Основы программирования на языке С++

Темы занятий

- Ввод/вывод данных
- Логический тип bool
- Работа со строками
- Выделение памяти

Занятие 1. Темы

- С и С++
- Ввод/вывод данных
- Логический тип данных
- Ссылки
- Цикл for для массивов
- Работа со строками
- Выделение памяти
- Работа с текстовыми файлами
- Работа с двоичными файлами
- Шаблоны функций

С и С++

- Язык С++, как следует из его названия, является инкрементом (увеличением) С.
- Язык С++ включает в себя все базовые конструкции языка С, но также добавляет новые возможности.
- Операторы цикла, ветвления, объявление переменных в С++ аналогичны тем, которые есть в С.
- Подключение библиотек с помощью директивы #include также используется в С++, однако там у готовых библиотек нет расширения .h
- В С++ добавлен новый тип данных - bool.
- В С++ добавлена удобная обработка ошибок времени исполнения.
- Также в С++ реализована другая парадигма создания программ «объектно-ориентированное программирование»

Вывод/вывод данных

- Для операций ввода/вывода в C++ используется библиотека **iostream**
- Вывод данных осуществляется с помощью оператора вставки **<<**
- Ввод данных осуществляется с помощью оператор извлечения **>>**
- Операторам вставки и извлечения нужно указать куда и откуда вставлять данные.
- Чтобы вывод данных был на экран нужно указать вставку в объект **cout**.
- Чтобы ввод данных был с клавиатуры нужно указать извлечение из объекта **cin**.
- Объекты **cin** и **cout** находятся в пространстве имен **std** и для удобства работы с ними следует указать на использование этого пространства с помощью команды **using**.

Пример:

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

int main()
{
    cout << "Hello world!" << endl;
}
```

Вывод данных

- Вывод данных осуществляется с помощью оператора вставки (вывода) **<<**
- Формат оператора вставки: *куда << откуда*
- Для вывода информации на экран в левой части указывается специальный объект cout.

  Пример:

  ```
  int a=5;
  cout << a;
  ```

- В одной строке можно указать несколько операций вставки, они будут выполнены слева направо.

  Пример:

  ```
  int a=5, b=3, c;
  cout << a << " + " << b << " = " << c;
  ```

- Для перехода на новую строку используется вставка **\n** либо спецслова **endl**

Ввод данных

- Для ввода данных используется оператор извлечения (ввода) **>>**
- Формат оператора извлечения: *откуда >> куда*
- Для ввода информации с клавиатуры в левой части указывается специальный объект cin.

  Пример:

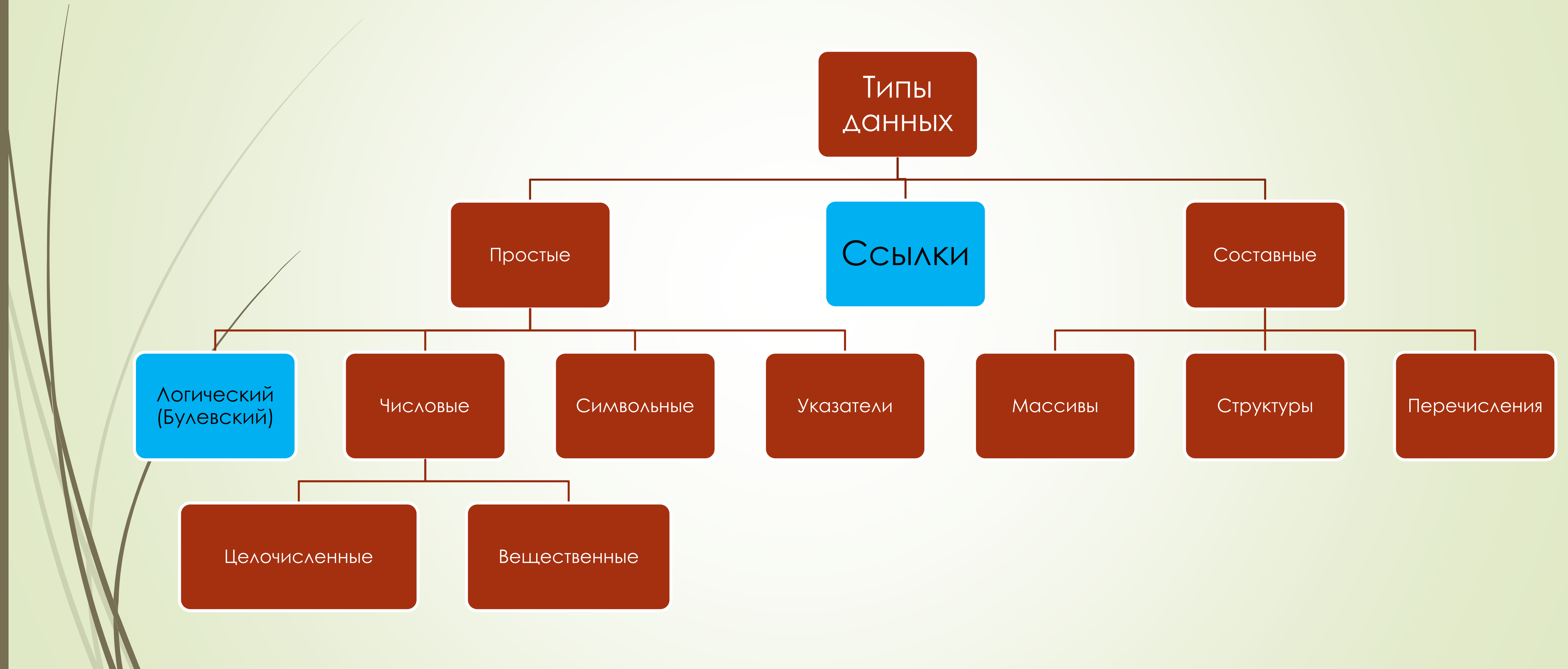
  ```
  int a;
  cin >> a;
  ```

- В одной строке можно указать несколько операций извлечения, они будут выполнены слева направо.

  Пример:

  ```
  int a, b
  float f;
  cin >> a >> b >> f;
  ```

Типы данных языка C++



Тип данных bool

- В языке C++ добавлен новый тип данных, логический – bool.
- Тип bool может принимать только два значения: 1 (true) и 0 (false)
- Все операции сравнения двух величин — вещественных и целых переменных или константы с переменной возвращают в качестве результата тип bool.
- Тип bool можно передавать в качестве условия в операторы цикла и ветвления. Если значение переменной типа bool равно 0 (false) значит условие не выполняется. Иначе выполняется.

Пример:

```
int n;
bool b=true;

while (b)
{
        cin >> n;
         if (n==0)
                b=false;
}
```

Ссылки

- В языке программирования C++ ссылка (англ. reference) — это простой ссылочный тип, менее мощный, но более безопасный, чем указатель.
- С помощью ссылки задаётся альтернативный идентификатор (имя) для уже созданного объекта.
- Ссылка — это, по сути, указатель, который жёстко привязан к области памяти, на которую он указывает, и которому не нужна операция разыменования.
- Ссылки нельзя объявлять без привязки к переменной (то есть не инициализировав при объявлении). После объявления ссылки её невозможно привязать к другой переменной.
- Для определения ссылки используется символ амперсанд (&)

Пример:

```
int i = 10;
int& reff = i;
cout << i << endl;
cout << reff << endl;
reff = 55;
cout << i;
```

Передача параметров по ссылке

- Если нужно передать в функцию переменные, значения которых следует изменить в так чтобы это изменение затронуло сами исходные переменные, то следует использовать передачу указателя либо использовать ссылки.

```
void changed(int& reff)  //в качестве параметра будет ссылка
{
   reff = reff + 5;
}
int main()
{
   int i = 5;
   cout << "i before = " << i << endl;
   changed(i);
   cout << "i after = " << i << endl;
}
```

Важно отличать ссылки от оператора взятия адреса &. Оператор взятия адреса используется для уже созданного объекта с целью получить его адрес. Ссылка это только задание альтернативного имени объекта.

Цикл for по массиву

- Начиная с версии C++11 в языке появилась новая конструкция for, которая позволяет проходить по всем элементам массива без использования переменной цикла и без необходимости указывать длину массива.
- Такой цикл for имеет следующий синтаксис: **for (элемент : массив)**. На каждом проходе цикла в элемент записывается копия очередного значения из массива, с которым затем можно работать в теле цикла.
- Объявляемый элемент должен быть того же типа, что и элементы массива

```
int arr[] = {1,2,3,4,5};

//обычный цикл for
for (i=0; i<5; i++)
     cout << arr[i] << " ";

//новый цикл for
   for (int n: arr)
     cout << n << " ";
```

Можно использовать ключевое слово **auto**, вместо указания конкретного типа данных, тогда тип будет определен по типу элементов массива.

```
int arr[] = {1,2,3,4,5};
char chArr[] = {‘a’,’b’,’c’};

for (auto n: arr)
     cout << n << " ";

for (auto n: chArr)
     cout << n << " ";
```

Работа с исходными элементами

```
int arr[] = {1,2,3,4,5};

for (auto &n: arr)
      n = 6;

for (i=0; i<5; i++)
      cout << arr[i] << " ";
```

Работа со строками. Объект string

- Для удобства работы со строками в C++ используется объект string.
- Объект string также содержится внутри пространства имен std.
- Для переменные типа string можно использовать операции присвоения и сложения.
- Оператор сложения (конкатенации) + соединяет две строки в одну.

Пример:

```
string str1, str2, str3;
str1 = "Hello";
str2 = "World";
str3 = str1+str2;
cout << str3;
```

- С помощью функции length() можно узнать длину строки.

```
cout << str3.length()
```

Выделение памяти. Оператор new

- В C++ для выделения памяти используется оператор **new**.
- С помощью оператора **new** можно динамически выделять память для массива. Для этого после слова new следует указать тип данных и в квадратных скобках сколько ячеек такого типа нужно выделить. Оператор возвращает адрес первого элемента.

  Пример:

```
int size;
int* arr;

cin >> size;
arr = new int[size];
```

- Память, выделенная при помощи new, должна быть освобождена при помощи оператора delete. Существует два варианта этого оператора: delete[] для массивов, delete — для единичных объектов.

  Пример: delete[] arr;

- В отличие от функции realloc в языке Си, при помощи оператора new[] невозможно напрямую перераспределить уже выделенную память. Для увеличения или уменьшения размера блока памяти нужно выделить новый блок нужного размера, скопировать данные из старой памяти и удалить старый блок

Работа с текстовыми файлами. Запись

- Для работы с файлами нужно подключить библиотеку fstream
- Для работы с текстовыми файлами в языке C++ также используются операторы вставки (<<) и извлечения (>>), но теперь они применяются к другим потокам.
- В C++ так же как в C работа с файлом состоит из трех частей:

  1)открытие, 2)работа, 3)закрытие
- Для того чтобы открыть файл для записи нужно создать объект типа ofstream и вызвать у этого объекта метод open, передав ему в качестве параметра путь и имя файла.
- Для записи к созданному объекту следует применять оператор вставки
- Для закрытия файла нужно вызвать метод close созданного объекта.

Пример:

```
int n=35;
ofstream fileOut;
fileOut.open("filename.txt");
fileOut << "Hello world! " << n << endl;
fileOut.close();
```

Создание и открытие в одной строке:

```
int n=35;
ofstream fileOut("fileshort.txt");
fileOut << "Hello world! " << n << endl;
fileOut.close();
```

Режим доступа к файлу

Вторым параметром функции open можно передать режим доступа, чтобы определить что будет при открытии файла.

Режим	Значение
ios_base::trunc	Создание пустого файла для записи. Если файл с таким именем уже существует его содержимое стирается.
ios_base::app	Добавление данных в конец файла. Если файл не существует то он создается.
ios_base::binary	Открыть файл в двоичном режиме

По умолчанию используется режим trunc, то есть при открытии файла для записи старое содержимое автоматически стирается.

Работа с текстовыми файлами. Чтение

- Для того чтобы открыть файл для чтения нужно создать объект типа ifstream и вызвать у этого объекта метод open, передав ему в качестве параметра путь и имя файла.
- Для чтения из файла к созданному объекту следует применять оператор извлечения
- Для закрытия файла нужно вызвать метод close созданного объекта.

Пример:

```
int n;
ifstream fileIn("filename.txt");
fileIn >> n;
fileIn.close();
cout << "n from file = " << n << endl;
```

- При чтении из файла текстовых данных оператор извлечения читает строку только до пробела. Если нужно прочесть всю строку целиком, то нужно использовать другие функции чтения, например getline:

Пример:

```
string str;
ifstream fileIn("filename.txt");
getline(fileIn, str);
fileIn.close();
cout << "str from file = " << str << endl;
```

Работа с двоичными файлами. Запись

- Для работы с двоичными файлами используются массивы, откуда данные считываются или записываются.
- Для того чтобы открыть файл для записи нужно создать объект типа ofstream и вызвать у этого объекта метод open, передав ему в качестве параметра путь и имя файла. Также нужно указать на двоичный режим доступа ios_base::binary
- Для закрытия файла нужно вызвать метод close созданного объекта.
- Для записи в файл нужно вызвать метод write созданного объекта.
- Метод write записывает данные в файл как последовательность байт.
- Метод write принимает два параметра: массив типа char и длину (сколько ячеек этого массива нужно записать.

Пример:

```
char buff[]={65,66,67};
ofstream fileBin;
fileBin.open("fileBin.txt", ios_base::binary);
fileBin.write(buff,3);
fileBin.close();
```

Если нужно записать массив отличный от char то можно воспользоваться операцией приведения типа:

```
int buffInt[]={1111, 2222, 3333};
ofstream fileBin("fileBin.txt", ios_base::binary);
fileBin.write((char*)buffInt,sizeof(int)*3);
fileBin.close();
```

Работа с двоичными файлами. Чтение

- Для того чтобы открыть двоичный файл для чтения нужно создать объект типа ifstream и вызвать у этого объекта метод open, передав ему в качестве параметра путь и имя файла. Также нужно указать на двоичный режим доступа ios_base::binary
- Для закрытия файла нужно вызвать метод close созданного объекта.
- Для чтения данных из файла нужно вызвать метод read созданного объекта.
- Метод read считывает данные из файла как последовательность байт.
- Метод read принимает два параметра: массив типа char, в который будут записаны данные из файла и длину (сколько ячеек этого массива нужно заполнить).

Пример:

```
char buffOut[10];
ifstream fileBinIn;
fileBinIn.open("fileBin.txt", ios_base::binary);
fileBinIn.read(buffOut, 3);
fileBinIn.close();
```

Если нужно прочитать данные отличные от char то можно воспользоваться операцией приведения типа:

```
int buffOutInt[10];
ifstream fileBinIn("fileBin.txt", ios_base::binary);
fileBinIn.read((char*)buffOutInt, sizeof(int)*3);
fileBinIn.close();
```

Функции

- В C++ можно создать несколько функций с одинаковыми именами, если они различаются списком своих параметров, так чтобы при вызове функции можно было однозначно определить какую версию функции следует вызывать.

Пример:

```
int divide(int a, int b)
{
    return a/b;
}
float divide(float a, float b)
{
    return a/b;
}
void main()
{
        cout << divide(5, 2) << endl;
}
```

Шаблоны функций

- Если есть несколько функций одинаковых алгоритмически, которые различаются только типом параметров и локальных переменных, то можно вместо этих функций написать шаблон, и уже в момент вызова определять тип параметров.
- Шаблоны предоставляют краткую форму записи участка кода, но их использование не сокращает исполняемый код, так как для каждого набора параметров компилятор создаёт отдельный экземпляр функции.
- Шаблон функции начинается с ключевого слова **template**, за которым в угловых скобках следует список параметров. После следует объявление функции с использованием этих параметров.
- Ключевое слово **typename** говорит о том, что в шаблоне будет использоваться встроенный тип данных, такой как: int, double, float, char и т. д.

Пример:

```
template <typename T>
T divide(T a, T b)
{
    return a/b;
}
```

При вызове функции на основе шаблона следует после имени функции в угловых скобках указать тип, который будет подставлен вместо T:

```
cout << divide<float>(5,2) << endl;
```

Занятие 2. Темы

- Классы и объекты
- Поля и методы класса
- Модификаторы доступа
- Создание объектов из класса
- Специальные методы: Конструктор/Деструктор
- Обращение объекта к самому себе (this)
- Статические данные класса
- Многофайловая программа
- Пространства имен

Процедурно-ориентированные языки

- C, Pascal, FORTRAN и другие сходные с ними языки программирования относятся к категории процедурных языков.
- Каждый оператор процедурного языка является указанием компьютеру совершить некоторое действие, например принять данные от пользователя, произвести с ними определенные действия и вывести результат этих действий на экран.
- Программы, написанные на процедурных языках, представляют собой последовательности инструкций, последовательность функций

Недостатки процедурного подхода:

- Процедурный подход не позволяет в достаточной степени упростить сложные программы
- Неконтролируемый доступ к данным. Есть неограниченность доступа функций к глобальным данным.
- Разделение данных и функций плохо отображает картину реального мира.

Объектно-ориентированное программирование

- Идея ООП - объединить данные и действия, производимых над этими данными, в единое целое.
- Три кита объектно-ориентированного подхода:
  1. Инкапсуляция
  2. Наследование
  3. Полиморфизм
- В C++ вводится новое понятие – **класс**. Класс содержит в себе:
  1. поля (данные)
  2. методы (функции для работы с данными)
- **Класс** является по сути пользовательским типом данных. После объявления класса, можно создавать переменные типа этого класса. Такая созданная переменная называется **объектом** (экземпляром) класса.
- Используя имя объекта, можно обращаться к его **полям** и **методам**.
- Типичная программа на языке C++ состоит из совокупности объектов, взаимодействующих между собой посредством вызова методов друг друга.

Класс и объект

Объявление класса:

```
class firstClass {
  private:
   int info;

   public:
   void setInfo(int n) {
       info = n;
   }
   void printInfo() {
       cout << "info = " << info << endl;
   }
};
```

- Доступ к полям и методам класса возможен только через конкретный объект этого класса.
- Для того чтобы получить доступ, необходимо использовать операцию точки (.), связывающую метод или поле с именем объекта.

Создание объекта из класса и вызов его методов:

```
void main()
{
    firstClass fc;
    fc.setInfo(23);
    fc.printInfo();
}
```

Сокрытие данных (инкапсуляция)

- Ключевой особенностью объектно-ориентированного программирования является возможность сокрытия данных.
- Данные и методы заключены внутри класса и защищены от несанкционированного доступа извне.
- Если необходимо защитить какие-либо данные, то их помечают ключевым словом **private**. Такие данные доступны только внутри класса.
- Данные, описанные с ключевым словом **public**, напротив, доступны за пределами класса.

```
class secondClass
{  private:
   int privInf;

   public:
   int publnf;
};
```

```
void main()
{
    secondClass sc;
    sc.privInf = 5;       //ошибка! нет доступа
    sc.publnf =12;
}
```

Конструкторы класса

- Конструктор — это метод класса, выполняющийся автоматически в момент создания объекта.
- С помощью конструктора удобно инициализировать поля объекта автоматически в момент его создания.
- У конструктора есть несколько особенностей, отличающих его от других методов класса.
- Имя конструктора в точности совпадает с именем класса. Таким образом, компилятор отличает конструкторы от других методов класса.
- У конструктора не существует возвращаемого значения. Конструктор автоматически вызывается системой, и, следовательно, не существует вызывающей программы или функции, которой конструктор мог бы возвратить значение.
- Конструкторы бывают двух видов:
  1. Конструктор по умолчанию (не принимает параметров)
  2. Параметризованный конструктор (со списком параметров)

Использование конструктора

```
class firstClass {
 private:
  int info;

  public:
  firstClass(){
      info = 0;
  }

  void setInfo(int n) {
      info = n;
  }
  void printInfo() {
      cout << "info = " << info << endl;
  }
};
```

```
void main()
{
    firstClass fc;
    fc.printInfo();
}
```

Деструктор класса

- Деструктор — это метод класса, выполняющийся автоматически в момент уничтожения объекта.
- В деструкторе обычно производится освобождение памяти, которая была выделена в процессе работы объекта.
- Имя деструктора совпадает с именем конструктора, но в начале имени деструктора добавляется также специальный символ ~ (тильда), чтобы их различать.
- У деструктора также не существует возвращаемого значения. Деструктор автоматически вызывается системой, и, следовательно, не существует вызывающей программы или функции, которой он мог бы возвратить значение.

Использование деструктора

```
class firstClass {
 private:
  int* mas;

  public:
  firstClass(int n)
  {
      mas = new int[n];
  }

  ~firstClass()
  {
      delete[] mas;
  }
};
```

```
void main()
{
    firstClass fc(10);
}
```

Обращение объекта к самому себе

- С помощью специального указателя **this** объект может обратиться к самому себе.
- Также как в случае с указателем на структуру обращение к полям и методам объекта по указателю осуществляется с помощью оператора ->
- Одно из возможных применений указателя this это разрешение неоднозначности контекста, когда входящий параметр метода назван так же, как поле данных класса.
- Также с помощью this объект может передать себя в качестве параметра в функцию.

```
class Car {
 private:
 int number;
 string vendor;

 public:
 Car(int number, string vendor) {
  this->number = number;
  this->vendor = vendor;
  }

 void show() {
   cout << "number: " << number << "\nvendor: “ << vendor;
 }
};
```

Статические данные класса

- Если поле или метод класса описаны с ключевым словом **static**, то для всех объектов данного класса будет существовать одно такое общее поле или метод.
- Значение static поля будет одинаковым для всех объектов данного класса.
- Переменная типа static инициализируется вне класса, потому что не относится к объекту.
- Статические данные класса полезны в тех случаях, когда необходимо, чтобы все объекты включали в себя какое-либо одинаковое значение.

Класс

```
class Car {
  private:
      static int total;
      int number;
      string vendor;

  public:
  Car() {
    total++;
  }

  void howMany() {
     cout << "total " << total << " cars"<<endl;
  }
};
```

Использование static переменной

```
int Car::total = 0;

void main()
{
   Car c1,c2,c3;
   c1.howMany();
}
```

Многофайловая программа

- Программу C++ использующую классы и объекты также можно разделить на несколько файлов. В этом случае:
  1. В заголовочный файл выносится объявление класса (поля и объявления методов).
  2. В файле исходного кода подключается заголовочный файл.
  3. В файле исходного кода пишут определения методов, но перед именем метода через :: (двойное двоеточие) следует добавить имя класса, к которому этот метод относится.
  4. В файле где происходит создание объектов из класса и использование его методов также необходимо подключить заголовочный файл с объявлением класса.

Пример многофайловой программы

classFile.h

```
class sepClass
{
private:
    int info;
public:
    sepClass();
    int getInfo();
    void setInfo(int i);
    void printInfo();
};
```

classFile.cpp

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

    sepClass::sepClass(){
        info = 0;
    }

    int sepClass::getInfo(){
        return info;
    }

    void sepClass::setInfo(int i){
        info = i;
    }

    void sepClass::printInfo(){
        cout << "info = " << info << endl;
    }
```

Main.cpp

```
#include "classFile.h“

int main()
{
    sepClass sc;
    sc.printInfo();
}
```

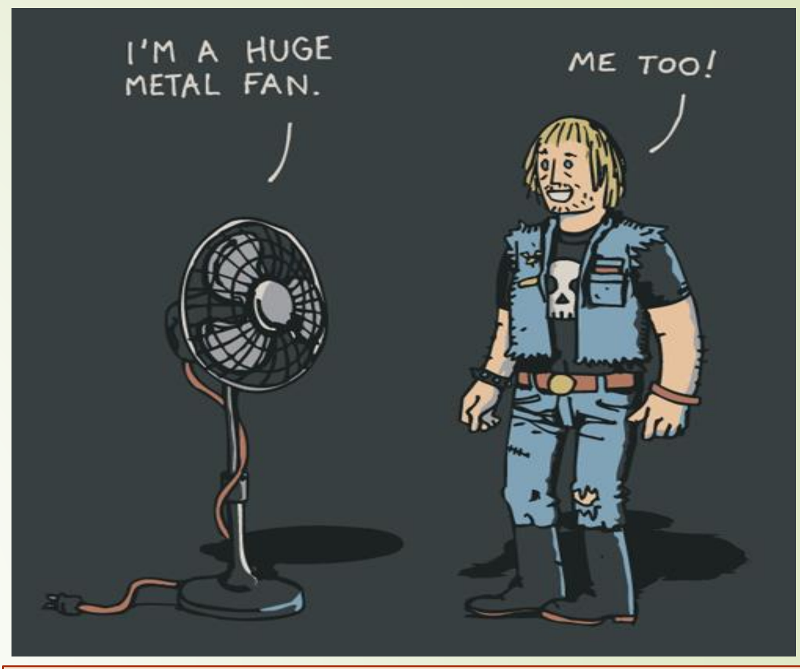
Пространство имен (namespace)

- Пространство имен это именованная область файла ограниченная фигурными скобками.
- Переменные и другие элементы программы, объявленные внутри, называются членами пространства имен.
- Часть кода, находящаяся вне пространства имен, не имеет доступа к его членам обычным способом. Пространство имен делает их невидимыми.
- Чтобы иметь доступ к элементам пространства имен извне, необходимо при обращении к ним использовать название этого пространства. Есть два способа это сделать:
  1. Перед именем элемента написать название пространства имен и оператор разрешения контекста (::) (std::cout << “Hello world”;)
  2. Использовать директиву using (using namespace std;)
- Область действия пространства имен можно сузить до блока, используя его, например, внутри функции.
- Пространство имен может быть безымянным.

Пример использования пространства имен

```
namespace people {
class Fan{
private:
 string name;
 int age;
 string passion;

public:
Fan(string name, int age, string passion)
 {
   this->name = name;
   this->age = age;
   this->passion = passion;
 }

 void show()
{
 cout << "My name is " << name
 cout << " and I'm a fan of " << passion;
 }
}; //конец класса
} //конец пространства
```

```
namespace household{
class Fan{
private:
    int size;
    float power;
    string color;

public:
Fan(int size, float power, string color)
{
   this->size = size;
   this->power = power;
   this->color = color;
}

void show()
{
   cout << "I'm fan " << power << "W. "
   cout << color " << color;
 }
};
}
```




```
using namespace people;
void main(){
Fan f1("John", 35, "Beatles");
f1.show();
household::Fan f2(10, 12.5, “blue");
f2.show();
}
```

```
void main(){
people::Fan f1(“Max", 28, “ABBA");
f1.show();
household::Fan f2(8, 10.5, “grey");
f2.show();
}
```

Занятие 3. Темы

- Пример программы с использованием классов
- Перегрузка операций
- Дружественные функции

Пример

- Полином от одной переменной это математическая функция заданная следующей формой:

$$a_n * x^n + * a_{n-1} * x^{n-1} + a_{n-2} * x^{n-2} \ldots a_2 * x^2 + a_1 * x^1 + a_0 * x^0 = 0$$

где $a_n \ldots a_0$ - коэффициенты полинома, x - переменная. Пример : $7x^4 + 10x^2 + 5 = 0$

- Напишем класс для работы с полиномами. Поскольку полином однозначно определяется своими коэффициентами, то в нашем классе следует их хранить.
- Для хранения набора данных одного типа разумно использовать массив.
- Длину этого массива тоже следует хранить внутри класса, она будет говорить о степени полинома.
- Пусть конструктор без параметров создает пустой массив.
- В конструктор с параметрами передадим массив коэффициентов и длину для создания и заполнения массива.
- В деструктора будем стирать созданный массив коэффициентов.

Класс для хранения полинома


```
class Polynom
{
  private:
    int* coeffs;
    int size;

  public:
    Polynom()
    {
        size = 0;
        coeffs = NULL;
    }

    Polynom(int co[], int size)
    {
       this->size = size;
       coeffs = new int[size];
       for (int i=0; i<size; i++)
          coeffs[i] = co[i];
    }
};
```

```
int main()
{
    int mas1[] = {7,0,10,0,5};
    int mas2[] = {1,2,3,4,6,7};
    Polynom poly1(mas1, 5);
    Polynom poly2(mas2, 6);
    poly1.print();
}
```

Методы класса Polynom

Дописать в класс Полином следующие методы:

- Деструктор
- Вывод полинома на экран
- Получение степени полинома
- Получение коэффициентов полинома
- Сложение двух полиномов

Перегрузка операторов

- Для того чтобы использовать стандартные операции с определенными пользователями типами используется перегрузка операторов.
- Перегрузка операторов позволяет определить или переопределить действия которые будут выполняться конкретным оператором применительно к объекту заданного класса.
- Перегрузить можно только те операторы, которые уже определены в C++ (например сложение, деление, проверка на равенство).
- Перегрузка операторов заключается в написании специального метода, который будет вызван, при обращении к оператору.
- Данный метод состоит из следующих частей
  1. Возвращаемый тип
  2. Ключевое слово **operator**
  3. Символ операции (например +)
  4. Список аргументов, заключенный в скобки

Пример: *Polynom operator+(Polynom pol)*

Перегрузка оператора доступа []

- Оператор доступа по индексу (**[ ]**) как и другие операторы можно перегрузить, чтобы он выполнял разные действия в зависимости от того к какому объекту он применяется.

```
class Polynom
{
  private:
    int* coeffs;
    int size;

    Polynom(int co[], int size)
    {
       this->size = size;
       coeffs = new int[size];
       for (int i=0; i<size; i++)
          coeffs[i] = co[i];
    }

int operator[] (int i)
    {
        return coeffs[i];
    }
};
```

```
int main()
{
    int mas1[] = {7,0,10,0,5};
    int mas2[] = {1,2,3,4,6,7};
    Polynom poly1(mas1, 5);
    Polynom poly2(mas2, 6);
    cout << "p1[0] = " << p1[0]
}
```

Перегрузка операторов

- Оператор может быть перегружен как внутри класса, так и снаружи.
- Если оператор перегружен внутри класса, то левым операндом этого оператора становится сам объект, а правый передается в качестве параметра.

```
class Polynom
{
  private:
    int* coeffs;
    int size;
public:
Polynom()
{
   size = 0;
   coeffs = NULL;
}

    Polynom operator+(Polynom pol)
    {
       ...
       return ...
    }
};
```

```
int main()
{
    Polynom poly1, poly2, poly3;

    poly3 = poly1 + poly2;
}
```

Перегрузка операторов

- Если оператор перегружен вне класса, то левым операндом этого оператора становится первый параметр функции, а правым второй параметр функции.
- Перегружать операторы вне классов необходимо, если объект является правым операндом, а не левым.
- Для того чтобы упростить оператору перегруженному вне класса доступ к скрытым полям класса, можно использовать механизм дружественных функций.

```
class Polynom
{
  private:
    int* coeffs;
    int size;
public:
Polynom()
{
   size = 0;
   coeffs = NULL;
}
};
```

```
Polynom operator+(int i, Polynom p)
  {

     return …
  }
```

```
int main()
{
    Polynom poly1, poly2;

    poly2 = 5 + poly1;
}
```

Дружественные функции

- Принцип инкапсуляции и ограничения доступа к данным запрещает функциям, не являющимся методами соответствующего класса, доступ к скрытым или защищенным данным объекта. Но в C++ это ограничение можно обойти используя дружественные функции.
- Для объявления в классе дружественной функции перед её именем пишут ключевое слово **friend**.
- Сама дружественная функция (определение) размещается вне класса, но имеет доступ к его скрытым полям.

```
class firstClass {
  private:
   int info;

   public:
   firstClass(int n){
       info = n;
   }
void printInfo() {
       cout << "info = " << info << endl;
   }
friend void showMeAll(firstClass);
};
```

```
void showMeAll(firstClass fc)
{
    cout << "secret info: "
    << fc.info;
}
```

```
void main()
{
    firstClass fc(5);
    cout << fc.info; //!!!
    showMeAll(fc) //ok
}
```

Перегрузка оператора вставки

- Оператор вставки (**<<**) как и другие операторы можно перегрузить, чтобы он выполнял разные действия в зависимости от того к какому объекту он применяется.
- Функции operator<< () и operator>> () должны быть дружественными по отношению к классу для которого они определены, чтобы они могли обратиться к его сокрытым полям.

Перегрузка оператора вставки

```
class Car
  {
     private:
       int number;
       char color;
       string mark;
    public:
     Car(){
          number = 0;
          color = 'R';
          mark = "Ford";
        }

        void editCar(int num, char col, string m){
          number = num;
          color = col;
          mark = m;
        }

     friend ostream& operator << (ostream&, Car&);
  };
```

```
ostream& operator << (ostream& s, Car& c)
 {
    s << "\nnumber: " << c.number
    << "\ncolor: " << c.color
    << "\nmark: " << c.mark << endl;
    return s;
 }

int main()
{

   Car myCar;
   myCar.editCar(321, 'Y', "Renault");
   cout << myCar;
}
```

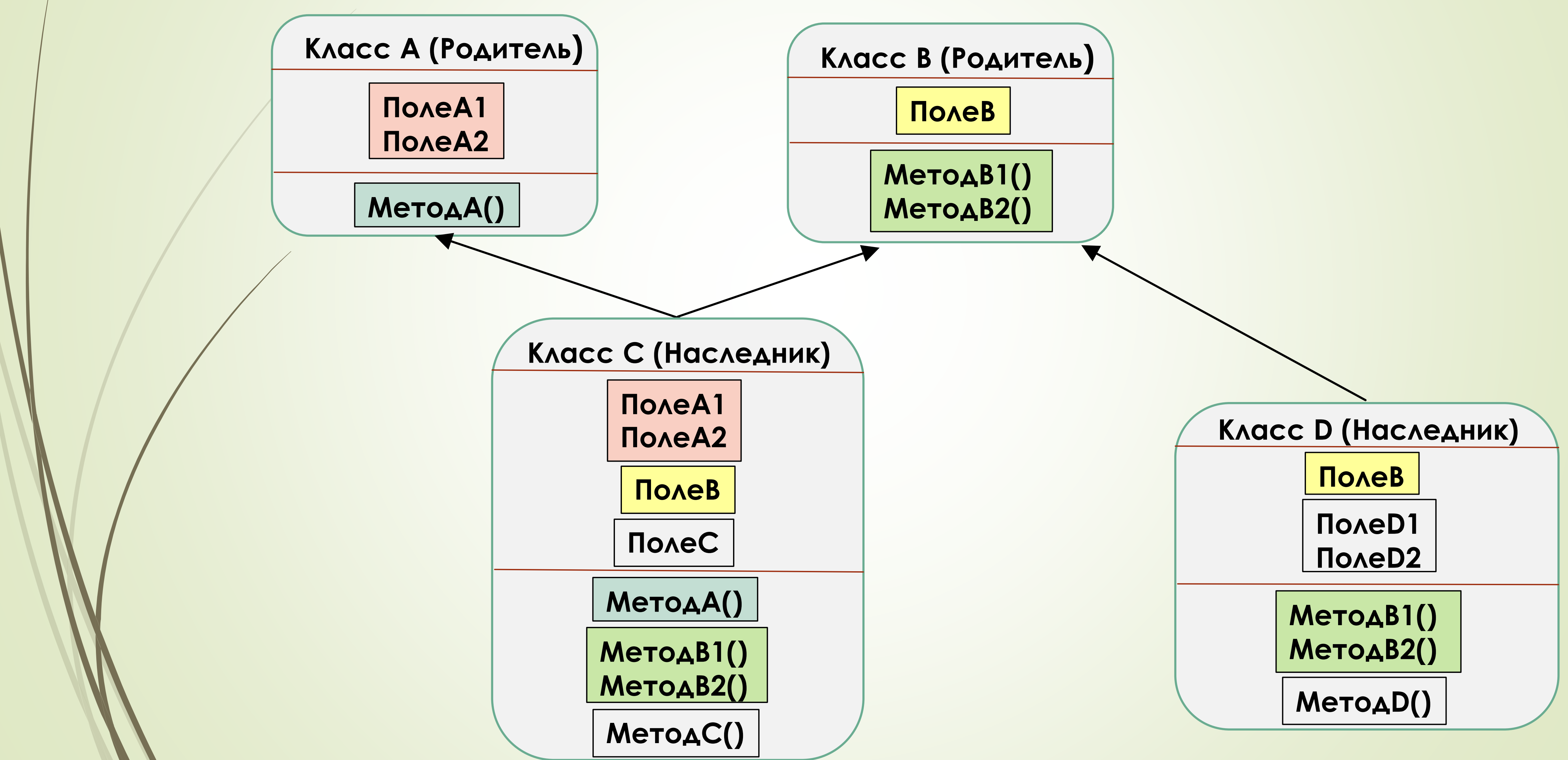
Занятие 4. Темы

- Наследование
- Полиморфизм
- Виртуальные функции

Наследование

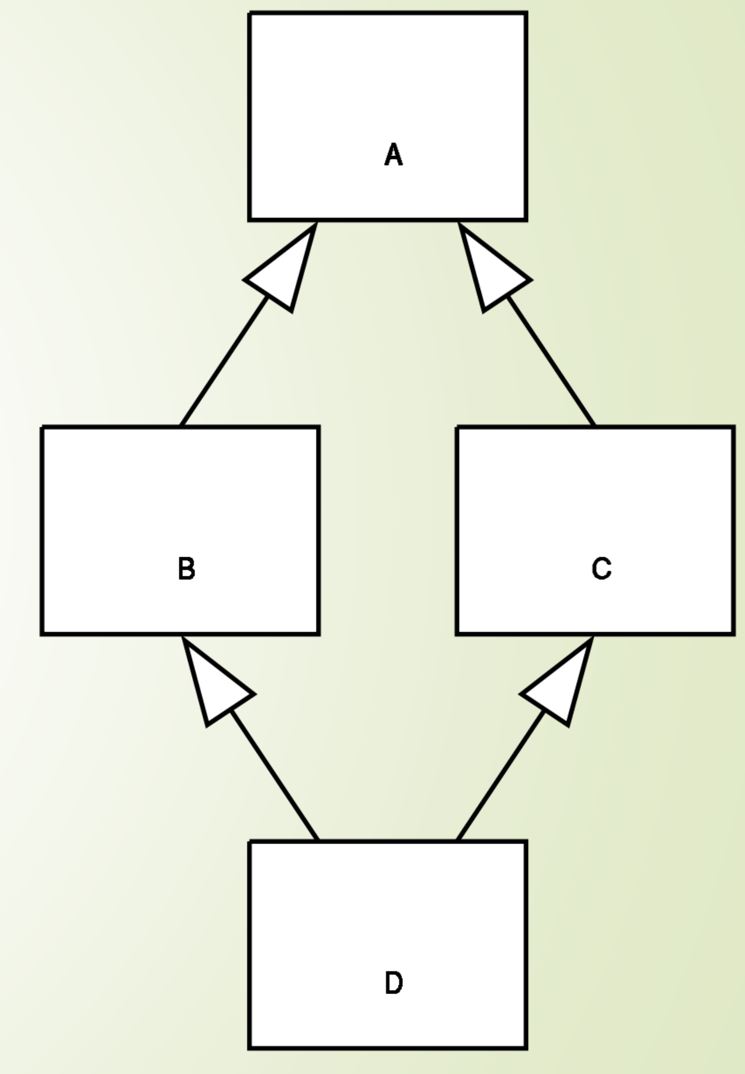
- Если есть несколько классов у которых часть полей и методов одинаковая то для того чтобы не дублировать код можно использовать наследование.
- Наследование это создание новых классов, называемых ***наследниками*** или ***производными классами***, из уже существующих или базовых классов. Производный класс получает все возможности базового класса, но может также быть усовершенствован за счет добавления собственных полей и методов.
- Наследование позволяет использовать существующий код несколько раз. Имея написанный и отлаженный базовый класс, необязательно модифицировать его при необходимости внесения в код изменений. Вместо этого можно использовать механизм наследования.
- Наследование позволяет использовать классы, созданные кем-то другим, без модификации кода, просто создавая производные классы, подходящие для частной ситуации.
- Отношение наследования объявляется через двоеточие, сначала приводится класс наследник, потом родительский класс: **class Наследник : Родитель**
- Через запятую можно перечислить несколько родительских классов.

Множественное наследование



Ромбовидное наследование

- Ромбовидное наследование - ситуация в объектно-ориентированных языках программирования с поддержкой множественного наследования
- Два класса B и C наследуют от A, а класс D наследует от обоих классов B и C.
- Возникает неоднозначность: если метод класса D вызывает метод, определенный в классе A, а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C?
- C++ по умолчанию не создает ромбовидного наследования: компилятор обрабатывает каждый путь наследования отдельно, в результате чего объект D будет на самом деле содержать два разных подобъекта A, и при использовании членов A потребуется указать путь наследования (B::A или C::A).



Пример наследования

Базовый класс

```
class Person
{
  protected:
  int id;
  string name;

  public:
  Person(){
     cout << “person created”;
     id = 0;
     name = "anonimus";
  }

  void show() {
     cout << "person show" << endl;
  }
};
```

Класс наследник 1

```
class Student : Person
{
   private:
   int group;

   public:
   Student() : Person()
   {
     cout << “student”<< endl;
     group = 0;
   }

   void show()
   {
      cout << “stud” << name;
      cout << group << endl;
   }
};
```

Класс наследник 2

```
class Teacher : Person
{
   private:
   char rnk;

   public:
   Teacher() : Person()
   {
     cout << "teacher" << endl;
     rnk = 'A';
   }

   void show()
   {
      cout << “teach” << name;
      cout  << "Rank: " << rnk;
   }
};
```

Указатели и классы

Обычный вызов методов объекта

```
void main()
{
    Student st;
    Teacher tea;
    Person per;

    st.show();
    tea.show();
    per.show();
}
```

Вызов методов через указатель

```
void main()
{
    Student st;
    Teacher tea;
    Person per;

    Student* stPtr;
    Teacher* teaPtr;
    Person* perPtr;

    stPtr = &st;
    teaPtr = &tea;
    perPtr = &per;

    stPtr->show();
    teaPtr->show();
    perPtr->show();
}
```

Публичное наследование

class Student : **public** Person

Указатель на базовый класс

```
void main()
{
    Student st;
    Teacher tea;
    Person per;

    Student* stPtr;
    Teacher* teaPtr;
    Person* perPtr;

    perPtr = &st;
    perPtr->show();
    perPtr = &tea;
    perPtr->show();
}
```

Полиморфизм

- Полиморфизм это свойство, которое позволяет одно и то же имя использовать для решения двух и более схожих, но технически разных задач.
- Концепцией полиморфизма является идея "один интерфейс, множество методов".
- Указатель на базовый класс на самом деле может также указывать на любой объект наследник.
- Можно создать полиморфную функцию, с одинаковым именем в разных классах наследниках. Выбор конкретной полиморфной функции будет зависеть от того на какой именно объект указывает указатель на базовый класс.
- Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор.
- Чтобы использовать полиморфизм в C++, необходимо выполнить два условия:
  1. Все классы должны являться наследниками одного и того же базового класса.
  2. Полиморфная функция должна быть объявлена виртуальной (virtual) в базовом классе

Виртуальные функции

- В объектно-ориентированном программировании виртуальная функция это функция класса, которая может быть переопределена в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.
- Виртуальные методы это один из важнейших приёмов реализации полиморфизма. Они позволяют создавать общий код, который может работать как с объектами базового класса, так и с объектами любого его класса-наследника.
- Базовый класс определяет способ работы с объектами и любые его наследники могут предоставлять конкретную реализацию этого способа.
- Техника вызова виртуальных методов называется ещё "динамическим связыванием".
- Имя метода, использованное в программе, связывается с адресом входа конкретного метода динамически (во время исполнения программы), а не статически (во время компиляции).

Виртуальные функции в C++

- Функция помечается в родительском классе как виртуальная с помощью ключевого слова **virtual.**
- Класс наследник переопределяет функциональность виртуальной функции.
- После определения функции как виртуальной при обращении к ней через указатель на родительский класс, будет вызвана функция класса наследника, а не родительская.

Базовый класс

```
class Person
{
  protected:
string name;

  public:
  Person(){
     id = 0;
     name = "anonimus";
  }

  virtual void show() {
     cout << "person show" << endl;
  }
};
```

Класс наследник

```
class Student : public Person
{
   private:
   int group;

   public:
   Student() : Person()
   {
     group = 0;
   }

   void show()
   {
      cout << group << endl;
   }
};
```

Указатель на базовый класс

```
void main()
{
   Student st;
   Teacher tea;
   Person per;

   Student* stPtr;
   Teacher* teaPtr;
   Person* perPtr;

   perPtr = &st;
   perPtr->show();
   perPtr = &tea;
   perPtr->show();
}
```

Занятие 5. Темы

- Перегрузка и переопределение
- Абстрактные классы
- Язык моделирования UML
- Динамически подключаемые библиотеки (DLL)

Перегрузка и переопределение

- Когда в объектно-ориентированной программе есть несколько методов с одинаковыми именами, то такая ситуация описывается похожими терминами, это может быть либо **перегрузка (overload)** либо **переопределение (override)**.
- Перегрузка (overload) – в этом случае методы находятся в одном классе и отличаются друг от друга сигнатурой (набором параметров). При вызове вызывается метод с подходящими параметрами.
- Переопределение (override) – есть родительский класс и класс наследник, в классе наследнике есть метод с таким же именем и параметрами как в родительском. Если родительский метод объявлен виртуальным, то переопределение делает возможным использование полиморфизма.

Перегрузка

```
class Person {
string name;
   public:
   Person(){
       name = "anonimus";
   }
Person(string n){
       name = n;
   }
};
```

Переопределение

```
class Person {
   protected:
   string name;

   public:
    void show() {
       cout << "person show" << endl;
   }
};
```

Переопределение в наследнике

```
class Student : public Person {
    private:
    int group;
    public:
    void show()
    {
       cout << “student” << endl;
    }
};
```

Абстрактный класс

- Абстрактным классом называется базовый класс, объекты которого никогда не будут реализованы.
- Абстрактный класс может существовать с единственной целью — быть родительским по отношению к производным классам, объекты которых будут реализованы.
- Чтобы указать что класс будет абстрактным необходимо ввести в класс хотя бы одну чистую виртуальную функцию.
- Чистая виртуальная функция — это функция, после объявления которой добавлено выражение =0.   (virtual void show() =0;)
- После указания класса как абстрактного создание из него объектов становится невозможным.

Абстрактный базовый класс

```
class Person
{
  public:
   virtual void show() =0;
};
```

Класс наследник

```
class Student : public Person
{

   void show()
   {
      cout << group << endl;
   }
};
```

Создание объектов

```
void main()
{
   Student st;
  Person per; //ошибка!!
}
```

Язык моделирования UML и диаграмма классов

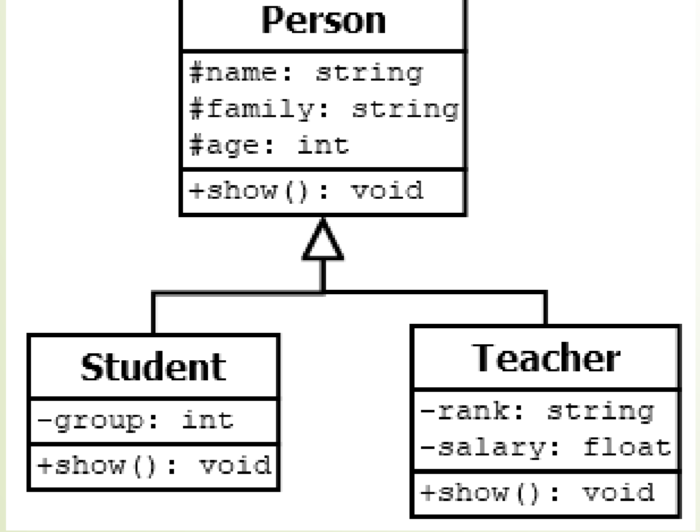
- Так же, как для визуального отображения алгоритмов используются блок-схемы, для графического описания объектно-ориентированной программы используются диаграммы языка UML (Unified Modeling Language).
- Язык UML состоит из диаграмм различного типа, наиболее употребляемой является диаграмма классов (class diagram) на которой изображаются классы присутствующие в программе и связи между ними.
- На диаграмме классов класс изображается в виде прямоугольника разбитого на три части:
  - В верхней части находится имя класса (абстрактный класс – курсивом).
  - В средней – поля класса.
  - В нижней – методы класса.

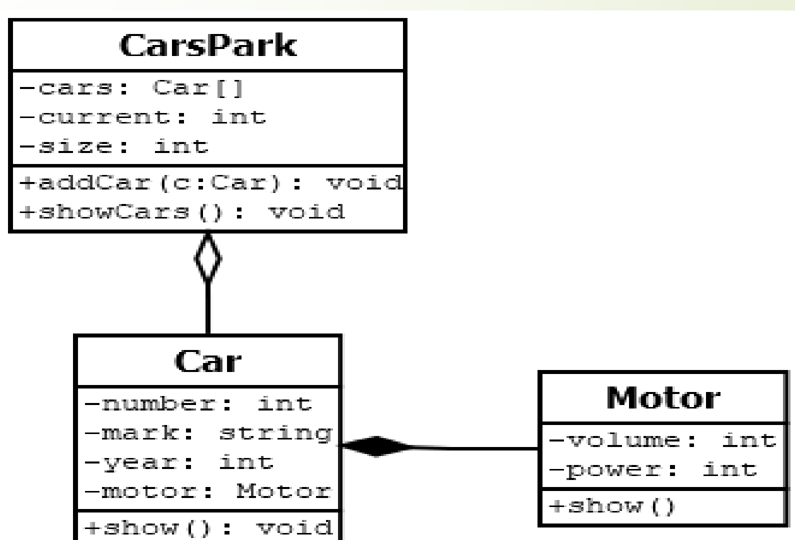
```
Person
#name: string
#family: string
#age: int
+show(): void
```

- Для описания модификаторов доступа полей и методов используются знаки:
  - - означает private.
  - + означает public.
  - # означает protected.

Диаграмма классов. Связи между классами

- На диаграмме классов стрелочками обозначаются следующие типы связей:
  - **Наследование** (**is a** – является) – стрелка с треугольным острием направленным на родительский класс:
  - **Реализация** (наследник реализует методы абстрактного класса, называемого интерфейсом) – такая же стрелка, но линия пунктирная:
  - **Агрегация** (**has a** – имеет) - один класс включает другой, но они независимы, линия с незакрашенным ромбом на конце указывающим на класс, который использует второй:
  - **Композиция** (has a – имеет) - один класс включает другой, второй класс не существует отдельно от первого, обозначается, как агрегация, но ромб закрашен (черный):
  - **Ассоциация** – иная логическая связь между двумя классами, простая стрелка, или линия:





Динамически подключаемые библиотеки (DLL)

- Динамически подключаемая библиотека DLL (англ. Dynamic Link Library) это специальный файл, обычно с расширением .dll в котором содержатся функции.
- После подключения dll можно использовать содержащиеся в ней функции в своей программе.
- Можно создавать свои библиотеки DLL и использовать их затем в других проектах.
- Существует два способа загрузки DLL: с явной и неявной компоновкой.
  - При неявной компоновке все функции и библиотеки DLL сразу подгружаются в программу.
  - Явная компоновка дает больше гибкости в загрузке DLL и подключении экспортируемых функций. Такой тип компоновки позволяет подгружать DLL по мере необходимости.
- Библиотеки dll используются только в операционной системе Microsoft Windows.

Создание DLL

- Чтобы создать свою динамическую библиотеку DLL нужно при создании нового проекта в IDE выбрать пункт «Dynamic Link Library».
  (в MS VS – Console Application -> в параметрах выбрать DLL)
- После компиляции проекта будет получен не файл .exe, а файлы .dll и .lib (или .a)
- Для того чтобы функция была экспортирована и могла использоваться вне dll нужно указать в заголовочном файле перед её именем модификатор **__declspec(dllexport)**

Lib.h

```
__declspec(dllexport)
void printArray(int arr[], int size);

__declspec(dllexport)
void printHello();
```

Модификатор **extern "C"** перед __declspec нужен чтобы функции сохраняли свои имена при экспорте и были доступны для динамической загрузки

Lib.cpp

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

void printArray(int arr[], int size){
   for (int i=0; i<size; i++)
              cout << arr[i] << " ";
}

void printHello(){
   cout << "Hello from dll!!";
}
```

Подключение DLL

- Для того чтобы подключить готовую библиотеку dll к проекту нужно выполнить следующие шаги:
  1. Скопировать и подключить в проект заголовочный файл .h из dll проекта.
  2. Скопировать в папку с .exe файлы .dll и .a (или .lib)
  3. В настройках проекта добавить файл .a (или .lib)

     В среде Code Blocks: Build Options -> Linker Settings -> Link libraries -> Add

     В Visual Studio: Properties -> Configuration Properties -> Linker -> Input -> Additional Dependencies

     В Visual Studio также нужно добавить путь к файлу .lib

     Properties -> Configuration Properties > Linker -> General -> Additional Library Directories

Динамическая загрузка DLL

- Для динамической загрузки требуется только DLL (не нужен ни .lib ни заголовочный файл)

```
//Загрузка DLL
#include <windows.h>

void main()
{
  HINSTANCE h;

  void (*DllFunc) ();
  void (*DllFuncPrintA) (int arr[], int size);

  h=LoadLibrary(TEXT("CPP_DLL.dll"));

  if (!h){
    cout << "Cannot load DLL!!!"
    return;
  }
```

```
//Загрузка функций из DLL
  DllFunc=(void (*)())
GetProcAddress(h,"printHello");

  DllFuncPrintA=(void (*)(int arr[], int size))
GetProcAddress(h,"printArray");

  if (!DllFunc || !DllFuncPrintA)
  {
    cout << "Cannot load fuctions!!!";
    return;
  }

  DllFunc();

  int arr[] = {9,7,5,3,1};
  DllFuncPrintA(arr, 5);

  FreeLibrary(h);
```

Занятие 6. Темы

- Изобретение велосипедов
- Стандартная библиотека STL
- Шаблон vector
- Шаблон list
- Паттерны проектирования
- Паттерн Наблюдатель

Стандартная библиотека STL

- Библиотека стандартных шаблонов (STL) (Standard Template Library) это библиотека для C++ содержащая набор алгоритмов, вспомогательных функций, контейнеров и средств доступа к их содержимому.
- Все компоненты STL расположены в пространстве имён std.
- В библиотеке STL выделяют пять основных компонентов:
  1. Контейнер (англ. container) — хранение набора объектов в памяти.
  2. Итератор (англ. iterator) — обеспечение средств доступа к содержимому контейнера.
  3. Алгоритм (англ. algorithm) — определение вычислительной процедуры.
  4. Адаптер (англ. adaptor) — адаптация компонентов для обеспечения различного интерфейса.
  5. Функциональный объект (англ. functor) — сокрытие функции в объекте для использования другими компонентами.

Контейнеры

- С помощью контейнеров в STL осуществляется хранение набора объектов в памяти.
- Контейнеры библиотеки STL можно разделить на четыре категории:
  1. Последовательные (vector, list, deque)
  2. Ассоциативные (set, multiset, map, multimap)
  3. Контейнеры-адаптеры (stack, queue, priority_queue)
  4. Псевдоконтейнеры (bitset, basic_string, valarray)
- Согласно стандарту C++, любой контейнер должен содержать методы:
  - begin() (Возвращает итератор на первый элемент контейнера)
  - end() (Возвращает итератор на место после последнего элемента контейнера)
  - size() (Возвращает количество элементов в контейнера)
  - max_size() (Возвращает максимально возможное количество элементов в контейнере)
  - empty() (Возвращает true, если контейнер пуст)
  - swap() (Обменивает содержимое двух контейнеров)

Итераторы

- С помощью итераторов в STL осуществляется проход по элементам контейнера.
- Стандарт C++ определяет пять категорий итераторов:

1. **Входные**. Обеспечивают доступ для чтения.

   Операции: ++, *, ->, =, ==, !=, конструктор копии

2. **Выходные**. Обеспечивают доступ для записи.

   Операции: ++, *, конструктор копии

3. **Однонаправленные**. Обеспечивают доступ для чтения и записи. Позволяют передвигаться вперед по контейнеру.

   Операции: ++, *, ->, =, ==, !=, конструктор копии, конструктор по умолчанию.

4. **Двунаправленные**. Обеспечивают доступ для чтения и записи. Позволяют передвигаться по контейнеру в обе стороны.

   Операции: ++, --, *, ->, =, ==, !=, конструктор копии, конструктор по умолчанию.

5. **Произвольного доступа**. Эквивалентны обычным указателям: поддерживают арифметику указателей.

   Операции: ++, --, *, ->, =, ==, !=, +, -, +=, -=, <, >, <=, >=, [], конструктор копии, конструктор по умолчанию

Контейнер vector

- Стандартный шаблон обобщённого программирования языка C++ std::vector<T> это реализация динамического массива переменного размера.
- Шаблон vector расположен в заголовочном файле <vector>.
- Данный контейнер эмулирует работу стандартного массива C (например, быстрый произвольный доступ к элементам), но также имеет дополнительные возможности, такие как изменения размера вектора при вставке или удалении элементов.
- Все элементы вектора должны принадлежать одному типу.
- Класс vector обладает стандартным набором методов для доступа к элементам, добавления и удаления элементов, а также получения количества хранимых элементов

Основные методы класса vector

Метод	Значение
push_back	Вставка элемента в конец вектора
pop_back	Удаление последнего элемента вектора
resize	Изменение размера вектора на заданную величину
insert	Вставка элементов в произвольное место вектора
clear	Удаление всех элементов вектора

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

void main()
{
vector<int> vec;
int i;
for (i=0; i < 10; i++)
   vec.push_back(i);

for (i=0; i<vec.size(); i++)
   cout << vec[i] << endl;
}
```

Контейнер list

- Стандартный шаблон std::list<T> это реализация двусвязного списка.
- Шаблон list расположен в заголовочном файле <list>.
- В отличие от контейнера vector элементы контейнера list хранятся в произвольных кусках памяти, а не в непрерывной области памяти.
- К отдельному элементу такого контейнера нельзя обратиться по индексу
- Поиск перебором медленнее, чем у вектора из-за большего времени доступа к элементу.
- Вставка и удаление производятся очень быстро в любом месте контейнера.
- У класса list больше методов, чем у класса vector

Метод	**Значение**
push_front()	Вставка элемента в начало списка
pop_front()	Удаление первого элемента списка
sort()	Сортировка элементов
remove(val)	Удаляет элемент значение которого равно val

Итераторы

- Итератор (обходчик) является указателем специального типа, с помощью которого можно проходить по элементам контейнера.
- Итератор описывается типом **iterator**, перед которым через двойное двоеточие указывается к контейнеру какого типа относится этот итератор. Например, определение итератора для контейнера list<int> выглядит так **list<int>::iterator iter;**
- Как и при работе с обычным указателем доступ к содержимому осуществляется с помощью операции * (разыменование).
- Метод begin() возвращает указатель на первый элемент контейнера
- Метод end() возвращает указатель на место после последнего элемента контейнера

Метод erase(iterator) удаляет элемент на который указывает итератор

С помощью операций инкремента (++) и декремента (--) можно передвигать итератор по элементам контейнера.

Итераторы всех типов, кроме list и forward_list поддерживают ряд дополнительных операций

iter + n, iter – n, iter += n, iter -= n,
iter1 - iter2, >, >=, <, <=:

```
list<int> lst;

for (i=0; i<10; i++)
   lst.push_back(i);

list<int>::iterator iter;

for (iter=lst.begin(); iter!=lst.end(); iter++)
     cout << *iter << " ";
```

Ассоциативный контейнер set

- Стандартный контейнер set<T> представляет из себя бинарное дерево поиска и обеспечивает хранение элементов в упорядоченном виде.
- Преимуществом использования контейнера set является быстрота выполнения операций. Операции добавление(insert), удаление(erase), поиска(find) выполняются очень быстро.
- Контейнер set расположен в заголовочном файле <set>.
- Данный контейнер не допускает хранение одинаковых элементов.
- Добавление элементов в контейнер set осуществляется с помощью метода **insert()**

```
set<int> st;
for (i=0; i<5; i++) {
   cin >> n;
   st.insert(n);
}

set<int>::iterator iterS;

for (iterS=st.begin(); iterS!=st.end(); iterS++)
     cout << *iterS << " ";
```

Метод **find(x)** возвращает итератор на первый объект в контейнере, равный x, или set::end, если такой объект отсутствует

Метод **lower_bound(x)** возвращает итератор на первый объект в контейнере, который больше или равен x

Ассоциативный контейнер map

- Стандартный контейнер map<T> является словарем и обеспечивает хранение элементов, состоящих из пары (ключ, значение)
- Элементы в данном контейнере хранятся в упорядоченном по ключу виде
- Контейнер map расположен в заголовочном файле <map>.
- Данный контейнер не допускает хранения одинаковых ключей.
- Добавление элементов осуществляется с помощью метода **insert(pair<T1,T2>(val1,val2))**
- Метод **find(key)** возвращает итератор на элемент, чей ключ равен искомому.
- Поля **first** и **second** обеспечивают доступ к ключу и значению элемента map.

```
map<int, string> mp;

mp.insert(pair<int, string>(2, "Vasya"));
mp.insert(pair<int, string>(1, "Petya"));
mp.insert(pair<int, string>(3, "Kolya"));
```

```
map<int, string>::iterator iterM;

 for (iterM=mp.begin(); iterM!=mp.end(); iterM++)
     cout << iterM->first << " " << iterM->second << endl;
   cout << endl;

iterM=mp.find(2);
cout << iterM->first << " " << iterM->second << endl;
```

Контейнеры адаптеры stack и queue

- Контейнеры адаптеры stack и queue это разновидности последовательного контейнера, у которых для простоты и ясности ограничен интерфейс (методы).
- Контейнеры-адаптеры не поддерживают итераторы.
- Контейнер queue расположен в заголовочном файле <queue>.
- Контейнер queue (очередь) соответствует FIFO (первым поступил — первым обслужен). Первый элемент, который вставляется, в очередь, должен быть первым элементом, извлекаемым из очереди.
- Метод **front()** возвращает ссылку на первый элемент в начале контейнера queue.
- Метод **push()** добавляет элемент в конец контейнера queue.
- Метод **pop()** удаляет элемент из начала контейнера queue

```
queue<int> qu;
 for (i=0; i<10; i++)
    qu.push(i);
 while (qu.size()>0){
    cout << qu.front() << " ";
    qu.pop();
 }
```

Контейнер адаптер stack

- Стандартный контейнер stack<T> представляет собой контейнер адаптер реализующий функциональность стека.
- Контейнер stack (стек)соответствует LIFO (последним поступил — первым обслужен). Последний элемент, отправленный в стек, становится первым извлекаемым элементом.
- Контейнер stack расположен в заголовочном файле <stack>.
- Метод **pop()** удаляет элемент из верхней части контейнера stack (последний добавленный)
- Метод **push()** добавляет элемент в верхнюю часть контейнера stack.
- Метод **top()** возвращает ссылку на элемент в верхней части контейнера stack.

```
stack<int> st;
for (i=0; i<10; i++)
   st.push(i);

 while (!st.empty())
 {
    cout << st.top() << " ";
    st.pop();
 }
```

Паттерны проектирования

- Шаблон проектирования или паттерн (англ. design pattern) это повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.
- Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях.
- Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.
- На наивысшем уровне существуют архитектурные шаблоны, они охватывают собой архитектуру всей программной системы.
- Шаблоны проектирования часто описываются с помощью диаграмм UML.
- Алгоритмы по своей сути также являются шаблонами, но не проектирования, а вычисления, так как решают вычислительные задачи.

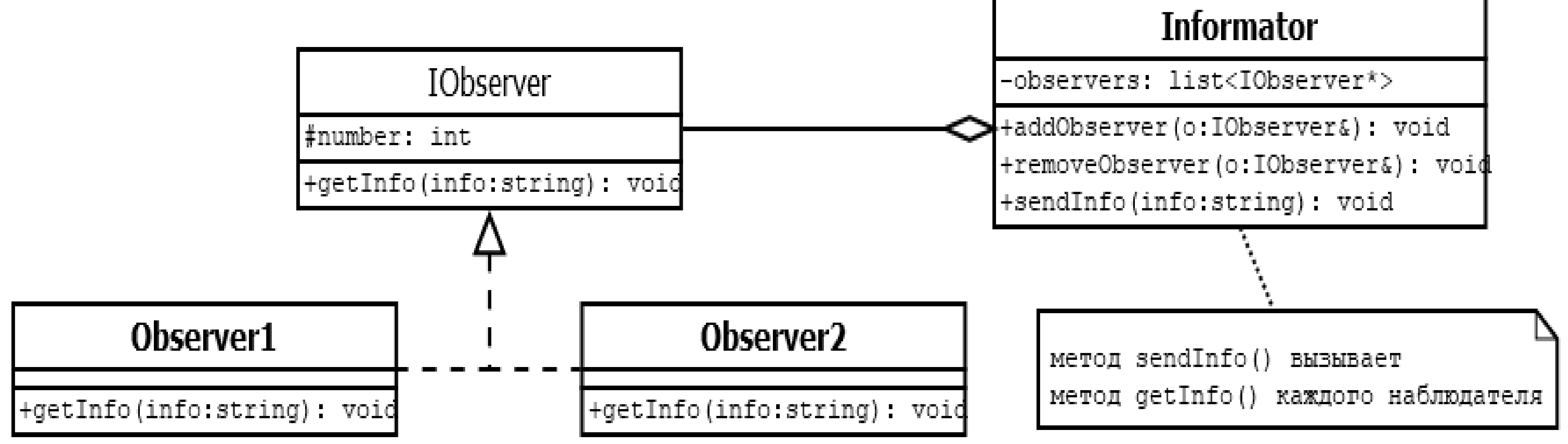
Некоторые паттерны проектирования

- К настоящему времени разработано несколько десятков паттернов проектирования для решения различных типовых задач. Вот лишь несколько:

Название	Англ.название	Описание
Шаблон делегирования	Delegation pattern	Объект внешне выражает некоторое поведение, но в реальности передаёт ответственность за выполнение этого поведения связанному объекту
Одиночка	Singleton	Класс, который может иметь только один экземпляр
Абстрактная фабрика	Abstract factory	Класс, который представляет собой интерфейс для создания компонентов системы
Строитель	Builder	Класс, который представляет собой интерфейс для создания сложного объекта
Наблюдатель	Observer	При изменении состояния одного объекта все зависящие от него оповещаются об этом событии

Паттерн «Наблюдатель»

- Шаблон «Наблюдатель» описывает ситуацию, когда в программе есть объект-Информатор, и несколько объектов-Наблюдателей.
  - Наблюдатели подписываются или отписываются от получения информации от Информатора.
  - Информатор высылает информацию всем кто на него подписался.



Паттерн «Наблюдатель»


```
class IObserver {
  protected:
  int number;

  public:
  virtual void getInfo(string info)=0;
};
```

```
class Observer1 : public IObserver {
   public:
   void getInfo(string info) {
      cout << "Observer of type 1
      got info: " << info << endl;
   }
};
```

```
class Observer2 : public IObserver {
   public:
   void getInfo(string info){
      cout << "Observer of type 2
      got info: " << info << endl;
   }
};
```

```
class Informator
{
   private:
   list<IObserver*> observers;

   public:
   void addObserver(IObserver& o)
   {
      observers.push_back(&o);
   }
   void removeObserver(IObserver& o)
   {
      observers.remove(&o);
   }
   void sendInfo(string info)
   {
     list<IObserver*>::iterator iter;
     for (iter = observers.begin();
              iter!=observers.end(); iter++)
       (*iter)->getInfo(info);
   }
};
```

```
void main()
{
   string info;
   Informator inf;
   Observer1 obs1(11), obs2(12);
   Observer2 obs3(21), obs4(22);

   inf.addObserver(obs1);
   inf.addObserver(obs2);
   inf.addObserver(obs3);
   inf.addObserver(obs4);

   cin >> info;
   inf.sendInfo(info);

   inf.removeObserver(obs2);

   cin >> info;
   inf.sendInfo(info);
}
```