        v.push_back(x);          //добавление в конец
   }
   sort(v.begin(), v.end());   //сортировка
   vector<int>::const_iterator i;
   for (i = v.begin(); i  != v.end(); ++i)
        cout << *I << " ";      //печать вектора
}
```

Контейнерные классы

- Алгоритм `sort` имеет две сигнатуры:
  - `template<class RandomAccessIt>`
    `void sort(RandomAccessIt first, RandomAccessIt last);`
    - обеспечивает сортировку элементов из диапазона `[first, last)`, причем для упорядочения по умолчанию используется операция `<`, которая должна быть определена для типа `T`. Таким образом, сортировка по умолчанию – это сортировка по возрастанию значений.
  - `template<class RandomAccessIt>`
    `void sort(RandomAccessIt first, RandomAccessIt last, Compare comp);` – позволяет задать произвольный критерий упорядочения. Для этого нужно передать через третий аргумент соответствующий предикат, то есть функцию или функциональный объект, возвращающие значение типа `bool`.
- **Функциональным объектом** называется объект некоторого класса, для которого определена единственная операция вызова функции `operator()`.
- В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык C++ .они возвращают значение типа `bool`.

Контейнерные классы

- **Шаблоны функциональных объектов для операций сравнения**

шаблон	операция
`equal_to`	`==`
`not_equal_to`	`!=`
`greater`	`>`
`less`	`<`
`greater_equal`	`>=`
`less_equal`	`<=`

- При подстановке в качестве аргумента алгоритма требуется инстанцирование этих шаблонов, например: `equal_to<int>()`.
- Если заменить в предыдущей программе вызов функции `sort` на `sort(v.begin(), v.end(), greater<int>())`, то вектор будет отсортирован по убыванию значений его элементов.
- Если сортировка выполняется для контейнера с объектами пользовательского класса, то программисту нужно самому позаботиться о наличии в классе предиката, задающего сортировку по умолчанию, а также (при необходимости) определить функциональные классы, объекты которых позволяют изменять настройку алгоритма `sort`.

Контейнерные классы

```
class person
{
   char name[20];
   int age;
   public:
   //конструктор без параметров
   person(){name[0]='\0';age=0;}
   person( char[20],int );    //конструктор с параметрами
   //функция для ввода объекта из потока
   friend istream& operator>>(istream&,person&);
   //функця для вывода объекта в поток
   friend ostream& operator<<(ostream&,const person&);
   char* get_name(){return name;} //получить имя
   int get_age(){return age;}     //получить возраст

   //перегрузка операции сравнения с константой типа int
   bool operator == (const int &x)
   {if (age==x) return true;return false;}
```

Контейнерные классы

```
    //перегрузка операции <
    bool operator < (const person & x)
    {if (age < x.age) return true; else return false;}
    //перегрузка операции <
    bool operator < (const int & x)
    {if (age < x) return true; else return false;}

    //перегрузка операции >
    bool operator < (const person & x)
    {if (age < x.age) return true; else return false;}
    //перегрузка операции >
    bool operator > (const int & x)
    {if (age > x) return true; else return false;};
};
```

Контейнерные классы

```
//функциональный класс для сортировки по возрастанию
class Comp1
{
public:
   bool operator()(person&p1,person&p2)
   {
   if(strcmp(p1.get_name(),p2.get_name())<0)return 1;
   else return 0;
   }
};

//функциональный класс для сортировки по убыванию
class Comp2
{
public:
   bool operator()(person&p1,person&p2)
   {
   if(strcmp(p1.get_name(),p2.get_name())>0)return 1;
   else return 0;
   }
```

Контейнерные классы

```
//предикат для поиска по условию "возраст < заданного возраста"
template <class t> class pred1
{
public:
   bool operator()(t& x)
   {
   if(x<year)return true; else return false;
   }
};
//предикат для поиска по условию "возраст == заданному возрасту"
template <class t> class pred2
{
public:
   bool operator()(t& x)
   {
   if(x==year)return true; else return false;
   }
};
…
```

Контейнерные классы

```
vector<person>temp;
sort(temp.begin(),temp.end(),Comp2()); //сортировка по убыванию
//найти все людей, у которых возраст меньше 18
//итератор tv1 поставили на начало вектора
vector<person>::iterator tv1=temp.begin();
//итератор tv2 поставили на конец вектора
vector<person>::iterator tv2=temp.end();
//поиск  значения, соответствующего предикату pp1
pred1<person> pp1;
cout<<"Persons which have age<18:\n";
if((tv1=find_if(temp.begin(),tv2,pp1))!=temp.end())
   {
   cout<<*tv1;
   }
   else
   {
   cout<<"\nThere is no such object\n";
   }
   cout<<"\n";
```

Контейнерные классы

```
//отсортировать контейнер по возрстанию
sort(temp.begin(),temp.end(),Comp1());
//найти количество человек, у которых возраст равен 18
//поиск  значения, соответствующего предикату pp2
pred2<person> pp2;
//итератор поставили на начало вектора
 tv1=temp.begin();
//итератор поставили на конец вектора
tv2=temp.end();
cout<<"Persons which have age=18:\n";
int count=count_if(tv1,tv2,pp2);
cout<<"Count="<<count<<endl;
cout<<"\n";
```

Контейнерные классы

Ассоциативные контейнеры

- В ассоциативных контейнерах элементы не выстроены в линейную последовательность. Они организованы в более сложные структуры (деревья), что дает большой выигрыш в скорости поиска. Поиск значений производится с помощью ключей как числового или строкового типа, так и представленных в виде объектов или указателей на них. Рассмотрим две основные категории ассоциативных контейнеров в `STL`: множества и словари.
- В **множестве** (`set`) хранятся объекты, упорядоченные по некоторому ключу, являющемуся атрибутом самого объекта. Например, множество может хранить объекты класса `Person`, упорядоченные в алфавитном порядке по значению ключевого поля `name`. Если в множестве хранятся значения одного из встроенных типов, например `int`, то ключом является сам элемент.
- Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение "меньше". Если объявить некоторое множество `set<int> si;` с опущенным вторым параметром шаблона, то по умолчанию для упорядочения членов множества будет использован предикат `less<int>`. Точно так же можно опустить второй параметр при объявлении множества `set<MyClass> s2`, если в классе `MyClass` определена операция `operator<()`.
- Для использования контейнеров типа `set` необходимо подключить заголовочный файл `<set>`.

Контейнерные классы

- Имеется три способа определить объект типа `set`:
  - `set<int> set1;` – создается пустое множество
  - `int a[5] = {1, 2, 3, 4, 5};`
  - `set<int> set2(a, a+5);` – инициализация копированием массива
  - `set<int> set3(set2);` – инициализация другим множеством
- Для вставки элементов в множество можно использовать метод `insert`(), для удаления – метод `erase`(). Также к множествам применимы общие для всех контейнеров методы. Во всех ассоциативных контейнерах есть метод `count`(), возвращающий количество объектов с заданным ключом. Так как и в множествах, и в словарях все ключи уникальны, то метод `count`() возвращает либо 0, если элемент не обнаружен, либо 1. Для множеств библиотека содержит некоторые специальные алгоритмы, в частности, реализующие традиционные теоретико-множественные операции:
  - `includes` выполняет проверку включения одной последовательности в другую, результат равен `true` в том случае, когда каждый элемент последовательности `[first2,last2)` содержится в последовательности [`firstl, lastl)`.
  - `set_intersection` создает отсортированное пересечение множеств, то есть множество, содержащее только те элементы, которые одновременно входят и в первое, и во второе множество.
  - `set_union` создает отсортированное объединение множеств, то есть множество, содержащее элементы первого и второго множества без повторяющихся элементов

Контейнерные классы

```
int main()
{
   const int N = 5;
   int a1[N] = {1,2,3,4,5};
   int a2[N] = {3,4,5,6,7};
   typedef set<int> SetS;//определяем тип множество с элементами int
   //создаем множество A и инициализируем его элементами массива a1
   SetS A(a1, a1 + N);
   //создаем множество B и инициализируем его элементами массива a2
   SetS B(a2, a2 + N);
   //печать множеств с помощью функции print(), описанной выше
   print(A); print(B);
   //множество для пересечения A и B
   SetS prod;
   //множество для объединения A и B
   SetS sum;
```

Контейнерные классы

```
    //пересечение
    //inserter() – итератор, который добавляет новый
    //элемент в произвольное место контейнера
    set_intersection(A.begin(), A.end(), B.begin(),
    B.end(),inserter(prod, prod.begin()));
    print(prod); //печать пересечения
    //объединение
    set_union(A.begin(). A.end(), B.begin(),
    B.end(),inserter (sum, sum.begin()));
    print(sum); //печать объединения
    //включение
    if (includes(A.begin(). A.end(), prod.begin(),
    prod.end())) cout << "Yes" <<endl;
    else cout <<"No"<< endl;
    return 0;
}
```

Контейнерные классы

- **Словарь** (`map`) можно представить себе как своего рода таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения.

- И в множествах, и в словарях все ключи являются уникальными (только одно значение соответствует ключу). Мультимножества (`multiset`) и **мультисловари** (`multimap`) аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

- В определении класса `map` используется тип `pair`, который описан в заголовочном файле <`utility`>. Шаблон `pair` имеет два параметра, представляющих собой типы элементов пары. Первый элемент пары имеет имя `first`, второй – `second`. В этом же файле определены шаблонные операции `==, !=, <, >, <=, >=` для двух объектов типа `pair`.

- Шаблон словаря имеет три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение "меньше".

Контейнерные классы

```
#include <iostream>
#include <map>
#include "person.h"
using namespace std;

typedef multimap<int,person> tmap;
void main()
{
   tmap m1;
   tmap::iterator mi1;
   int n;
   cout<<"\nn-?";
   cin>>n;
   person a;
   for(int i=0;i<n;i++)
   {    cout<<"?";      cin>>a;
   m1.insert(make_pair(i,a)); } //вставка элемента в словарь
   //печать словаря с помощью функции print(), определенной выше
   print("M1:",m1);
}
```

Контрольные вопросы

- 1) Что такое контейнеры и для чего они нужны?
- 2) Что такое алгоритмы и как они используются?
- 3) Какие виды контейнеров существуют?
- 4) Что такое итератор? Перечислите виды итераторов.
- 5) Какие существуют последовательные контейнеры и чем они отличаются друг от друга?
- 6) Как можно определить объект для последовательного контейнера(несколько способов)?
- 7) Алгоритмы и итераторы для работы с каждым контейнером.
- 8) Как работает алгоритм сортировки?
- 9) Что называется функциональным объектом?
- 10) Что такое ассоциативные контейнеры? Какие они бывают и чем отличаются друг от друга?
- 11) Опишите каждый контейнер(map, vector, list, queue), укажите различия между ними и основные алгоритмы и итераторы для работы с ними.

Обработка исключительных ситуаций

Обработка исключительных ситуаций

- **Понятие исключительной ситуации**
- При выполнении операторов программы может возникнуть ошибочная ситуация (деление на 0, обращение к элементу массива с несуществующим индексом и т. п.). В этом случае программа может предпринять одно из следующих действий:
  - прервать выполнение программы;
  - возвратить код ошибки;
  - вывести сообщение об ошибке и вернуть программе некоторое значение, которое позволит ей продолжить работу.

Для решения таких проблем в C++ были введены средства генерации и обработки исключений. *Исключительная ситуация*, или исключение – это возникновение непредвиденного или аварийного события, которое может порождаться некорректным использованием аппаратуры.

- Исключения позволяют логически разделить вычислительный процесс на две части – обнаружение аварийной ситуации и ее обработка.
- Другое достоинство исключений состоит в том, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение, параметры или глобальные переменные, поэтому интерфейс функций не раздувается. Это особенно важно, например, для конструкторов, которые по синтаксису не могут возвращать значение

Обработка исключительных ситуаций

- **Синтаксис исключений**
- Исключение может генерироваться только в контролируемом блоке:

```
try {…}  //контролируемый блок
```

- Все функции, прямо или косвенно вызываемые из `try`-блока, также принадлежат контролируемому блоку.

- Генерация (порождение) исключения происходит по ключевому слову `throw`, которое употребляется либо с параметром, либо без него:

```
throw [выражение ]; //генерация исключения
```

- Тип выражения, стоящего после `throw`, определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, и происходит поиск соответствующего обработчика и передача ему управления. Как правило, исключение генерируется не непосредственно в `try`-блоке, а в функциях, прямо или косвенно в него вложенных.

Обработка исключительных ситуаций

- Обработчики исключений начинаются с ключевого слова `catch`, за которым в скобках следует тип обрабатываемого исключения. Они должны располагаться непосредственно за `try`-блоком. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений.
- Существует три формы записи:
  - `catch`(тип имя) {тело обработчика}
  - `catch`(тип) {тело обработчика}
  - `catch(...) {`тело обработчика}
- Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий – например, вывода информации об исключении.
- Вторая форма не предполагает использования информации об исключении, играет роль только его тип. Многоточие вместо параметра обозначает, что обработчик перехватывает все исключения. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа следует помещать после всех остальных.
- Таким образом, для того, чтобы определить исключительную ситуацию, надо:
  - выделить контролируемый блок;
  - предусмотреть генерацию исключений в этом блоке;
  - разместить обработчики исключений после `try`-блока.

Обработка исключительных ситуаций

```
try    //контролируемый блок
{
   …
   if (ошибка) throw E(); //генерация исключения
   …
}
catch (H) //обработка исключения
{
…
}
```

- В этом пример обработчик вызывается, если:
  - `E` и `H` одного типа,
  - `H` – базовый класс `E` (с наследованием `public`),
  - `E` и `H` – указатели или ссылки и для них выполняется 1 или 2.

Обработка исключительных ситуаций

- **Механизм обработки исключений**
- Обработка исключения начинается с появления ошибки. Функция, в которой она возникла, генерирует исключение (`throw` с параметром, определяющим вид исключения). Параметр может быть константой, переменной или объектом и используется для передачи информации об исключении его обработчику.
- Отыскивается соответствующий обработчик исключения (`catch` с таким же параметром, как у `throw`) и ему передается управление.
- Если обработчик исключения не найден, вызывается стандартная функция `terminate`, которая вызывает функцию `abort`, аварийно завершающую текущий процесс. Можно установить собственную функцию завершения процесса.
- Когда с помощью `throw` генерируется исключение, функции библиотеки C++ выполняют следующие действия:
  - создают копию параметра `throw` в виде статического объекта, который существует до тех пор, пока исключение не будет обработано;
- в поисках подходящего обработчика раскручивают стек, вызывая деструкторы локальных объектов, выходящих из области действия;
- передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.
- *Раскручиванием стека* называется процесс освобождения памяти из-под локальных переменных и возврата управления вызывающей функции. Когда функция завершается, происходит естественное раскручивание стека. Тот же самый механизм используется и при обработке исключений.

Обработка исключительных ситуаций

- Обработчик считается найденным, если тип объекта, указанного после `throw`:
  - тот же, что и указанный в параметре `catch` (параметр может быть записан в форме `T`, `const T`, `T&` или `const T&`, где `T`— тип исключения);
  - является производным от указанного в параметре `catch` (если наследование производилось с ключом доступа `public`);
  - является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре `catch`.

- Таким образом, обработчики производных классов следует размещать до обработчиков базовых, поскольку в противном случае им никогда не будет передано управление. Обработчик указателя типа `void` автоматически скрывает указатель любого другого типа, поэтому его также следует размещать после обработчиков указателей конкретного типа.

Обработка исключительных ситуаций

```
//демонстрационная программа
class A   //демонстрационный класс
{
public:
   A(){cout<<"Constructor of A\n";}  //конструктор без параметров
   ~A(){cout<<"Destructor of A\n";}  //деструктор
};
void f1()
{
   A a;         //создали объект класса A
   throw 1;     //генерируется исключение типа int
}
int main()
{
   try          //контролируемый блок
   {
        f1(); //выход по исключительной ситуации throw 1
   }
   //обработчик исключительной ситуации
   catch(int){ cout<<"catch of int";}
   return 0;
}
```

Обработка исключительных ситуаций

- **Результатом выполнения этой программы будет вывод сообщений:**

```
Constructor of A
Destructor of A  (!!!)
catch of int
```

- Здесь надо обратить внимание на то, что после порождения исключения был вызван деструктор локального объекта(!!!), хотя управление из функции `f1` было передано обработчику, находящемуся в функции `main`

Обработка исключительных ситуаций

- **Неперехваченные исключения**
- Если исключение сгенерировано, но не перехвачено, вызывается стандартная функция `terminate`(), по умолчанию эта функция вызывает функцию `abort`(). Результатом выполнения функции `abort`()будет окно с сообщением об аварийном завершении программы.
- Вызов функции `abort`()можно заменить вызовом своего обработчика с помощью функции `set_terminate`().

```
void other_abort()
{
cerr<<"\nProgram is terminated";
exit(1);
}
int main()
{
set_terminate(other_abort); //замена функции abort()
      //обработчиком other_abort()
…
}
```

Обработка исключительных ситуаций

- **Классы исключений**
- Средствами C++ можно генерировать исключения любого встроенного типа, а также создавать специальные классы исключений и использовать в `throw` либо объекты этих классов, либо анонимные экземпляры.
- Использование собственных классов исключений предпочтительнее применения стандартных типов данных.

```
class MathError //базовый класс обработки ошибок
{
virtual void ErrorProc(){…}; //обработка ошибки
};
class Overflow : public MathError //класс ошибки переполнения
{
virtual void ErrorProc(){…}; //обработка ошибки
};
class ZeroDevide : public MathError //класс ошибки деление на 0
{
virtual void ErrorProc(){…}; //обработка ошибки
};
```

Обработка исключительных ситуаций

```
…
try
{
   …
   //генерируется исключение класса ZeroDevide
   if (деление на 0) throw ZeroDevide();
   …
   //генерируется исключение класса Owerflow
   if (переполнение) throw Owerflow();
   …
}
catch (MathError& me)
{
   me.ErrorProc();
}
```

- Единственный обработчик `catch` принимает объект базового класса и, на этапе выполнения будет обрабатывать как исключения типа `MathError`, так и любого производного типа. Благодаря полиморфизму, каждый раз будет вызываться версия метода `ErrorProc()`, соответствующая данному типу исключения.

Обработка исключительных ситуаций

- **Спецификации исключений**
- В заголовке функции можно задать список типов исключений, которые она может прямо или косвенно порождать. Этот список приводится в конце заголовка и предваряется ключевым словом `throw`.

```
void Func(int a) throw (ex_1, ex_2);
```

- Такое объявление означает, что функция `Func` может сгенерировать только исключения `ex_1, ex_2` и исключения, являющиеся производными от этих типов. Заголовок является интерфейсом функции, поэтому такое объявление дает пользователям функции определенные гарантии. Это очень важно при использовании библиотечных функций, так как определения функций в этом случае не всегда доступны.
- Если функция сгенерирует исключение, не соответствующее спецификации исключений, то система вызовет обработчик `unexpected()`, который может попытаться сгенерировать свое исключение, и если оно не будет противоречить спецификации, то продолжится поиск подходящего обработчика, в противном случае вызывается функция `terminate`().
- Вместо функции `unexpected` () можно установить собственную функцию, для этого нужно воспользоваться функцией `set_unexpected`().
- Если спецификация исключений задана в виде `throw`(), это означает, что функция вообще не генерирует исключений.

Обработка исключительных ситуаций

- **Исключения в конструкторах**
- Исключения предоставляют единственную возможность передать информацию об ошибке, случившейся в процессе создания нового объекта.

```
//класс вектор
class Vect
{
private:
     int* p; // указатель на массив целых чисел, составляющих
             // вектор
     char size;                  //размер вектора
public:
     Vect(char);                 //конструктор с параметром
     ~Vect(){delete [] p;}       //деструктор
     int& operator [] (int i)    //операция []
     {return p[i];}
     void Print();               //печать вектора
};
```

Обработка исключительных ситуаций

```
//реализация методов класса Vect
Vect::Vect(char n)
{
   size = n;
   p = new int[n]; //выделение памяти
   if(!p) throw "error in constructor"; //генерирует исключение при
                                        //ошибке выделения памяти
   for(int i=0;i<n;i++)
        p[i] = 0; //заполнение вектора нулевыми элементами
}
void Vect::Print()
{
   for(int i=0;i<size;i++)
   cout<<p[i]<<" ";
   cout<<endl;
}
```

Обработка исключительных ситуаций

```
int main()
{
   try
   {
        Vect a(3); //создаем вектор из 3 элементов
        a[0]=0;a[1]=1;a[2]=2;
        a.Print();

        Vect b(200); //создаем вектор из 200 элементов
        b[10] = 5;   //ошибочная ситуация, т. к. объект не создан
        b.Print();
   }
   catch(char *msg)
   {
        cerr<<"Error:"<<msg<<endl;
   }
   return 0;
}
```

Обработка исключительных ситуаций

- **Исключения в деструкторах**
- Если деструктор, вызванный во время раскрутки стека, попытается завершить свою работу при помощи исключения, то система вызовет функцию `terminate`(). На этапе отладки программы это допустимо, но в готовом продукте появление таких сообщений должно быть исключено.
- Следовательно, ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы. Чтобы выполнить это требование:
- нельзя генерировать исключения в теле деструктора с помощью `throw`.
- если удаление объекта связано с вызовом других функций, относительно которых нет гарантий отсутствия генерации исключений, то рекомендуется объединить эти действия в некотором методе, например `Destroy`(), и вызывать данный метод с использованием блока `try/catch`.

```
T::Destroy()
{ // код, который может генерировать исключения
}
T::~T()
{   try
    {    Destroy();    }
    catch(…) { … }
}
```

Контрольные вопросы

- 1) Что может сделать программа при возникновении ошибки?
- 2) Что такое исключение?
- 3) С помощью каких ключевых слов работает обработка исключения?
- 4) Перечислите основные этапы обработки исключений.
- 5) Спецификация исключений.
- 6) Исключения в конструкторах и деструкторах.