

Билеты ПКШ

1. Структуры и классы языка C++

Классы и структуры являются конструкциями, в которых пользователь определяет собственные типы. Классы и структуры могут включать данные-члены и функции-члены, позволяющие описывать состояние и поведение данного типа.

Классы

Класс – это объединение полей данных и методов. Это тип данных, определяемых пользователем.

Исторически «класс» в C++ является расширением типа данных «структура» в C.

Классы в программировании состоят из свойств и методов.

Переменные, которыми можно охарактеризовать объект класса, называются свойствами класса.

Функции, которые принадлежат к описанию класса, называют методами. Любой метод класса имеет доступ к любым данным этого класса.

```
class Human
```

```
{
```

```
public:
```

```
string name;    //имя
```

```
int age;        //возраст
```

```
int height;     //рост
```

```
void Print() //метод класса, отвечающий за печать информации о человеке
```

```
{
```

```
cout << "Имя: " << name << endl << "Возраст: " << age << endl << "Рост: " << height << endl;
```

```
}
```

```
};
```

Доступ к элементам класса в программе определяется модификаторами доступа `private`, `public`, `protected`.

Чаще всего используются `private`(доступ лишь в собственных методах) и `public`(доступ извне). По умолчанию члены класса являются `private`.

Следует различать описание класса и определение объектов класса. Сначала нужно описать класс, а затем можно создавать объекты класса.

В C++ класс – это практически аналог структуры. Оба объявляют тип данных, и разница между ними есть только на стыке наследования и инкапсуляции.

Структуры

В языке C++ структура представляет собой класс, все члены которого по умолчанию `public`. Структуру можно объявить точно так же, как и класс, наделив ее такими же переменными-членами и функциями. Чаще всего структуры используются без функций.

```
struct Planet
```

```
{
```

```
    string name;
```

```
    int diameter;
```

```
    int moons;
```

```
};
```

Объявление структуры завершается точкой с запятой!

2. Заголовочные файлы классов

Определения классов могут размещаться в заголовочных файлах для упрощения их повторного использования. Это позволяет разбить программу на связанные фрагменты; размещать их в разных папках, которые объединяют логически несколько файлов в один пакет. В заголовочных файлах размещается информация, которая должна быть видна на этапе препроцессирования и компиляции программного файла.

Обычно определение класса помещается в заголовочный файл с тем же именем, что у класса, а методы, определённые вне тела класса, помещаются в файл .cpp с тем же именем, что у класса.

Классы — это пользовательские типы данных, которые могут определяться не только в одном месте. Поэтому класс, определённый в заголовочном файле, можно свободно подключать в другие файлы.

Использование препроцессора позволяет подключать заголовочные файлы, избегая их двойного подключения.

```
#ifndef ----.h
```

```
#define ----.h
```

```
...
```

```
#endif
```

После препроцессирования содержимое заголовочного файла будет вставлено в контекст соответствующего программного файла.

3. Файлы реализации классов

Существуют следующие типы файлов: .h (или .hpp - header файл,) и .cpp. В .h объявляется сам класс, все его методы, поля и указываются дружественные функции. В .cpp файле прописываются реализации методов, объявленных в .h файле. При этом в .cpp файле необходимо подключить заголовочный файл класса, а используемые библиотеки можно подключить в .h файле. Для использования класса в нашей программе (например, main.cpp) нужно подключить в неё заголовочный файл .h *{здесь вставьте свой пример}*

4. Классы и объекты

Класс – совокупность полей данных и методов (для их обработки и взаимодействия с ними). Методы – функции, описанные в классе.

Пример создания класса:

```
class student {
```

```
private:
```

```
    string name;
```

```

        int group_number; // поля данных

public:

    student() = default; // конструкторы класса

    student(string a, int b){name = a; group_number = b} ;

    ~student(); // деструктор класса

    void say_hello(){cout << "Hello" << endl;}; // методы класса

    void change_group(int new_group){

        group_number = new_group;

        cout << "Group changed" << endl;

    }

};

```

Объект – экземпляр класса.

Пример создания и использования объекта класса:

```

int main(){

    student a; // создание объекта с пустыми полями

    student b("Ivan Ivanov", 1); // создание объекта класса с заполнением полей

    a.say_hello(); // использование методов класса

    a.change_group(5);

    return 0;

}

```

5. Члены класса – методы и поля

Поле – переменная внутри класса. Существует принцип инкапсуляции, согласно которому поля с данными следует размещать в `private`, чтобы их нельзя было изменить извне. Метод – функция класса, как правило они размещаются в `public`. Использовать поля и методы можно следующим образом.

```

class house{

```

```

private:

int house_num;

public:

bool is_on_sale;

int get_house_num(){return house_num;}

house(int a, bool b){house_num = a; is_on_sale = b}

};

Int main(){

house b(5);

b.is_on_sale = 0;// обращение и изменение поля класса

cout << b.get_house_num() << endl;//использование метода

} // как по мне какой-то алкашный пример, но можно и свой придумать - Ваня

```

6. Конструкторы классов

Конструктор – функция класса, имеющая то же название что и сам класс, не имеющая типа, которая вызывается, чтобы выделить память для объекта класса и заполнить поля данными.

Рассмотрим пример:

```

class car{

private:

int horse_powers;

string mark;

public:

car() = default;// конструктор для заполнения полей значениями по умолчанию,

//car(){horse_powers = 0;} идентично верхнему // 1

car(int a, string b){horse_powers = a; mark = b;} // 2 //

```

```
car(car& a){horse_powers = a.horse_powers;mark = a.mark;}// 3 // конструктор копирования
```

```
}
```

```
// что ниже лучше не писать // а ты зачем написал? :D - Ваня
```

Конструкторы 1 2 3, можно переписать с использованием list initialization, тогда это будет выглядеть так:

```
car() : horse_powers(0){}
```

```
car(int a, string b) : horse_powers(a), mark(b){}
```

```
car(car &a) : : horse_powers(a.mark), mark(a.mark){}
```

7. Типы защищенности членов класса.

Одной из важных особенностей ООП является инкапсуляция. В случае C++ она реализована на основе 3 спецификаторов доступа: public, protected и private. Public является общедоступным типом доступа, то есть к нему могут обратиться из любой точки программы. Private – это спецификатор, который обеспечивает сокрытие данных, а точнее к сокрытой информации можно обращаться только изнутри самого класса. Также он является спецификатором по умолчанию, то есть без упоминания public и private все данные внутри класса будут private. Последний тип protected. Он обеспечивает аналогичное сокрытие данных, но используется при наследовании классов, так как может передать классу-наследнику методы из protected поля, когда в то же самое время private не может похвастаться такой особенностью. В случае отсутствия наследования использование protected не имеет особого смысла.

```
class track:
```

```
{
```

```
private:
```

```
int bass;//Пример приватного члена
```

```
public:
```

```
string text;// Пример public члена, возможно изменение извне
```

```
void GetBass();// Пример метода, которым можно пользоваться из любой точки.
```

```
}
```

8. Обеспечение доступа к защищенным данным класса

Доступ к защищенным членам класса обеспечивается при помощи методов (функций) класса или друзей класса. Т.е. для получения данных из защищенного `private` или `protected` поля необходимо использовать включенные в класс методы или дружественные классу функции, так как они обладают доступом в сокрытые поля класса.

Рассмотрим такой принцип на основе следующего примера.

```
class streak:{  
  
private:  
  
int value;  
  
public:  
  
streak();  
  
int getstreak() {return value;};  
  
void setstreak(const int &a) {value = a;};  
  
void increasestreak(const int &a){value+=a;};
```

Здесь мы можем видеть класс с некоторой переменной, к которой могут обращаться только члены класса, и методы, которые обеспечивают различное взаимодействие с этой переменной. Таким образом, мы обеспечили доступ у защищенной переменной.

9. Деструкторы классов

Деструктор - это функция, которая вызывается при разрушении объекта.

Деструкторы удаляют из памяти отработавшие объекты и освобождают выделенную для них память. Деструкторы не принимают аргументов и не возвращают значений.

Имя деструктора совпадает с именем конструктора (и соответственно с именем класса), он предваряется символом тильда “~”.

```
~Dog();//объявление деструктора
```

Если деструктор не определяется в классе, любой компилятор неявно создает деструктор по умолчанию, который занимается очисткой памяти, занятой объектом.

10. Статические поля данных и методы классов

Ключевое слово **static** можно применять к членам класса. Когда мы объявляем член класса статическим, мы уведомляем компилятор о том, что существует только одна копия этого static члена. *Один статический член класса разделяется всеми объектами класса.* Если не представлено других значений инициализации, все статические данные при первом создании объекта инициализируются нулевыми значениями. //объявление статического члена не обеспечивает распределения памяти и инициализацию.

Глобальное определение члена данных класса вне этого класса осуществляется повторным объявлением этой статической переменной с помощью оператора **“::”**.

```
class Share
```

```
{static int n;
```

```
public:
```

```
void setn ();
```

```
};
```

```
int Share::n; определение статического члена
```

Если static переменная является открытой public переменной, к ней можно обращаться напрямую через имя ее класса, без ссылки на какой-либо конкретный объект.

```
class Share
```

```
{ public:
```

```
static int n;
```

```
void setn ();
```

```
};
```

Обращаемся к статической переменной напрямую

```
Share::n=100;
```


Также можно объявить метод класса статическим. К статической функции, члену класса, могут получить доступ только другие static члены этого класса. Статические методы классов не имеют указателя this. Также их нельзя объявлять с модификаторами const и volatile. Статический метод класса можно вызвать для объекта ее класса или независимо от какого бы то ни было объекта, для обращения к ней достаточно использовать имя класса и оператор разрешения области видимости "::".

11. Методы открытия файлов для обмена информацией

Файловые операции ввода-вывода реализуются после записи в заголовок `<fstream>`, в котором определены все необходимые для этого классы и значения.

Файл открывается путем связывания его с потоком. Объявляем потоковый объект типа `ifstream`, для открытия выходного потока объявляем поток класса `ofstream`, поток, который предполагается использовать для операций как ввода, так и вывода, должен быть объявлен как объект класса `fstream`.

Чтобы открыть файл, используется функция `open()`.

Чаще всего файл открывается посредством конструкторов классов `ifstream`, `ofstream`, `fstream`.

`ifstream mystream("myfile");` - открытие файла для ввода при помощи конструкторов.

Чтобы закрыть файл используется функция `close()`.

```
mystream.close();
```

```
char str[80];
```

```
ifstream in ("test");
```

```
in>>str; Для начала нужно подключить библиотеку fstream.h:
```

```
#include <fstream.h> Потом создать файловый поток для необходимого файла:
```

```
ifstream myFStr; //для чтения
```

```
ofstream myFStr; //для записи
```

```
fstream myFStr; //для любых операций
```

Потом нужно связать поток с нужным файлом:

```
myFStr.open(filename [, метод открытия]);
```

Методы открытия:

ios::app ios::ate ios::binary ios::in ios::out ios::trunc

Константа	Описание
ios::in	открыть файл для чтения
ios::out	открыть файл для записи
ios::ate	при открытии переместить указатель в конец файла
ios::app	открыть файл для записи в конец файла
ios::trunc	удалить содержимое файла, если он существует
ios::binary	открытие файла в двоичном режиме

Их можно комбинировать логической операцией “ИЛИ”, т.е.

```
myFStr.open(filename, ios::in | ios::out);
```

```
//откроет файл filename для чтения и записи
```

```
myFStr.open(filename);
```

Потом нужно проверить, открылся ли файл:

```
if (!myFSTR) { //Файл не открылся, паникуем }
```

12. Встроенные методы классов

Встраиваемая функция-это небольшая по объему кода функция, код которой подставляется вместо ее вызова. Существует два способа создания встраиваемой (встроенной) функции. Первый способ состоит в использовании модификатора inline. Причиной существования встраиваемых функций является их эффективность. При каждом вызове обычной функции должна быть выполнена целая последовательность инструкций, связанных с обработкой самого вызова, включающего помещение ее аргументов в стек, и с возвратом из функции. В некоторых случаях значительное количество циклов центрального процессора используется именно для выполнения этих действий. Но если функция встраивается в строку программы, подобные системные затраты попросту отсутствуют, и общая скорость выполнения программы возрастает. Продемонстрируем использование встраиваемой функции.

```
#include <iostream>
```

```

using namespace std;

class Visitors {

int number;

public:

void put(int a);

int get();

};

inline int Visitors::get()

{

return number;

}

inline void Visitors::put(int a) {

number = a;

}

int main() {

Visitors a;

a.put(5);

cout << a.get();

}

```

Существует еще один способ создания встраиваемой функции. Он состоит в определении кода для метода класса в самом объявлении класса. Любая функция, которая определяется в объявлении класса, автоматически становится встраиваемой. В этом случае необязательно предварять ее объявление ключевым словом `inline`.

```

#include <iostream>

using namespace std;

class Visitors {

```

```

int number;

public:

void put(int a) {

number = a;

};

int get() {

return number;

};

};

int main() {

Visitors a;

a.put(5);

cout << a.get();

}

```

13. Объекты как параметры функций

Объекты можно передавать функциям в качестве аргументов точно так же, как передаются данные других типов. При этом будет создан локальный экземпляр объекта, являющийся независимой копией переданного объекта.

Следовательно, изменения, внесенные в объект при выполнении функции, не оказывают никакого влияния на объект, используемый в качестве аргумента для функции.

```

#include <iostream>

using namespace std;

class Person {

public:

string name;

};

```

```

void change(Person a) {

a.name = "b"; // изменяется локально

}

int main() {

Person a;

a.name = "a";

change(a);

cout << a.name; // "a"

}

```

Способ порождения нового объекта — через конструктор копирования. В случае сложных типов (особенно, имеющих в атрибутах указатели на динамические данные) копирующий конструктор, определённый по умолчанию, требуется перегрузить собственным (или явно запретить через помещение его в `private` зону класса).

Следует понимать, что для локального объекта при выходе из функции будет вызван деструктор.

Кроме того, возможна передача объекта по ссылке. В таком случае не будет создаваться локальный экземпляр объекта, а также изменения, внесенные в объект при выполнении функции, изменят объект, который использовался в качестве аргумента для функции.

```

#include <iostream>

using namespace std;

class Person {

public:

string name;

};

void change(Person &a) {

// не создаётся локальный объект

```

```

Person a.name = "b"; // изменяется внешний объект, переданный по ссылке
}

int main() {

Person a;

a.name = "a";

change(a);

cout << a.name; // "b"

}

```

14. Операторные функции - члены класса

Оператор - это некоторый метод выполняющий различные функции, которые заложены в самом коде C++. (обозвал бы это так) Существуют различные типы операторов, но они все описаны в самом языке C++.

Оператор является членом класса, если он описан в классе, а также перегружен. Можно сказать, что оператор, который ранее не умел взаимодействовать с объектами нашего класса, научился это делать. Структура такого оператора: `t::operator F(type p){something};`.

Как это реализовано на практике:

```

class intrnl

{

public:

intrnl(int c) {b=c;}

intrnl() {b=0;}

int gett(){return b;}

intrnl operator- (intrnl & a2)

{ return intrnl(b-a2.b); }

prin(){ cout << "struct=" << d.b << endl; }

```

```
private :
```

```
int b;
```

```
};
```

Здесь мы видим оператор вычитания (-), который является членом класса и умеет с ним взаимодействовать, в отличие от изначального оператора, которые не может выполнять эту функцию. К тому же оператор все еще может выполнять все свои предыдущие возможности.

15. Операторные функции – не члены класса

Оператор - это некоторый метод выполняющий различные функции, которые заложены в самом коде C++. (обозвал бы это так) Существуют различные типы операторов, но они все описаны в самом языке C++.

Оператор – не член класса, это либо обычный оператор, который никак не перегружен для работы с классом, то есть он не может взаимодействовать с объектами этого класса, также не членами класса являются дружественные операторы, которые объявляются в классе как дружественная, потом перегружаются, и функции которые не упоминаются в классе но способны взаимодействовать с классом посредством других методов. Структура такого оператора: `operator F(type1 p1, type2 p2){something}` (иногда с упоминанием `friend`).

Как это реализовано на практике:

```
class intrnl
```

```
{
```

```
public:
```

```
intrnl(int c) {b=c;}
```

```
intrnl() {b=0;}
```

```
int gett(){return b;}
```

```
friend ostream& operator<<(ostream& out, const intrnl & a);
```

```
private :
```

```
int b;
```

```
};
```

```

intrnl operator+ (intrnl a1, intrnl a2)

{ return intrnl(a1.gett()+a2.gett()); }

ostream& operator<< (ostream& out, const intrnl & a)

{

out<<a;

return out;

}

```

Здесь мы имеем 2 оператора не являющимися членами класса: сложение и оператор вывода.

Первый реализован при помощи других методов, которые реализованы в классе, т.е. сам класс не является членом класса, но может работать с ним. Второй пример – дружественная функция, она объявлена в классе, и описана. Но она не является членом класса, а просто имеет доступ к приватным полям класса. После таких манипуляций оператор получил возможность работы с классом.

16. Перегрузка операторов в классе

Цель перегрузки - повышение эффективности программы и упрощение ее кода. Операторы в C++ перегружаются с помощью функции operator. Перегрузка операторов в классе возможна через методы самого класса или же через дружественные функции. Форматы перегрузки:

Внутри класса: 1) friend тип_данных operator значение_оператора(список параметров){...}; 2) тип_данных operator значение_оператора(список параметров){...}; Вне класса: 1) friend тип_данных имя_класса::operator значение_оператора(список параметров); 2) тип_данных имя_класса::operator значение_оператора(список параметров); В случаях, когда вместо левого операнда(параметра) выступает не объект класса, мы не можем использовать метод класса. В данных случаях используем дружественные функции. Например, перегрузка оператора <<.

Примеры перегрузок.

```

#include <iostream>

#include <cstring>

#include <fstream>

```



```

using namespace std;

class Dog {
public:
    string name;
    int mass;

    Dog() :name(" "), mass(0) {};

    //Перегрузка оператора +;
    Dog operator+(const Dog& d) {
        Dog res;
        res.mass = d.mass + mass;
        return res;
    }

    //Перегрузка оператора >>;
    friend std::ostream& operator<< (ostream& out, const Dog& point);
};

ostream& operator<< (ostream& out, const Dog& dog)
{
    cout << "Вес собаки: " << dog.mass << "; Кличка: " << dog.name << endl;
    return out;
}

int main()
{
    Dog m;
    Dog b;
    m.mass = 10;

```

```

        b.mass = 3;

        Dog c = m + b;

        cout << c;

    }

```

(Пример пишите не полностью, только перегрузки.)

17. Функции – друзья класса

Существует возможность разрешить доступ к закрытым членам класса функциям, которые не являются членами этого класса. Для этого достаточно объявить эти функции дружественными(или друзьями) по отношению к рассматриваемому классу. Чтобы сделать функцию другом класса, необходимо включить ее прототип в public раздел объявления класса и предварить прототип ключевым словом friend. Пример: friend int find(...){}

Функция может быть другом нескольких классов.

Объявление, определение и использование функций-друзей

Есть два вида определяющих операций класса, задающих его интерфейс:

- функции-члены класса (методы)
- функции-друзья класса

И функции-члены, и функции-друзья имеют доступ к закрытой части класса. И те и другие должны быть объявлены в структуре класса.

18. Функции потокового ввода и вывода объектов

Поточный ввод-вывод в C++ выполняется с помощью функций сторонних библиотек.

Поток-это последовательный логический интерфейс, который связан с физическим файлом.

В C++ разработана новая библиотека ввода-вывода iostream, использующая концепцию объектно-ориентированного программирования:

Библиотека iostream определяет три стандартных потока:

- cin стандартный входной поток
- cout стандартный выходной поток
- cerr стандартный поток вывода сообщений об ошибках

В C++ также предусмотрены двухбайтовые (16-битовые) символьные версии стандартных потоков, именуемые wcin, xcout, wcerr, wclog. Они

предназначены для поддержки языков, для предоставления которых требуются большие символьные наборы.

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

- >> вводит информацию из потока
- << выводит информацию в поток

Оператор << называется оператором вывода или вставки, поскольку он вставляет символы в поток, а оператор >> называется оператором ввода или извлечения, поскольку он извлекает символы из потока.

19. Принципы объектно-ориентированного программирования.

На этих трех параметрах базируется объектно-ориентированное программирование.

Принцип наследования - один из трёх основных принципов объектно-ориентированного программирования. Он реализуется через механизмы наследования и виртуальных классов, Наследование в C++ происходит между классами и имеет тип отношений «является».

Класс, который наследует данные, называется подклассом (subclass), производным классом (derived class) или дочерним классом (child).

Класс, от которого наследуются данные или методы, называется суперклассом (super class), базовым классом (base class) или родительским классом (parent). Как ребенок получает характеристики своих родителей, производный класс получает методы и переменные базового класса.

Наследование полезно, поскольку оно позволяет структурировать и повторно использовать код, что, в свою очередь, может значительно ускорить процесс разработки.

Принцип инкапсуляции - второй принцип объектно-ориентированного программирования.

Инкапсуляция – это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.

В ООП код и данные могут быть объединены вместе; в этом случае говорят, что создаётся "чёрный ящик". Когда коды и данные объединяются таким способом, создаётся объект (object). Объект – это то, что поддерживает инкапсуляцию.

Внутри объекта коды и данные могут быть закрытыми (private). Закрытые коды или данные доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта. Если коды и данные являются открытыми, то, несмотря на то, что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть

объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов объекта.

Принцип полиморфизма (полиморфизм означает буквально многообразие форм) - ещё один принцип ООП. Он заключается в способности объекта во время выполнения программы динамически изменять свои свойства. Возможность настройки указателя на объект базового класса на объекты производных классов и механизм виртуальных функций лежат в основе этого принципа объектно-ориентированного программирования. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

20. Отношения классов

{ИЗ ЛЕКЦИИ КОЗЛОВА, ПРОЧИТАЙ И НАПИШИ СВОИМИ СЛОВАМИ}

Наследование обеспечивает отношения между классами типа предок/потомок; другие отношения тоже возможны. Когда мы разрабатываем новые классы из уже существующих, мы должны обозначать их отношения. Полезно определять, когда использовать наследование и, если и использовать, какой тип наследования лучше описывает отношения. Существует три базовых типа отношения классов.

Отношения “Это” (Is-a) - их мы используем, когда новый класс это частный представитель (подвид) главного класса, при таком отношении один класс расширяет (детализирует) возможности другого класса. Например, китайцы, американцы и РУССКИЕ - всё это люди, или мяч - это сфера. Что истинно для базового класса SphereClass, то истинно и для производного класса BallClass. Можно использовать экземпляр производного класса вместо экземпляра его базового класса, но не наоборот. Эта функция называется **совместимостью типов объектов** и применяется как к аргументам значений, так и к ссылочным функциям. Здесь использовано **публичное наследование**:

```
void DisplayDiameter(SphereClass One)
{
    cout << One.Diameter() << endl;
}
```

```
SphereClass Sphere;
BallClass Ball;
```

```
DisplayDiameter(Sphere);
DisplayDiameter(Ball);
{по идее выведет и то и то, если class SphereClass: public BallClass}
```

Отношения “Содержит” (Has-a) - их мы используем когда один класс **содержит в себе** экземпляр другого. При данных отношениях существует два типа взаимодействия классов:

- 1) **Агрегация** - объект (экземпляр) нового класса, который объявлен в классе, не является его составной частью и его использование не влияет на функциональную работу класса.
- 2) **Композиция** - объект, который объявлен в классе, является составной частью этого класса.

Например, в шариковой ручке есть шарик, но, конечно же, сама ручка шаром не является. Мы можем определить поле данных Point типа BallClass внутри класса PenClass.

```
class PenClass
{
    ...
    BallClass Point;
}
```

Отношения “Использует” (Uses) - их мы используем, когда один класс содержит программный код другого вложенного класса, к которому он имеет доступ.

21. Доступность членов класса в классах-наследниках

*В зависимости от того, какой модификатор доступа стоит перед полями и методами класса, класс-наследник этого класса может иметь или не иметь к ним доступ. Класс-наследник имеет доступ к полям и методам родительского класса с модификаторами доступа **public** и **protected** и не имеет доступа к ним, если к ним относится модификатор **private**.*

В конструкции для создания класса-наследника используется модификатор наследования. В зависимости от того, каким является этот модификатор, класс-наследник унаследует поля и методы либо с такими же модификаторами доступа, либо с измененными. Изменение модификаторов доступа происходит согласно следующей таблице.

	Исходный модификатор доступа		
	<i>public</i>	<i>private</i>	<i>protected</i>
<i>Public-наследование</i>	public	private	protected
<i>Private-наследование</i>	private	private	private

Protected-наследование	protected	private	protected
------------------------	-----------	---------	-----------

Так как в классе-наследнике модификаторы доступа к полям и методам могут измениться по сравнению с соответствующими из класса родителя, это может повлиять на доступ “класса-внука” к полям своего родителя: он может не иметь доступа к некоторым членам своего родителя, но правила имени доступа остаются такими же, как были в первом абзаце.

22. Типы наследования классов

Наследование—один из трех фундаментальных принципов объектно-ориентированного программирования, поскольку именно благодаря ему возможно создание иерархических классификаций.

Простое наследование

Класс, от которого произошло наследование, называется *базовым* или *родительским*. Классы, которые произошли от базового, называются *потомками*, *наследниками* или *производными классами*

В некоторых языках используются абстрактные классы. *Абстрактный класс* — это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. Т.е. от Абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник офиса», от которого наследуются классы «бухгалтер», «директор» и т.д. Т.к. производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «бухгалтер», «директор», но нет смысла создавать объект на основе класса «сотрудник офиса».

Множественное наследование

Множественное наследование позволяет одному дочернему классу иметь несколько родителей. Предположим, что мы хотим написать программу для отслеживания работы учителей. Учитель — это Human. Тем не менее, он также является Сотрудником .

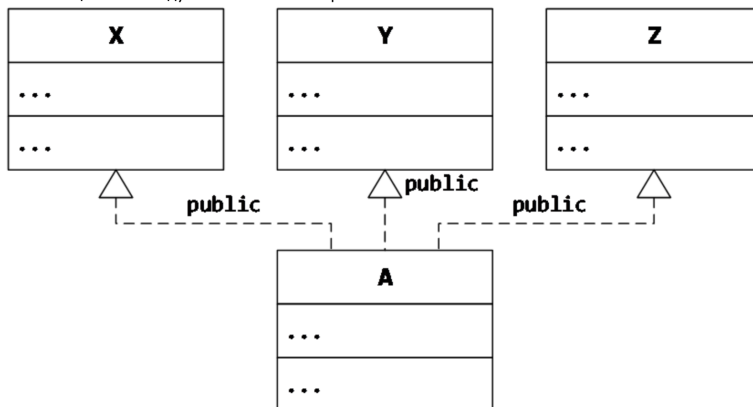
Множественное наследование позволяет порожденному классу наследовать элементы более, чем от одного базового класса. Синтаксис заголовков классов расширяется так, чтобы разрешить создание списка базовых классов и обозначения их уровня доступа

```

class X
{...};
class Y
{...};
class Z
{...};
class A : public X, public Y, public Z
{...};

```

Класс A обобщенно наследует элементы всех трех основных классов.



Для доступа к членам порожденного класса, унаследованного от нескольких базовых классов, используются те же правила, что и при порождении из одного базового класса.

Хотя множественное наследование кажется простым расширением одиночного наследования, оно может привести к множеству проблем, которые могут заметно увеличить сложность программ и сделать кошмаром дальнейшую поддержку кода. Проблемы могут возникнуть в следующих случаях:

1. если в порожденном классе используется член с таким же именем, как в одном из базовых классов;
2. когда в нескольких базовых классах определены члены с одинаковыми именами.

В этих случаях необходимо использовать оператор разрешения контекста для уточнения элемента, которому осуществляется доступ, именем базового класса.

Большинство задач, решаемых с помощью множественного наследования, можно решить и с использованием одиночного наследования. Многие объектно-ориентированные языки программирования (например, Smalltalk, PHP) даже не поддерживают множественное наследование.

Многие, относительно современные языки, такие как Java и C#, ограничивают классы одиночным наследованием обычных классов, но допускают множественное наследование интерфейсных классов. Суть идеи, запрещающей множественное наследование в этих языках, заключается в том, что это излишняя сложность, которая порождает больше проблем, чем удобств.

Наследование в языке C++

Права доступа при наследовании ведут себя по-разному в зависимости от **типа наследования**.

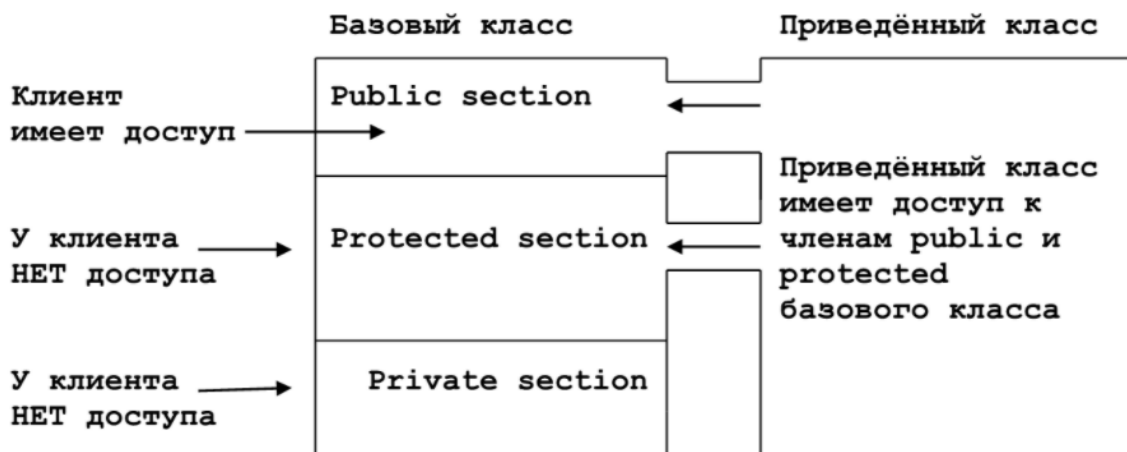
Для спецификации доступа в C++ есть три уровня: **private**, **protected** и **public**. Само наследование также может быть **private**, **protected** и **public**.

По умолчанию классы наследуются, как **private**, а структуры - как **public**.

Важно понимать, что независимо от типа наследования, ни один из наследников не сможет получить доступ к **private**-данным базового класса.

- при **public**-наследовании все спецификаторы остаются без изменения.
- при **protected**-наследовании все спецификаторы остаются без изменения, кроме спецификатора **public**, который меняется на спецификатор **protected** (то есть **public**-члены базового класса в потомках становятся **protected**).
- при **private**-наследовании все спецификаторы меняются на **private**.

Тип наследования	Раздел члена базового класса	Видимость компонента в приведённом классе
private	private	недоступен
	protected	private
	public	private
protected	private	недоступен
	protected	protected
	public	protected
public	private	недоступен
	protected	protected
	public	public



23. Раннее и позднее связывание классов

Раннее связывание - связывание происходящее во время компиляции.

Позднее связывание - связывание происходящее во время исполнения программы.

В случае вызова функции одного названия в обоих классах(при этом один - родитель, другой - ребенок) при раннем связывании будет использована функция из родителя.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class A
6  {
7  public:
8      char* getName() { return "A"; }
9  };
10
11 class B : public A
12 {
13 public:
14     char* getName() { return "B"; }
15 };
16
17 int main()
18 {
19     B b;
20     A &rParent = b;
21     cout << rParent.getName();
22 }
```

```
A
...Program finished with exit code 0
Press ENTER to exit console.
```

Для решения данной проблемы требуется использование виртуальных функций.

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class A
6  {
7  public:
8      virtual char* getName() { return "A"; }
9  };
10
11 class B : public A
12 {
13 public:
14     virtual char* getName() override { return "B"; }
15 };
16
17 int main()
18 {
19     B b;
20     A &rParent = b;
21     cout << rParent.getName();
22 }

```

```

B
...Program finished with exit code 0
Press ENTER to exit console.

```

24. Виртуальные методы классов

Виртуальный метод класса – метод базового класса, который можно переопределить для классов-наследников, чтобы при вызове метода с соответствующим названием для класса-наследника была осуществлена именно та реализация метода, которая соответствует этому классу-наследнику. То есть вариант реализации метода будет определяться во время работы программы.

Использование виртуальных методов класса является одним из важнейших приемов реализации полиморфизма. Они позволяют создать общий код для базового класса и его наследников, таким образом их использование сокращает длину итогового кода.

Чтобы сделать функцию виртуальной, нужно перед ее объявлением напечатать слово **virtual**. Все функции, являющиеся вариантами переопределения виртуальной функции, также являются виртуальными. Типы возврата и параметры виртуальной функции и ее вариантов переопределения

должны совпадать, иначе не произойдет переопределения. Чтобы избежать ошибок при переопределении виртуальной функции, можно после объявления варианта переопределения функции печатать **override**. Тогда компилятор сообщит об ошибке, если функция будет неправильно переопределена. Не следует использовать виртуальные функции в конструкторах или деструкторах объектов классов, потому что при вызове метода класса будет исполняться реализация “класса-родителя”.

25. Абстрактные классы и их применение

Абстрактный класс – класс, который содержит в себе хотя бы одну чисто виртуальную функцию и ни один экземпляр которого нельзя создать. Чисто виртуальная функция – метод класса, тело которого не определено. Хотя в абстрактном классе и не определена чисто виртуальная функция, ее варианты переопределения обязательно должны быть определены в классах-наследниках, которые по задумке не должны быть абстрактными. Чисто виртуальные функции объявляются так же, как и обычные, но до типа возвращаемых данных печатается слово **virtual**, а после () печатается = 0; вместо тела функции.

Абстрактные классы применяются для создания классов-наследников, которые имеют один и тот же метод, но который может по-разному в них реализовываться. То, что указатель на базовый (в нашем случае – еще и абстрактный) класс может хранить адреса и ссылки на любых классов-наследников, позволяет нам создать общий код для базового класса и его наследников, сокращая тем самым общую длину кода и экономя наше время. Помимо этого такой код проще редактировать или исправлять в нем логические ошибки, ведь достаточно изменить его в одном месте, а не для каждого класса-наследника в отдельности. Также с помощью абстрактных классов можно реализовать один из принципов ООП – полиморфизм.

26. Сравнение статического и динамического связывания классов

В объектно-ориентированном программировании есть 2 вида связывания: раннее и позднее.

Раннее связывание означает, что вся информация, необходимая для вызова функции или классического объекта определяется во время компиляции. Примерами раннего связывания могут служить вызовы перегруженных (наследуемых) методов класса. Именно из-за того, что определение используемой функции происходит во время компиляции, а невыполнения

программы, экономится некоторое количество времени. Главный недостаток такого связывания - отсутствие гибкости.

Позднее связывание означает, что точное решение о вызове функции будет принято во время выполнения программы. Для того, чтобы обычная функция, например `int get();` могла использоваться в позднем связывании, достаточно перед `int` дописать `virtual`. Таким образом функция станет виртуальной и определение этой функции будет происходить во время выполнения программы.

Преимущество позднего связывания состоит в том, что оно обеспечивает большую степень гибкости. Его можно применять для поддержки общего интерфейса и разрешать при этом различным объектам, которые используют данный интерфейс, определять их собственные реализации. Но к его недостаткам можно отнести, понижение скорости выполнения программ, из-за того, что каждый раз программе придется искать виртуальную функцию в таблице этих самых виртуальных функций.

Если подводить итог, то позднее связывание лучше использовать, когда необходимо улучшить (упростить) работу с программой, лучше организовать ее структуру, а ранее лучше использовать, когда важно увеличить скорость работы кода. Какое из связываний выбрать - решение программиста в каждой конкретной ситуации.

27. Основные правила построения абстрактных классов.

Абстрактные классы — это классы, которые содержат или наследуют без переопределения хотя бы одну чистую виртуальную функцию. Чистая виртуальная функция – функция, тело которой неопределено и объявление которой завершается значением `"=0"`:

```
virtual тип имя_функции (список_параметров) = 0;
```

Пример (абстрактный класс, который представляет геометрическую фигуру):

```
1 class Figure
2 {
3     public:
4         virtual double Square() = 0;
5         virtual double Perimeter() = 0;
6     };
7
```

Класс `Figure` содержит 2 чистые виртуальные функции. Реализацию этих функций должны определять классы-наследники. Объект абстрактного класса

создать нельзя. Для построения и использования абстрактного класса необходимо наличие производных классов.

Объявление производного класса может быть выполнено в трех видах:

```
class имя_производного_класса : private имя_базового_класса
```

```
{body of class}
```

```
class имя_производного_класса : protected имя_базового_класса
```

```
{body of class}
```

```
class имя_производного_класса : public имя_базового_класса
```

```
{body of class}
```

Пример (с абстрактным базовым классом Figure и классом наследником Rectangle):

```
#include <iostream>
```

```
using namespace std;
```

```
class Figure // базовый абстрактный класс с двумя виртуальными функциями
```

```
{
```

```
public:
```

```
    virtual double Square() =0;    // чистые
```

```
    virtual double Perimeter() =0; // виртуальные функции
```

```
};
```

```
class Rectangle : public Figure // класс наследник абстрактного класса Figure
```

```
{
```

```
private:
```

```
    double a;
```

```
    double b;
```

```
public:
```

```
    Rectangle(double A, double B) : a(A), b(B)
```

```
    {}
```

```
    //реализация виртуальных функций
```

```

double Square() override
{
    return a * b;
}

double Perimeter() override
{
    return a * 2 + b * 2;
}

};

int main()
{
    Rectangle rectangle(10, 20);

    cout << rectangle.Square() << endl;

    cout << rectangle.Perimeter() << endl;

    return 0;
}

```

28. Файлы и потоки.

Файл – последовательный набор записей, хранимый на внешних накопителях. В C++ файловые операции ввода и вывода можно реализовать после включения в программу заголовка `<fstream>`, в котором определены все необходимые классы и значения. В C++ файл открывается путем связывания его с потоком. Существуют три типа таких данных:

`<ifstream>` - представляет собой поток входных файлов и используется для чтения информации из файлов;

`<ofstream>` - представляет собой поток выходных файлов и используется для создания файлов, а также для того, чтобы записывать информацию в файлы;

`<fstream>` - представляет поток файлов; он может использоваться для операций как ввода, так и вывода, т.е. может проделывать несколько операций с файлами;

Прежде чем работать с файлом, его нужно открыть и для этого используют функцию `open()`. В первом аргументе функции указывают имя и расположение файла, а во втором – режим, в котором должен открыться файл. Таких режимов бывает несколько:

`ios::in` – открыть файл для чтения;

`ios::out` – открыть файл для записи;

`ios::app` – весь вывод будет добавлен в самый конец файла;

Обычно файлы открывают посредством конструкторов классов `ifstream`, `ofstream`, `fstream`. Чтобы закрыть файл, используется функция `close()`.

29. Прямой доступ к отдельным записям в файлах.

Существуют несколько функций доступа к отдельным записям в файлах. Функция `get()` считывает символ из файла. На примере можно проанализировать, как она работает: `istream &get(char &text)`. `get()` считывает один символ из потока и перемещает его значение в переменную. Она возвращает ссылку на поток, связанный с заранее открытым файлом. При достижении конца файла значение ссылки будет равняться нулю.

Для считывания блока двоичных данных, используют функцию-члена `read()`. Функция `read()` считывает байт данных из связанного с файлом потока и помещает их в буфер, адресуемый параметром. Обнаружить конец файла можно с помощью функции-члена `eof()`.

Рассмотрим один из примеров применения прототипа перегруженных версий функции `get()`: `istream &get(char *avg, streamsize tel)`. Функция считывает символы в массив, заданный параметром `avg`, до тех пор, пока не будет считано `(tel-1)` символов или не встретится символ новой строки, или не будет достигнут конец файла. После выполнения функции `get()` массив, адресуемый параметром `avg`, будет иметь завершающий нуль-символ. Символ новой строки, если такой обнаружится в потоке, не извлечется.

Также существует функция `getline()`, которая представляет собой еще один способ ввода данных и ее различие между функцией `get()` состоит в том, что `getline()` считывает и удаляет символ-разделитель из входного потока, а `get()` – нет. Функция `seekg()` перемещает указатель, отвечающий за ввод данных.

30. Особенности сортировки файлов.

Файл – последовательный набор записей, хранимый на внешних накопителях. Фактически максимальные размеры файла ограничены техническими возможностями компьютера, но они столь велики, что могут считаться бесконечными.

Известные методы сортировки массивов не работают с файлами по следующим причинам:

1) Массив осуществляет прямой доступ к своим элементам и предоставляет возможность их свободно изменить. В случае файлов это не прокатит: данные в них хранятся последовательно, и к отдельным записям доступ отсутствует.

В файлах поиск данных происходит перебором значений от точки начала до момента, пока не найдётся нужная запись.

2) Объём файла часто может превышать размеры оперативной памяти. Вариант работать с файлами во внешней памяти не эффективен, т.к. внешняя память очень медленная.

Всё это свидетельствует о том, что для сортировки файлов нужны некоторые более эффективные методы.

Алгоритм сортировки файлов:

Файл состоит из опр. записей, и все они какого-то размера. С некоторым количеством записей можно работать в оперативной памяти как с массивом. Размер массива зависит от величины ОП и отрезков. И так каждый массив быстро сортируется. Получили подфайлы. Из оперативной памяти переместить массив в отдельный файл. Повторить М раз, пока не получится пройти весь файл.

$M = \text{всего_записей_в_файле} / \text{размер_массива}$.

Все М подфайлов отсортированы, но между собой не связаны. Дальнейшие манипуляции над подфайлами осуществляют различные методы сортировок.

31. Сортировки файлов методом (сбалансированного) слияния.

1) Начальный файл с диска разделяется на подфайлы такого размера, чтобы их можно было разместить в оперативной памяти, где они и сортируются. После сортировке во внутренней памяти подфайл размещается на диске А под названием А1. И следующий подфайл сортируется в ОП, после чего перемещается на диск Б под именем Б1. Третий подфайл А2 записывается на диск А, и следующий после него Б2 – на диск Б, и так далее: А3, Б3, ..., пока исходный файл не израсходуется. Так получено некоторое число отсортированных подфайлов, но между собой они не связаны.

2) Подфайлы A1, B1 сливаются в подфайл D1, который записывается на диск D.

На диск E записывается подфайл E1, полученный слиянием A2 и B2.

Таким образом:

$A_3 + B_3 = D_2$ на D

$A_4 + B_4 = E_2$ на E

$A_5 + B_5 = D_3$ на D, и тд.

Как происходит слияние? Для примера возьмём пару подфайлов A1 и B1.

Сравниваем первые элементы из A1 и B1. Если, например, элемент из A1 больше, то перемещаем его в D1. Теперь сравниваем следующий элемент из A1 и всё ещё первый из B1, перемещаем в D1 наибольший из них. И так до тех пор, пока в A1 и B1 ничего не останется.

Таким образом, число подфайлов уменьшается в 2 раза в результате выполнения 2 пункта.

3) После нескольких повторений 2) получится итоговый файл, соразмерный исходному, но уже отсортированный.

Эффективность метода.

-Уменьшение количества чтений и записей в файл.

Каждый элемент исходного файла единожды считывается в оперативную память, один раз перемещается из ОП во внешнюю память, а после этого достаточно просто сливаются.

Сложность сортировки – двоичный логарифм, это даже меньше линейной функции.

32. Осциллирующая (колебательная) сортировка.

1)Возьмём из исходного файла некоторую часть элементов, разделим их на 3 массива, отсортируем их в ОП по возрастанию.

Во внешней памяти создадим 4 подфайла: A,B,C,D.

На подфайлы A,B,C перенесём по массиву, а подфайл D остается пустым.

Произведём слияние A,B,C в подфайл D.

(Процесс слияния: сравниваются первые элементы подфайлов A,B,C, и нужный из них (в нашем случае – наименьший) перемещается в подфайл D. Так, три подфайла сравнивались до тех пор, пока в итоге не опустеют, и D не станет их отсортированной комбинацией).

2)Но изначальный файл еще не опустел. Снова вытащим из него 3 массива и припишем их в подфайлы A,B и конец подфайла D. Подфайл C остаётся пустой, но в него мы сливаем подфайлы A,B и конец подфайла D (то, что мы записали в него последним). Получается, что последние 3 массива, взятые из основного файла, скомбинированы и отсортированы в подфайле C. Подфайлы A,B пустые.

Снова вытащим из оригинального файла 3 массива, разложим их так: 1й массив пойдёт в подфайл A, 2й – в конец подфайла C, 3й – в конец подфайла D. Сольём эти массивы в пустой подфайл B, после чего подфайл A остается пустым, а подфайл B содержит отсортированную комбинацию последних 3х массивов.

Итак, наша ситуация: подфайл A пуст, подфайлы B,C,D длинные и отсортированные. Сольём их в подфайл A, получим еще более длинный, но тоже упорядоченный подфайл.

3) Если оставшийся подфайл A не равен по размеру исходному файлу, повторять 2), пока этого не случится.

Сложность - логарифм по основанию 3.

33. Многофазная сортировка.

1) Из первых элементов исходного файла создаётся максимально длинный массив, который может обработать ОП. Он сортируется и отправляется в подфайл на диске.

2) пункт 1) повторяется несколько раз, создавая подфайлы A_i на диске A и подфайлы B_i на диске B, пока исходный файл не исчерпается. Итак, количество подфайлов на дисках A и B было либо одинаково, либо отличалось на 1. Тем не менее есть и другой, более эффективный способ расположения подфайлов на дисках:

Наибольшую эффективность можно получить при условии, если суммарное количество подфайлов равно числу из последовательности Фибоначчи: $F_i = F_{i-1} + F_{i-2}$. Тогда по этой формуле нужно, чтобы на диске A было записано F_{i-1} подфайлов, а на диске B записано F_{i-2} подфайла.

Если сумма всех исходных подфайлов не принадлежит последовательности Фибоначчи, то нужно просто в неё добавить несколько пустых подфайлов, чтобы в итоге получилось следующее число Фибоначчи: F_{i+1} .

3) Теперь имеется несколько подфайлов на дисках А и В. Следующее действие – постепенное слияние каждой пары подфайлов двух дисков в некоторый третий диск, пока не останется один файл. Это будет итоговый файл.

Применение этого метода при сортировке последовательности позволяет значительно сократить время, необходимое для обработки данных. Однако очевидно, что реализация многофазной сортировки требует значительных трудозатрат на кодирование и отладку программного кода. Поэтому применение данной сортировки целесообразно при большом объеме данных и необходимости минимизации времени, затрачиваемого на сортировку. В том случае, если объем данных небольшой, а при современных вычислительных мощностях это может быть до нескольких сот тысяч элементов, рекомендуется использовать более простые методы внешней сортировки.

По сложности - числа Фибоначчи.

34. Оценка сложности алгоритмов сортировки файлов.

Рассмотрим сортировку файлов методом сбалансированного слияния. После каждого шага (слияния всех подфайлов одинаковой длины) количество подфайлов уменьшается вдвое (а длина этих подфайлов увеличивается вдвое). Таким образом, сложность сортировки файла методом слияния – $O(\log_2 (n))$, где n – это количество подфайлов, на которые разделен изначальный файл.

При сортировке файла методом осцилирующей (или колебательной) сортировки, мы сливаем подфайлы по три, из чего следует, что после каждого шага количество подфайлов уменьшается втрое. Таким образом, сложность сортировки файла методом колебательной сортировки – $O(\log_3 (n))$, где n – это количество подфайлов, на которые разделен изначальный файл.

Можно сделать вывод, что колебательная сортировка быстрее сортировки методом слияния (хоть и не на много).

35. Разреженные матрицы: определение и области применения

Разреженная матрица - матрица, у которой многие элементы равны нулю.

Разреженные матрицы возникают при решении многих прикладных задач.

1) дискретизация уравнений математической физики - разностные схемы и метод конечных элементов.

2) задачи линейного программирования (теория оптимизации).

3) задачи теории электрических цепей.

Оптимизировать процесс решения разреженных матриц с точки зрения затрат машинной памяти и времени можно за счет того что

1) арифметические операции с нулями не производятся.

2) нули не обязательно хранить в машинной памяти.

Способ использования разреженности очевиден для итерационных методов вычислительной линейной алгебры, основной операцией которых является умножение матрицы на вектор. Для решения линейных систем с разреженными матрицами более общего вида разработан ряд алгоритмов. Они базируются на известных прямых методах – методе Гаусса, ортогональных методах – и ставят целью по возможности уменьшить заполнение матрицы.

36. Способы хранения разреженных матриц

1. Дональд Кнут предложил следующую идею хранения разреженной матрицы (немного измененный пример из лекции):

		1	2	3	4
A	1	9		5	
	2				8
	3	6		5	3
	4			7	

Pos.	1	2	3	4	5	6	7	
AN	9	5	8	6	5	3	7	Ненулевые элементы
I	1	1	2	3	3	3	4	Соответствующий номер строки
J	1	3	4	1	3	4	3	Соответствующий номер столбца
NR	2	0	0	5	6	0	0	Позиция в AN следующего ненулевого элемента этой же строки
NC	4	5	6	0	7	0	0	Позиция в AN следующего ненулевого элемента этого же столбца
ER	1	3	4	7				Позиция в AN первого ненулевого элемента строки
EC	1	0	2	3				Позиция в AN первого ненулевого элемента столбца

Такой подход наиболее удобен в случае, когда алгоритм не может предсказать где будут расположены будущие ненулевые элементы

Иногда бывает полезным в списках NR и NC в случае, когда текущий элемент – последний ненулевой элемент в строке/столбце, хранить не 0, а позицию первого ненулевого элемента в этой строке/столбце, тем самым зациклить обход строки/столбца. Т.е. для нашего примера:

NR	2	1	3	5	6	4	7
NC	4	5	6	1	7	3	2

2. Еще один способ хранения такой матрицы – “сжатый линейный формат”: если ненулевые элементы храним по рядам – “сжатый строчный формат”, по столбцам – “сжатый столбцовый формат”. Рассмотрим на примере строчного формата (по столбцам аналогично): ненулевые элементы исходной разреженной матрицы храним строго по рядам в массиве AN, массив JA хранит соответствующие номера столбцов и,

наконец, новый массив IA хранит позиции элементов в AN, с которых начинается каждая строка. Последний элемент в IA – дополнительный - первая свободная позиция в AN и JA, таким образом длина IA на единицу больше чем количество строк в исходной матрице. При этом, если $IA[k] = IA[k+1]$, то это значит, что строка k – пустая. Для IA бывает удобнее использовать не массив, а список.

Немного измененный пример из лекции:

		1	2	3	4	5	6	7
A	1		1		9		5	
	2							
	3				6			3

Pos.	1	2	3	4	5	(6)
AN	1	9	5	6	3	
JA	2	4	6	4	7	
IA	1	4	4	6		

Такой способ хранения становится очень эффективным для операции транспонирования.

37. Алгоритм сложения разреженных векторов



Сложение разреженных векторов

Первый алгоритм.

Допустим мы имеем два разреженных вектора. Выпишем в массив AN ненулевые элементы первого вектора, а в массив BN - ненулевые элементы второго вектора. В массив JA запишем номера позиций, на которых находились ненулевые элементы первого вектора, в массив JB - номера позиций, на которых находились ненулевые элементы второго вектора. Причем нумерация позиций пусть будет начинаться с 1 (для удобства в будущем, но на самом деле это непринципиально). Создадим массив X такой длины, каким было значение наибольшего числа из массивов JA и JB. Положим



массивов JA и JB. Положим элементы массива AN в массив X на места, соответствующие номерам позиций в исходном разреженном векторе. Если существует такая позиция, что на ней в первом векторе был нулевой элемент, а во втором - ненулевой, то на этой позиции в массиве X положим нулевой элемент. Остальные, не упомянутые выше позиции, оставим пустыми (там были и будут нули). На следующем шаге к элементам массива X прибавляются элементы массива BN на соответствующих позициях. Составим массив JC, состоящий из элементов, чье множество является совокупностью чисел массивов JA и JB. Создадим массив CN, состоящий из элементов, входящих в массив X



22:13



элементов, входящих в массив X (ненулевых элементов и "паразитных" нулей), и длина которого равна длине массива JC . Позиции этих элементов будут такими же, какие позиции будут у элементов массива JC , обозначающих позицию суммы в массиве X . Информацию о "паразитных" нулях можно не хранить, поэтому из массива CN можно вычеркнуть их, а из массива JC - элементы, равные их позиции в массиве X . Таким образом получим итоговые массивы CN и JC , полностью описывающие результат сложения разреженных векторов. Сложность линейная. Второй алгоритм. Так же создаётся массив JC , но сложение будем производить сразу в массиве CN . Создадим вместо массива X целочисленный массив



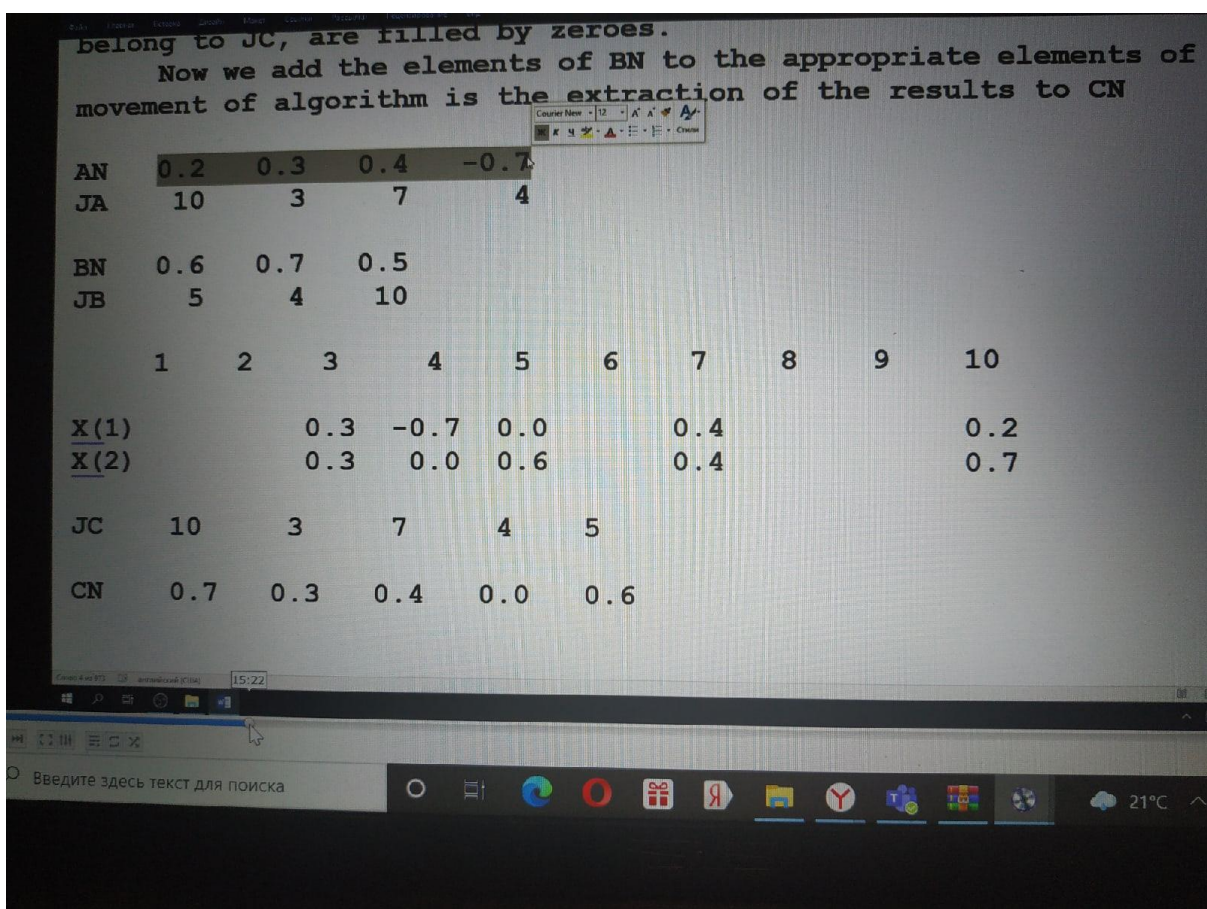
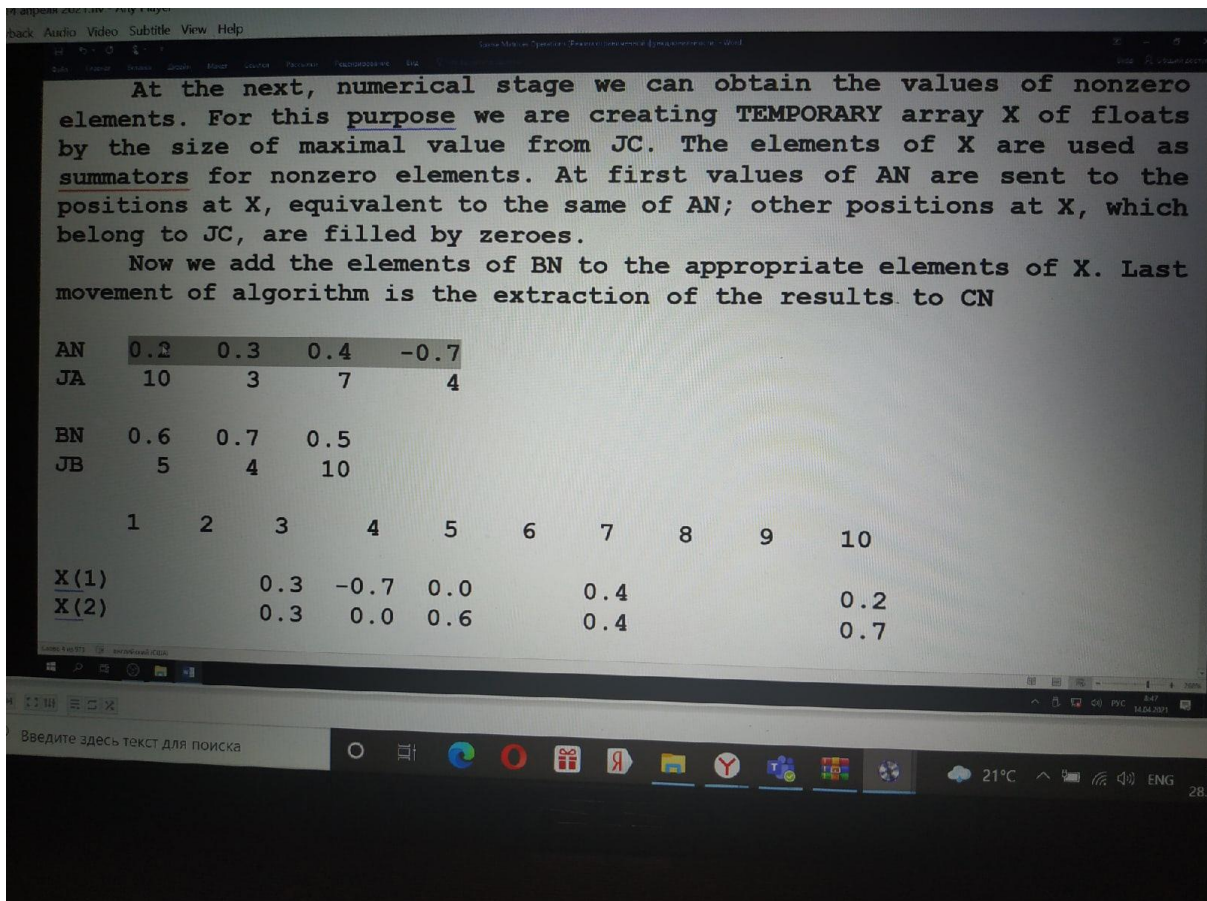
22:13



так же создается массив JC , но сложение будем производить сразу в массиве CN . Создадим вместо массива X целочисленный массив IP такой же длины, как и X в прошлом алгоритме. В этом массиве на позициях, равных элементам массива JC , будут стоять элементы, равные индексам этих самых элементов массива JC . На пустых местах будут нули. Сначала элементы массива CN (состоящего вначале из нулей) складываются с элементами массива AN , а затем с элементами массива BN . Так получается итоговый вектор CN , который с вектором JC описывает сумму исходных разреженных векторов.



Описание by Дениска. Теперь скрины из презы козлова



Another variant of addition algorithm uses temporary array JC and INTEGER indexes IP. At first we create array JC as at previous algorithm and then store the indexes of JC at vector IP. For example, the element of C is stored at 2-d position of CN. In this algorithm numerical stage consists of 3 steps:

	1	2	3	4	5	6	7	8	9	10
IP		2	4	5			3			1

Creating EMPTY array CN $CN[IP[JC[k]]] = 0.0$
 Addition AN to CN $CN[IP[JA[k]]] += AN[k]$
 Addition BN to CN $CN[IP[JB[k]]] += BN[k]$

	1	2	3	4
AN	0.2	0.3	0.4	-0.7
JA	10	3	7	4
BN	0.6	0.7	0.5	

	1	2	3	4	5	6	7	8	9	10
JB		5	4	10						
X(1)			0.3	-0.7	0.0		0.4			0.2
X(2)			0.3	0.0	0.6		0.4			0.7
JC	10		3	7	4	5				
CN	0.7	0.3	0.4	0.0	0.6					

$X[JC[k]] = 0.0; \quad X[JA[k]] += AN[k]; \quad X[JB[k]] += BN[k];$

$CN[k] = X[JC[k]];$

This algorithm doesn't suppress the parasite zeroes. Its complexity is compared by the length of compressed array and is LINEAR.

Another variant of addition algorithm uses temporary array JC and INTEGER indexes IP. At first we create array JC as at previous algorithm

38. Алгоритм скалярного умножения разреженных векторов

Пусть имеем два разреженных вектора A и B (немного измененный пример из лекции):

Pos.	1	2	3	4	5
AN	-0.5	0.4	0.1	-0.2	0.2
JA	10	7	3	5	9
BN	0.3	0.4	0.5		
JB	5	4	10		

1. Необходимо создать целочисленный массив индексов IP и перенести в него информацию из JA:

	1	2	3	4	5	6	7	8	9	10
IP			3		4		2		5	1

2. Реализуем цикл по элементам JB с условием:

```
if ( IP[JB[k]] != 0)
    S += BN[k] * AN[IP[JB[k]]]
```

3. S – результат скалярного произведения

Если же нужно один вектор A умножить несколько раз на вектор B или на несколько других векторов, то нам достаточно один раз создать массив IP и использовать его каждый раз. Сложность алгоритма зависит от длины разреженного вектора B, т.е. линейная сложность, при этом количество операций довольно мало и будет еще меньше, если в массив IP поместить разреженный вектор наибольшей длины, т.к. на шаге 2 в цикле будет меньше действий.

Алгоритм:

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

int main() {

    int dim, total = 0;
    cout<<"Insert Dimension \n";
    cin>>dim;
    float *v1, *v2;

    v1 = new float [dim];
    v2 = new float [dim];

    cout<<"values vector 1 \n";
    for(int i = 0; i < dim; i++){
        v1[i] = rand() % 100;
        cout<< v1[i] <<endl;
    }
    cout<<"values vector 2 \n";
    for(int j = 0; j < dim; j++){
        v2[j] = rand() % 100;
        cout<< v2[j]<<endl;
    }

    for(int k = 0; k<dim; k++){
        total= total+v1[k]*v2[k];
    }

    cout<<"Result : "<<total<<endl;
    return 0;
}
```

{скоро добавим комментарии}

39. Транспонирование разреженных матриц

“Сжатый линейный формат” – один из способов хранения разреженной матрицы, который позволяет реализовать очень эффективный алгоритм транспонирования матрицы, при котором элементы матрицы не только не меняют своего положения, но и не меняют своего значения.

Пусть имеем разреженную матрицу A (немного измененный пример из лекции):

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} & * & & & * & \\ * & & * & & & \\ & * & & * & & * \\ * & & & & * & \\ & & * & & & \end{bmatrix} \end{matrix}$$

Здесь * - ненулевые элементы
В данном случае нам без разницы какое значение имеет данный элемент

IA	1		3		5		8	10	11
JA	5	2	1	3	4	2	6	1	5
Row	1	2	3		4		5		

JA – номера столбцов соответствующих ненулевым элементам, IA – стартовые точки строк.

Главная идея заключается в том, чтобы по данному представлению матрицы найти представление транспонированной матрицы, при этом не меняя исходной. При транспонировании строки переходят в столбцы и наоборот, тогда:

1. Создадим 6 списков (в общем случае число списков равно числу столбцов исходной матрицы) и заполним их соответствующим образом:
 $IA[1] = 1, IA[2] = 3 \Rightarrow JA[1] = 5, JA[2] = 2$ – номера столбцов тех элементов, которые лежат в первой строке \Rightarrow в списки под номерами 2 и 5 в столбец 1 запишем 1. Аналогично $IA[2] = 3, IA[3] = 5 \Rightarrow JA[3] = 1$ и $JA[4] = 3$ – номера столбцов тех элементов, которые лежат во второй строке \Rightarrow в списки под номерами 1 и 3 во вторую позицию запишем 2 и т.д.

Номер списка	Pos.	1	2	3	4	5
	1		2		4	
	2	1		3		
	3		2			5
	4			3		
	5	1			4	
	6			3		

Таким образом, номера списков обозначают номера столбцов в исходной матрице, а числа напротив – строки, в которых находятся ненулевые элементы. Соответственно для транспонированной матрицы номера этих списков – номер строки, а числа напротив – номер столбца

2. Записываем представление полученной разреженной матрицы:

JAT	2	4	1	3	2	5	3	1	4	3
Column	1		2		3		4	5		6

Алгоритм:

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

int main() {

    int matrix[10][10], transpose[10][10], row, column;
    cout << "Enter rows and columns of matrix\n";
    cin >> row >> column;

    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < column; ++j) {
            matrix[i][j] = rand() % 100;
        }
    }

    cout << "Entered Matrix: \n" << endl;
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < column; ++j) {
            cout << " " << matrix[i][j];
            if (j == column - 1)
                cout << endl << endl;
        }
    }

    for (int i = 0; i < row; ++i)
        for (int j = 0; j < column; ++j) {
            transpose[j][i] = matrix[i][j];
        }

    cout << "Transpose \n" << endl;
    for (int i = 0; i < column; ++i){
        for (int j = 0; j < row; ++j) {
            cout << " " << transpose[i][j];
            if (j == row - 1)
                cout << endl << endl;
        }
    }
    return 0;
}
```

Результат:

```
Enter rows and columns of matrix: 4
3
Entered Matrix:
83 86 77
15 93 35
86 92 49
21 62 27

Transpose of Matrix:
83 15 86 21
86 93 92 62
77 35 49 27
```


40. Методы вычисления обратной матрицы

1. (Из лекций)

$A * A^{-1} = E$ и $A^{-1} = \frac{E}{A} = \frac{E}{E-(E-A)} = \frac{E}{E-X}$, $X = E - A$, E – единичная матрица. Данное выражение похоже на сумму бесконечной убывающей геометрической прогрессии: $\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots$

, которая сходится (имеет конечную сумму в пределе) при $|x| < 1$. Это

даёт нам следующую идею: $A^{-1} = \frac{E}{E-X} = E + X + X^2 + X^3 + \dots$ и,

подставляя $X = E - A$, получим:

$A^{-1} = E + (E - A) + (E - A)^2 + (E - A)^3 + \dots$, если $|\det(E - A)| < 1$.

Каждый раз умножая матрицу $(E - A)$ на себя, мы будем уменьшать детерминант результата, таким образом мы можем остановить вычисление суммы и, соответственно обратной матрицы тогда, когда такой детерминант станет достаточно мал. Алгоритм имеет линейную сложность.

2. Метод Жордана-Гаусса (лаба про обратную матрицу из первого сема)

- 1) Исходная матрица должна быть квадратная;
- 2) Вычислим определитель матрицы, если он не ноль, то обратная матрица существует. Сложность $O(n!)$. Этот этап можно пропустить, т.к. существование обратной матрицы можно определить в следующем пункте;
- 3) Создадим единичную матрицу. Приведем исходную матрицу к единичной с помощью элементарных преобразований: умножение строк/столбцов на число, сложение строк/столбцов, перестановка строк/столбцов. Параллельно с этим будем производить абсолютно те же действия с созданной нами единичной матрицей. Если при этом во время прямого хода Гаусса (приведение к ступенчатому виду) получим нулевую строку, то определитель исходной матрицы равен нулю и обратной матрицы не существует. Иначе, полученная из единичной матрицы такими преобразованиями итоговая матрица будет являться обратной к исходной. Сложность $O(n^3)$.

3. Метод присоединенной матрицы (алгебраические дополнения)

Крайне неэффективный алгоритм и использовать его при $n > 3$ не стоит:

для матрицы $N \times N$ для каждого из n^2 элементов нужно найти алгебраическое дополнение - по сути определитель, который считается за факториал, т.е. сложность $n^2 * (n - 1)!$ как минимум

- 1) Исходная матрица должна быть квадратная;
- 2) Вычислим определитель исходной матрицы, если он не ноль, то обратная матрица существует;
- 3) Составляем матрицу из соответствующих алгебраических дополнений исходной матрицы;

- 4) Транспонируем полученную матрицу и делим её на определитель исходной матрицы – получили обратную матрицу.

41. Структура рекурсивных алгоритмов

Рекурсия в программировании – когда функция вызывает сама себя.

Рекурсия делится на прямую и косвенную. В прямой рекурсии функция вызывает сама себя напрямую. В косвенной сначала функция А вызывает функцию Б, которая снова вызывает функцию А.

Рекурсивный алгоритм – алгоритм, использующий рекурсию. Построение рекурсивного алгоритма сводится к разбиению главной задачи на множество малых вариантов этой же задачи.

Простейшим примером рекурсивного алгоритма является алгоритм нахождения наибольшего общего делителя двух чисел (Алгоритм Евклида)

```
int NOD(int a, int b) {  
    if (a != 0 && b != 0) {  
        if (a > b) NOD(a % b, b);  
        else NOD(a, b % a);  
    }  
    else return a + b;  
}
```

При разработке рекурсивного алгоритма важно чётко обозначить точку выхода, чтобы избежать ситуаций, когда рекурсия происходит до бесконечности.

Любой рекурсивный алгоритм можно преобразовать в цикл, и наоборот.

```
int NOD(int a, int b) {  
    while (a != 0, b != 0) {  
        if (a > b) a %= b;  
        else b %= a;  
    }  
    return a + b;  
}
```

(Алгоритм Евклида в виде цикла)

Хорошим примером рекурсивного алгоритма является алгоритм вычисления определителя матрицы «в тупую».

$$\text{Опр}(A) = \sum (-1)^{n+m} \cdot \text{Опр}(\text{минор } A_{mn})$$

Однако временная сложность этого алгоритма оставляет желать лучшего, поэтому рациональным такое использование рекурсии не назовёшь.

Простейший рекурсивный алгоритм содержит:

1. условие выхода;
2. рекурсивный вызов с изменением параметров.

Из 2го пункта также следует, что рекурсивный алгоритм должен иметь начальные параметры. Циклические алгоритмы имеют те же обязательные элементы (условие выхода и изменение параметров).

42. Применение рекурсии

Рекурсия применяется обычно при решении задач, которые легко можно разложить на несколько таких же задач. С помощью рекурсии можно представить, например, формулу вычисления определителя матрицы:

$$\text{Опр}(A) = \sum (-1)^{n+m} \cdot \text{Опр}(\text{минор } A_{mn})$$

```
int size(int** A);
```

```
int** minor(int** A, int m, int n);
```

```
int det(int** A) {  
    if (size(A) == 2) return A[0][0] * A[1][1] - A[0][1] * A[1][0];  
    else {  
        int rez = 0;  
        for (int i = 0; i < size(A); i++) {  
            if (i % 2 == 0) rez += det(minor(A, i, 0));  
            else rez -= det(minor(A, i, 0));  
        }  
    }  
}
```

```

    }

    return rez;

}

}

```

Также рекурсией можно заменить любой циклический алгоритм, в некоторых случаях это может дать прибавку к производительности, но есть нюанс.

Сравнение рекурсии с циклом:

- Если мы преобразовываем рекурсивный алгоритм в циклический, их сложность будет одинакова.
- Зачастую рекурсия выполняется быстрее цикла, но очень легко может стать неэффективной по памяти, так как компьютеру придётся хранить все локальные переменные функции, пока она не завершится. Однако не всегда легко предугадать какая программа будет работать быстрее. Чтобы эффективно работать с рекурсией крайне важно понимать тонкости работы языка и использования памяти.

Рекурсия позволяет записать сложный алгоритм в компактной форме (хотя не всегда читабельной и удобной для понимания)

Пример: расчет факториала числа (хотя это достаточно простой)

```

int factor(int a) {

    if (a == 1 || a == 0) return 1;

    else return a * factor(a - 1);

}

```

Как можно заметить, алгоритм уместился всего в 2 строки

43. Оценка сложности рекурсивных алгоритмов

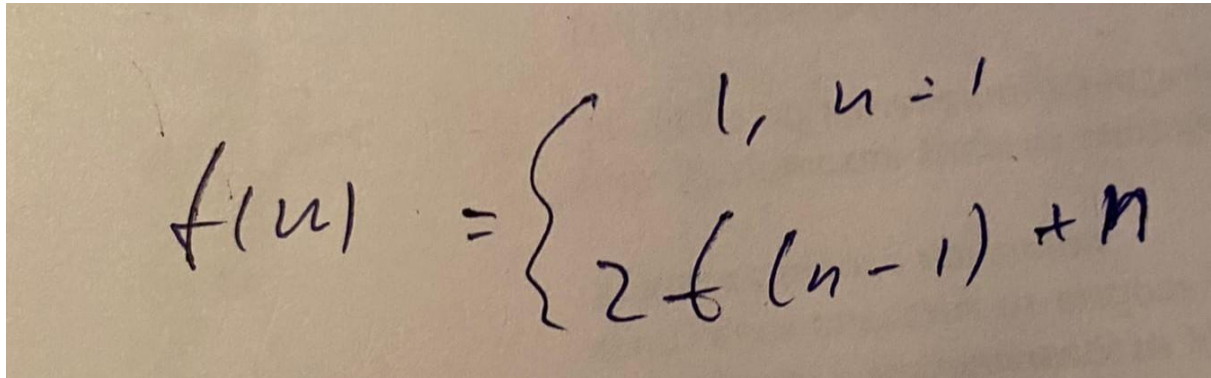
Любой циклический алгоритм можно преобразовать в рекурсивный (смотри выше).

Оценить сложность рекурсивного алгоритма аналитически сложно (но циклического то можно блин, нафига нужно вот это всё что дальше?) но для некоторых случаев это можно сделать математически. Этот метод называется методом рекуррентных соотношений.

Этим методом можно определить только временную сложность алгоритма.

Для этого мы должны определить две вещи: кол-во действий при простейшем варианте, когда оно = const (начальные условия) и формулу, определяющую рекурсивную зависимость.

Предположим, мы получили нечто подобное.



A photograph of a piece of paper with a handwritten recursive formula in black ink. The formula is:
$$f(n) = \begin{cases} 1, & n = 1 \\ 2f(n-1) + n \end{cases}$$

Теперь мы преобразуем это в однородное уравнение, и предположим, что $t(n) = Ax^n$.

Составим харур, получим корни, получим общее решение, из которого используя НУ получим частное. В нашем случае корни харура получились очень плохими, что бывает часто при использовании этого метода. (Он вообще говно так-то. (Так Козлову в билете и напишем))

В случае, если рекурсивная зависимость определяется не как $t(n - 1)$ как в первом случае, а как $t(n/2)$, то при решении нужно будет произвести замену $n = 2^k$, после чего в ответе может вылезти логарифм.

В итоге для использования этого метода обязательно, чтобы изменение n было линейным. ($n - k$ или n/k).

elib.sfu-kras.ru/bitstream/handle/2311/772/%20%20%20%20%20%20%20.pdf?sequence=1

вот еще доп инфа, на всякий случай. там из первых двух абзацев можно много воды взять.

44. Списковые структуры и операции с ними

// ПРО ОБЫЧНЫЙ СПИСОК ПИСАТЬ НА СВОЙ СТРАХ И РИСК

Список - структура данных, которая построена на том, что каждый элемент имеет ссылку на следующий.

Связный список- структура данных, состоящая из узлов. Каждый узел, по аналогии с стандартным списком, содержит ссылку на следующий узел.

Связные списки могут быть односвязными и двусвязными.

В списке с одиночными связями каждый элемент содержит ссылку на следующий элемент данных.

В списке с двойными связями каждый элемент содержит ссылку не только на следующий элемент, но и на предыдущий.

Двусвязные списки являются более распространенными, т.к. в случае повреждения их легче перестраивать в силу наличия двух ссылок, а не одной.

Стоит отметить, что двусвязные списки являются динамическими, а значит они могут удлиняться и укорачиваться в процессе выполнения программы.

Также в случае удаления/добавления нового элемента, ссылки в списке меняются соответствующим образом.

Наиболее ярко особенности списка можно рассмотреть на примере операций добавления и удаления элементов списка.

Наиболее простой динамической структурой является однонаправленный список.

Для создания списка определяется структура T, у которой есть одно поле next, объявленное как указатель на следующий элемент. Последний элемент списка не указывает ни на что (значение next есть NULL)

Над списком, можно выполнять операции поиска элемента, удаления элемента и занесение нового элемента в начало, конец или середину списка. Понятно, что все эти операции будут выполняться путем манипуляций над содержимым поля next уже имеющихся элементов списка. Для оптимизации операций над списком иногда создают вспомогательную переменную-структуру (заголовок списка), состоящую из двух полей - указателей на первый и последний элементы списка. Для этих же целей создают двунаправленные списки, элементы которых, помимо поля next, включают поле previous, содержащее указатель на предыдущий элемент списка.

45. Построение списков при помощи массивов

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения и исключения.

Существует два основных способа хранения списков: последовательное и связанное. При первом способе хранения элементы располагаются в памяти в последовательных ячейках, т.е. друг за другом. При связанном хранении каждый элемент списка содержит связь со следующим, при этом их расположении в памяти может быть произвольным.

Рассмотрим реализацию связанного хранения списка с помощью массивов. При такой реализации элементы списка располагаются в смежных ячейках массива, а вместо указателя на следующий элемент используется индекс следующего элемента. Это позволяет удобно просматривать содержимое списка, а также вставлять новые элементы в конец списка. Но для вставки в середину списка, потребуется переместить все последующие элементы на одну позицию к концу списка. Для удаления элемента в середине или начале также потребуется перемещение элементов на одну позицию к началу списка.

46. Построение списков при помощи указателей

Список – это простая структура данных, которая представляет собой упорядоченный набор элементов произвольного размера. Существует несколько способов реализации такого типа данных. Рассмотрим построение при помощи указателей, которые связывают последовательные элементы между собой. Такой способ позволяет избавиться от использования непрерывной области памяти, поэтому при удалении или вставки нового элемента не нужно перемещать элементы списка.

При данной реализации используется вложение классов. Первый – *List* – налаживает механизм связанного списка, в него вкладывается класс *Node*, который описывает сами элементы списка, а именно: указатель на следующий "узел"(элемент) и данные, которые будут храниться в текущем "узле". Также в классе *List* содержится информация о текущем размере списка и указатель на "головной" элемент списка, если список пустой, **head = nullptr*. Последний элемент указывает на *nullptr*.

Пример вложения классов для списка, в котором хранятся целочисленные элементы:

```
class List {
private:

    class Node {
public:
        Node* pNext;
        int data;
```

```

        Node(int data, Node* pNext = nullptr) {
            this->data = data;
            this->pNext = pNext;
        }
    };

    int Size;
    Node* head;
};

```

Плюсы:

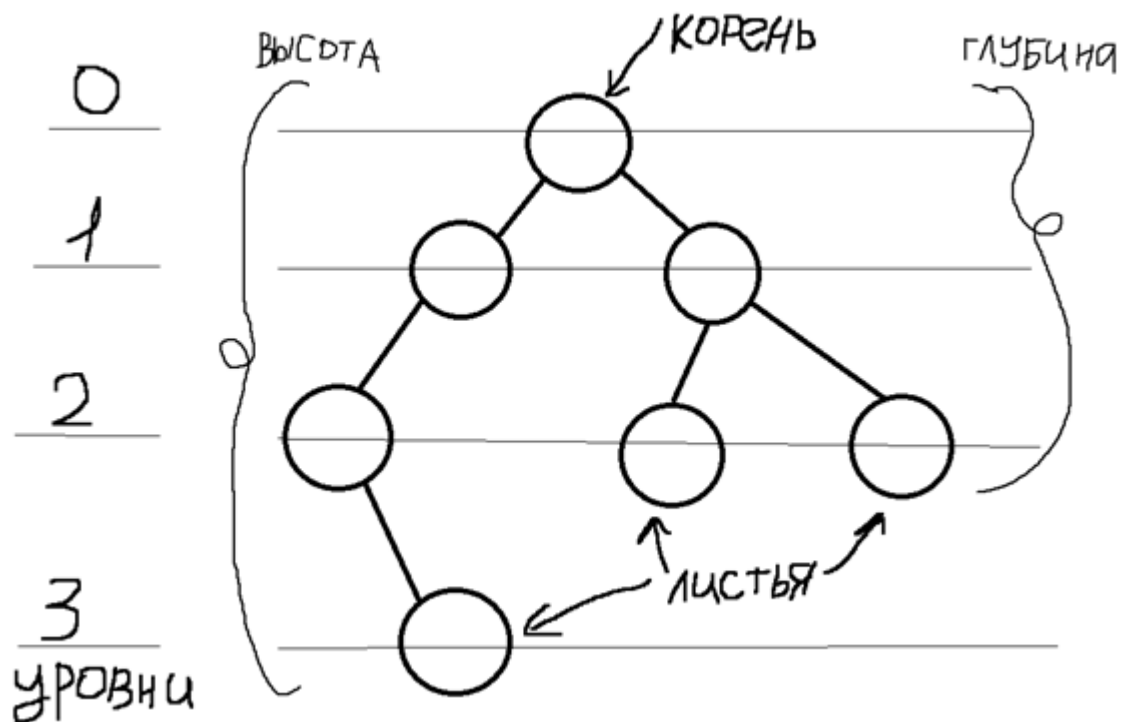
- легко вставлять и удалять элементы в любое место списка;
- размер ограничен техническими возможностями компьютера;

Минусы:

- сложность прямого доступа к элементу;
- трудно производить параллельные векторные операции, например, вычисление суммы из-за постоянного перебора элементов.

47. Древовидные структуры данных, двоичные деревья

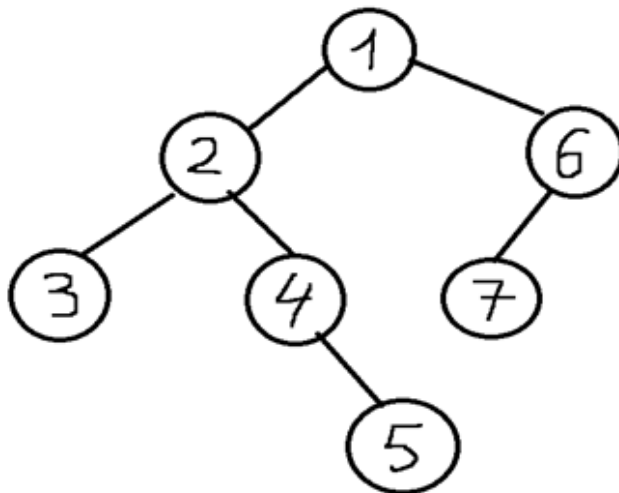
Древовидные структуры данных – нелинейные структуры, которые иерархически упорядочивают данные. Они представляет собой набор объектов (список или динамический массив указателей), называемых узлами, которые соединены между собой ветвями. Каждый узел содержит значение или данные, и он может иметь или не иметь дочерний узел. Первый (верхний) узел называется корнем. Последние узлы, не имеющие потомков, называются листьями (или терминальными узлами). Также оно разделено на уровни, корень находится на нулевом уровне, его потомки на первом и т.д. Существуют ещё такие понятия, как высота дерева (длина самого дальнего пути от корня к листу или номер самого нижнего уровня) и глубина дерева (длина самого короткого пути от листа к корню или номер уровня самого высокого листа).



Двоичное (бинарное) дерево – иерархическая структура данных, в которой каждый узел имеет не более двух потомков, называемыми левыми и правыми наследниками и также являющимися двоичными деревьями. Каждый узел такого дерева как правило имеет три атрибута: <значение>, <левый наследник> и <правый наследник>, но также он может быть пустым.

Древовидные структуры данных используются, например, в файловых системах. Они позволяют осуществлять быстрый поиск, вставку и удаление элементов (если оно отсортировано и сбалансировано). Сбалансированным дерево называется, если число узлов в левом и правом поддеревьях отличается не более, чем на 1.

48. Алгоритмы обхода вершин двоичных деревьев



T-Tree – корень
LST-Left Side Tree
RST-Right Side Tree

1) Прямой обход («сверху вниз»: обход начиная с корня,
корень -> левое поддерево -> правое поддерево)

```
Pre_order(Tree) {  
  //обработка корня T  
  Pre_order(T(LST));  
  Pre_order(T(RST));  
}
```

Результат: 1 2 3 4 5 6 7

2) Поперечный обход («слева направо»: начиная с самого левого листа,
левое поддерево -> корень -> правое поддерево)

```
In_order(Tree) {  
  In_order(T(LST));  
  //обработка корня T  
  In_order(T(RST));  
}
```

Результат: 3 2 4 5 1 7 6

3) Обратный обход («снизу вверх»: начиная с самого левого листа,
левое поддереву -> правое поддереву -> корень)

```
Post_order(Tree) {  
    Post_order(T(LST));  
    Post_order(T(RST));  
    //обработка корня T  
}
```

Результат: 3 5 4 2 7 6 1

49. Двоичные деревья поиска

Двоичное дерево поиска – это древовидная структура данных при условиях:

- У каждого узла не более двух наследников;
- Любое значение меньше или равное значению узла становится левым наследником или наследником левого наследника;
- Любое значение больше значения узла становится правым наследником или наследником правого наследника.

Для поиска по значению используется индивидуальный сравнимый ключ.

Поиск по двоичному дереву, алгоритм:

```
Search( T, k ) {  
    Data(T); //возвращает значение поля корня  
    If (is_empty()) { неуспех }  
    Else if ( k==Data(T) ) { return T(root); }  
    Else if ( k<Data(T) ) { Search(LST, k); }  
    Else if ( k>Data(T) ) { Search(RST, k); }  
}
```

Высота сбалансированного двоичного дерева с n вершинами составляет не более, чем $\log_2 n$ (логарифм двоичный), поэтому при применении таких деревьев в качестве деревьев поиска требуется не более $\log_2 n$ сравнений.

50. Вставка и удаление вершин из двоичных деревьев поиска.

Добавление узла не представляет особой сложности. Оно становится еще проще, если решать эту задачу рекурсивно. Если дерево пустое, мы просто создаем новый узел и добавляем его в дерево. Во втором случае мы сравниваем переданное значение со значением в узле, начиная от корня. Если добавляемое значение меньше значения рассматриваемого узла, повторяем ту же процедуру для левого поддерева. В противном случае — для правого.

```
Insert(T, k){  
  
  If ( is_empty() ) { вставляем k в корень T }  
  
  Else if ( k < Data(T) ) { insert(LST, k); }  
  
  Else if ( k > Data(T) ) { insert(RST, k); }  
  
}
```

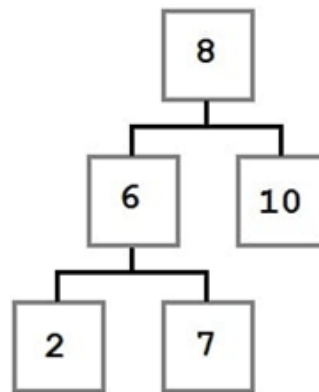
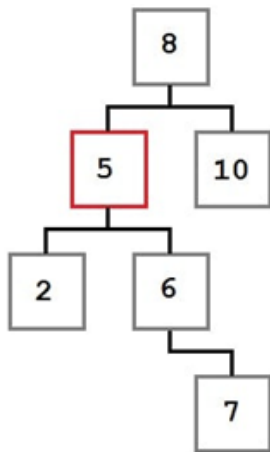
Удаление узла из дерева — одна из тех операций, которые кажутся простыми, но на самом деле таят в себе немало подводных камней.

В целом, алгоритм удаления элемента выглядит так: Найти узел, который надо удалить и удалить его. После того, как мы нашли узел, который необходимо удалить, у нас возможны три случая.

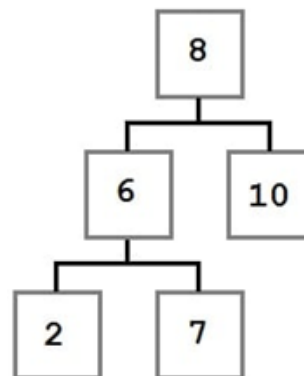
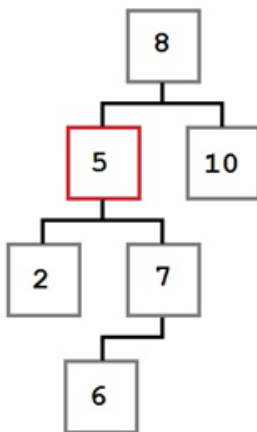
Случай 1: У удаляемого узла нет правого ребенка. В этом случае мы просто перемещаем левого ребенка (при его наличии) на место удаляемого узла.



Случай 2: У удаляемого узла есть только правый ребенок, у которого, в свою очередь нет левого ребенка. В этом случае нам надо переместить правого ребенка удаляемого узла на его место.



Случай 3: У удаляемого узла есть первый ребенок, у которого есть левый ребенок. В этом случае место удаляемого узла занимает крайний левый ребенок правого ребенка удаляемого узла.



51. Структура стандартной библиотеки шаблонов

Механизм шаблонов встроен в компилятор C++, естественно, существуют и стандартные библиотеки, реализующие этот механизм. STL является самой эффективной библиотекой C++ на сегодняшний день. Библиотека стандартных шаблонов (Standard template Library - STL) представляет шаблонные классы и функции общего назначения, которые реализуют многие популярные и часто используемые алгоритмы и структуры данных. Например, она включает в себя поддержку векторов, списков, очередей и стеков, а также определяет различные функции, обеспечивающие к ним доступ. Поскольку STL состоит из шаблонных классов, алгоритмы и структуры данных могут быть применены к данным практически любого типа.

Для использования коллекции в своем коде используйте следующую директиву:

`#include <T>`, где T — название коллекции

Ядро STL включает три основных элемента: контейнеры, алгоритмы, итераторы и функциональные объекты, все они работают совместно друг с другом.

Контейнеры - это объекты, которые содержат другие объекты.

Итераторы - указатели, позволяющие алгоритмам перемещаться по данным контейнера, точнее итератор - это некий обобщенный указатель, который позволяет алгоритму не задумываться о типе передаваемых данных.

Алгоритмы – обрабатывают содержимое контейнеров.

В библиотеке STL существует группа функций, выполняющих некоторые стандартные действия, например, определение размера, поиск, преобразование, сортировку, копирование и т. д. Они называются алгоритмами. Параметрами для алгоритмов, как правило, служат итераторы.

Контейнеры

Библиотека STL имеет в своем арсенале элементы, называемые контейнерами. Контейнеры - это объекты, хранящие в себе другие объекты. В STL таких контейнеров десять:

- `vector` - массив с произвольным доступом, который изменяет свой размер по мере необходимости;
- `list` – двунаправленный связный список, похож на вектор, эффективен при добавлении и удалении данных в любое место цепочки;
- `map` – сохраняет пары вида `<ключ, значение>`, при этом однозначная (каждому ключу соответствует единственное значение), где ключ — некоторая характеризующая значение величина, для которой применима операция сравнения
- `multimap` - то же, что и `map`, но допускающий повторяющиеся ключи; для использования в коде используется `#include <map>`
- `set` - это отсортированная коллекция одних только ключей, т.е. значений, для которых применима операция сравнения, при этом уникальных
- `multiset` - то же, что и `set`, но допускающий повторяющиеся ключи. для подключения: `#include <set>`
- `queue` - данные добавляются и вынимаются в том же порядке;
- `deque` - контейнер, удобный для вставки данных в начало или конец;

- stack - данные добавляются в одном порядке, а вынимаются в обратном;
- priority queue - то же, что и queue, но может сортировать данные по приоритету.

Для представления строк в STL есть классы:

`string` — коллекция однобайтных символов в формате ASCII;

`wstring` — коллекция двухбайтных символов в формате Unicode; включается командой `#include <xstring>`

Строковый поток: `<stringstream>`

52. Применение стандартной библиотеки шаблонов

Библиотека стандартных шаблонов (STL) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Шаблоны функций — это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов. По описанию шаблон функции похож на обычную функцию: разница в том, что некоторые элементы не определены (типы, константы) и при использовании шаблона указываются в угловых скобках.

Шаблоны классов — обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа. Представляют собой конструкции, по которым могут быть сгенерированы действительные классы путём подстановки вместо параметров конкретных типов.

Итак, следующая программа иллюстрирует использование класса vector.

```
// Короткая программа демонстрации
// работы класса vector,
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector v; // создаем вектор нулевой длины
    int i;

    // Отображаем исходный размер объекта v.
    cout << "размер = " << v.size() << endl;

    /* Помещаем значения в конец вектора:
    вектор будет расти по мере необходимости.*/
    for(i=0; i<10; i++) v.push_back(i);

    // Отображаем текущий размер объекта v.
    cout << "размер сейчас = " << v.size() << endl;

    // Доступ к содержимому вектора можно получить, используя индекс.
    for(i=0; i<10; i++) cout << v[i] << " ";
    cout << endl;

    // Доступ к первому и последнему элементам вектора.
    cout << "первый = " << v.front() << endl;
    cout << "последний = " << v.back() << endl;

    // Доступ через итератор.
    vector::iterator p = v.begin();
    while (p != v.end()) {
        cout << *p << " ";
        p++;
    }

    return 0;
}
```

Ниже показан результат работы этой программы.

```
размер = 0
размер сейчас =10
0123456789
первый = 0
последний = 9
0123456789
```