

Легенда билетов:

Deprecated

Nota bene

Легенда таблицы:

Зеленый квадратик - билет написан, можно учить

Красный квадратик - к билету есть вопросы

Синий квадратик - билет только что был написан, нужно, чтобы посмотрели другие

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48		50

51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75

76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

101	102	103	104

ООП

1. Аргументы, передаваемые функции по умолчанию.

Это аргументы, которые можно не указывать при вызове функции, потому что они добавляются компилятором автоматически (пример, $b = 0$). Эти аргументы объявляются в прототипе функции, причем после аргументов, у которых нет значений по умолчанию (иначе ошибка компиляции).

```
int fun(int a, int b = 0) {  
    return a * b;  
}
```

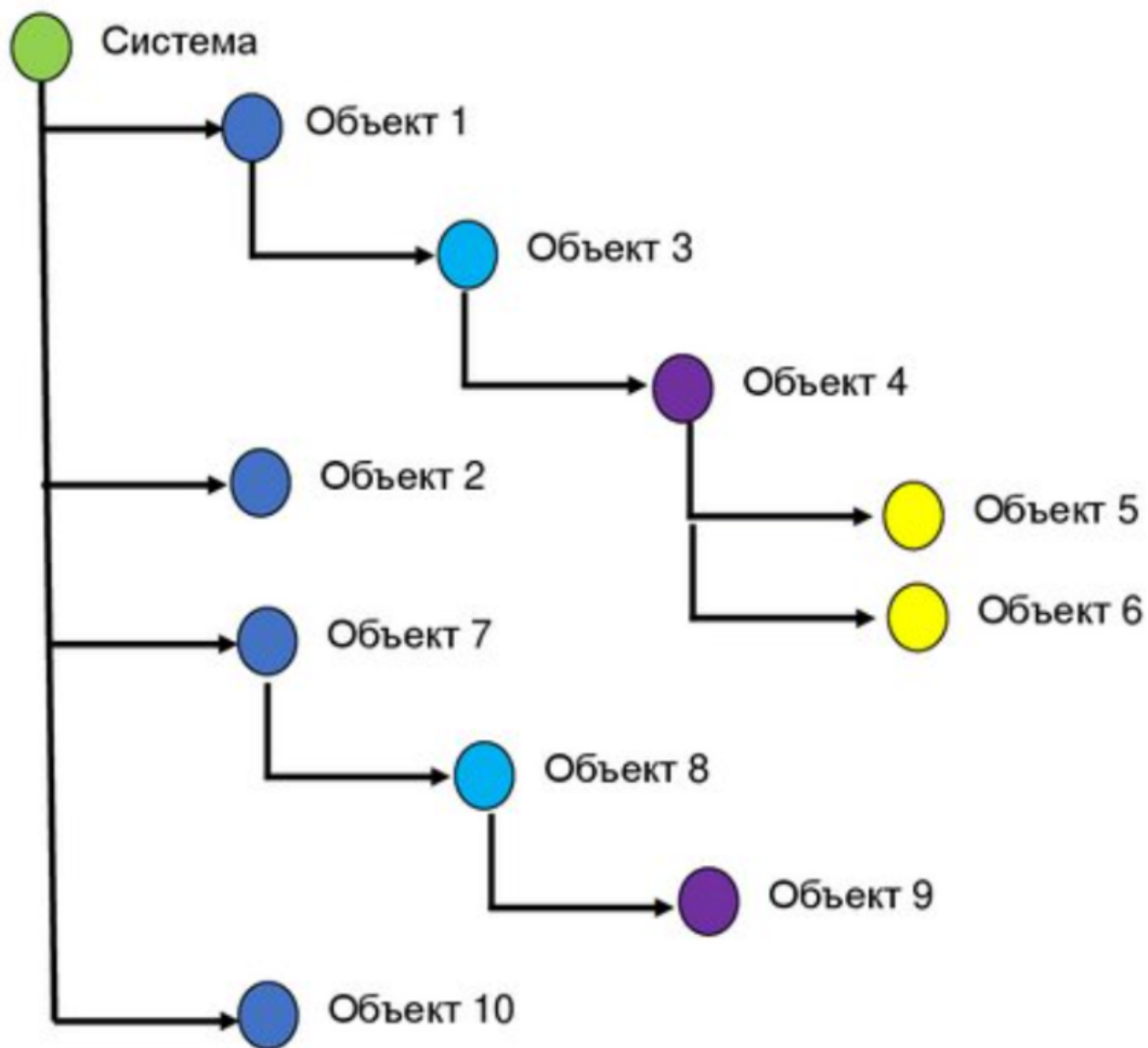
2. Архитектура системы. Иерархия объектов.

Архитектура – это набор значимых решений по поводу организации **системы** программного обеспечения, набор структурных элементов и их интерфейсов, при помощи которых конструируется **система**.

Архитектура системы так или иначе строится путем создания иерархии объектов и взаимодействия между объектами.

Разбиение системы на объекты является **объектной декомпозицией**.

Такая декомпозиция существенно упрощает конструирование сложных систем. Разработка интерфейсов взаимодействия происходит между объектами системы и внешней средой. Способ организации интерфейса зависит от особенности объекта и его назначения. Схематически это представлено на следующем.



3. Ввод-вывод в C++. Потоки.

1) В C++ нет встроенных средств ввода-вывода, поэтому поточный ввод-вывод выполняется с помощью функций сторонних библиотек (**stdio.h** и **iostream**).

2) Библиотекой **iostream** определено два типа: **istream** (поток ввода), **ostream** (поток вывода)

3) **Поток** - последовательность символов, которая записывается на устройство ввода или считывается с него (например консоль)

4) Для записи или вывода символов на консоль применяется объект **cout**, для чтения - объект **cin**. Есть еще **cerr** - объект потока вывода сообщений об ошибках

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

>> - получить из входного потока

<< - поместить в выходной поток

6) Ввод информации: из входного потока читается последовательность символов до пробела, затем эта последовательность преобразуется к типу идентификатора, и получаемое значение помещается в **идентификатор**

7) Вывод информации: определяется тип данных переменной, подлежащей выводу, и выбирается соответствующий оператор вставки потока для отображения значения. Оператор << перегружен для вывода элементов данных встроенных типов и значений указателя

8) Работа с файлами: функционал для работы с ними предоставляет заголовочный файл **fstream**. Там определены два класса: `ifstream` и `ofstream`, реализующие возможности чтения и записи информации.

4. В чем различие между ссылкой и указателем?

Указатель – переменная, значением которой является адрес ячейки памяти.

Ссылки – особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается.

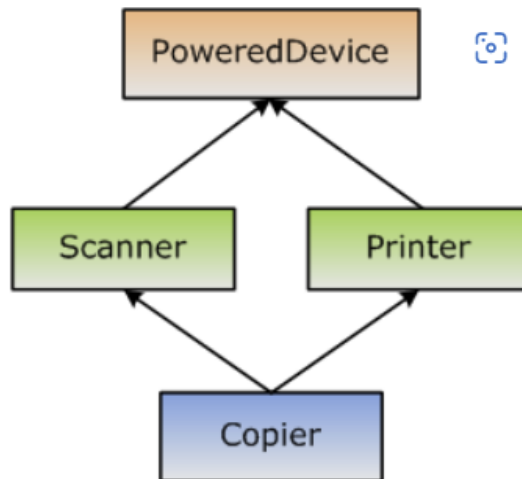
Различия:

- Указатель хранит адрес объекта, ссылка ссылается на него напрямую (скрыто хранит указатель)
- ссылка, в отличие от указателя, не может быть неинициализированной;
- ссылка не может быть изменена после инициализации.
- У ссылки нет адреса (это просто другое имя для того же объекта)
- Существует арифметика указателей, но нет арифметики ссылок.
- Ссылка не обладает квалификатором `const`

5. Виртуальные базовые классы.

Виртуальный базовый класс – решение так называемой проблемы ромба, когда существует две копии базового класса – по одной для каждого наследника. Иногда желательно, чтобы частью обоих классов наследников была один и тот же экземпляр базового класса.

Чтобы использовать базовый класс совместно, просто вставьте ключевое слово `virtual` в список наследования производного класса. Это создает так называемый **виртуальный базовый класс**, что означает, что существует только один базовый объект. Этот базовый объект используется всеми объектами в дереве наследования и создается только один раз.



Нюансы:

```
1  #include <iostream>
2
3  class PoweredDevice
4  {
5  public:
6      PoweredDevice(int power)
7      {
8          std::cout << "PoweredDevice: " << power << '\n';
9      }
10 };
11
12 // обратите внимание: PoweredDevice теперь является виртуальным базовым классом
13 class Scanner: virtual public PoweredDevice
14 {
15 public:
16     Scanner(int scanner, int power)
17         // эта строка требуется для создания объектов Scanner,
18         // но в данном случае игнорируется
19         : PoweredDevice{ power }
20     {
21         std::cout << "Scanner: " << scanner << '\n';
22     }
23 };
24
25 // обратите внимание: PoweredDevice теперь является виртуальным базовым классом
26 class Printer: virtual public PoweredDevice
27 {
28 public:
29     Printer(int printer, int power)
30         // эта строка требуется для создания объектов Printer,
31         // но в данном случае игнорируется
32         : PoweredDevice{ power }
33     {
34         std::cout << "Printer: " << printer << '\n';
35     }
36 };
37
38 class Copier: public Scanner, public Printer
39 {
40 public:
41     Copier(int scanner, int printer, int power)
42         : PoweredDevice{ power }, // PoweredDevice создается здесь
43         Scanner{ scanner, power }, Printer{ printer, power }
44     {
45     }
46 };
```

Во-первых, виртуальные базовые классы всегда создаются перед неvirtуальными базовыми классами, что гарантирует создание всех базовых классов до их производных классов.

Во-вторых, обратите внимание, что у конструкторов `Scanner` и `Printer` всё еще есть вызовы конструктора `PoweredDevice`. При создании экземпляра `Copier` эти вызовы конструктора просто игнорируются, потому что за создание `PoweredDevice` отвечает `Copier`, а не `Scanner` или `Printer`. Однако если бы мы создавали экземпляр `Scanner` или `Printer`, эти вызовы конструктора были бы использованы, и были бы применены обычные правила наследования.

В-третьих, если класс наследует один или несколько классов, имеющих виртуальных родителей, за создание виртуального базового класса отвечает наиболее производный класс. В этом случае `Copier` наследует `Printer` и `Scanner`, оба из которых имеют виртуальный базовый класс `PoweredDevice`. `Copier`, самый производный класс, отвечает за создание `PoweredDevice`. Обратите внимание, что это верно даже в случае одиночного наследования: если `Copier` унаследован только от `Printer`, а `Printer` был унаследован от `PoweredDevice`, `Copier` по-прежнему отвечает за создание `PoweredDevice`.

В-четвертых, все классы, наследующие виртуальный базовый класс, будут иметь виртуальную таблицу, даже если в противном случае у них ее обычно не было бы, и, таким образом, размер экземпляров этих классов будет больше на размер указателя.

Поскольку `Scanner` и `Printer` фактически являются производными от `PoweredDevice`, `Copier` будет только с одним подобъектом `PoweredDevice`. И `Scanner`, и `Printer` должны знать, как найти этот единственный подобъект `PoweredDevice`, чтобы иметь доступ к его членам (потому что, в конце концов, они являются производными от него). Обычно это делается с помощью магии виртуальной таблицы (которая, по сути, сохраняет смещение от каждого подкласса к подобъекту `PoweredDevice`).

(Встретив ключевое слово `virtual`, компилятор помечает, что для этого метода должно использоваться позднее связывание: для начала он создает для класса таблицу виртуальных функций, а в класс добавляет новый скрытый для программиста член — указатель на эту таблицу)

6. Взаимодействие объектов. Три примера взаимодействия объектов.

Сигналы и обработчики, вызов внутри метода одного объекта методов другого, друж классы.

Возможны следующие отношения объектов:

- **Наследование** (класс наследник имеет все поля и методы родительского класса и, как правило, добавляет новый функционал и поля). Наследование описывается словом **"является"**
- **Композиция** — включаемый объект создается в конструкторе класса-владельца (двигатель создается при создании автомобиля и полностью им управляется)
- **Агрегация** — включаемый объект создается где-то еще и потом передается в качестве параметра в конструктор класса-владельца

Композиция и агрегация — частный случай **ассоциации**, то есть включения одного класса в другой в качестве одного из полей. **Ассоциация** описывается словом **"имеет"**

Псевдокод агрегации

```
class Ob1
public:
    Ob1(Ob2 data) : ob2(data) {}
private:
```

ob2 ob2;

7. Виртуальные методы. Наследование виртуальных методов.

Виртуальная функция – это особый тип функций, которая при вызове преобразуется в версию функции, которая принадлежит самому дочернему классу из имеющихся функций в базовом и производном классах.

Чтобы сделать функцию виртуальной, просто поместите перед объявлением функции ключевое слово `virtual`

```
class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    virtual std::string_view getName() const { return "B"; }
};

class C: public B
{
public:
    virtual std::string_view getName() const { return "C"; }
};

class D: public C
{
public:
    virtual std::string_view getName() const { return "D"; }
};

int main()
{
    C c;
    A& rBase{ c };
    std::cout << "rBase is a " << rBase.getName() << '\n';

    return 0;
}
```

c – объект

rBase – ссылка на объект класса A, хранящийся в c

`rBase.getName` это `A::getName()`, однако `A::getName` – виртуальная функция, поэтому компилятор вызовет наиболее дочернюю совпадающую функцию. Ей будет `C::getName`

Если функция помечена как виртуальная, все совпадающие переопределения также считаются виртуальными, даже если они явно не отмечены как таковые.

Виртуальные функции не должны вызываться из конструкторов или деструкторов потому что часть производного класса к этому моменту еще не будет создана или будет уничтожена.

Кстати, разрешение вызова виртуальной функции занимает больше времени, чем разрешение вызова обычной функции.

8. В чем отличие между классом и структурой?

Члены класса, определенного с помощью ключевого слова `class`, по умолчанию являются `private`. Члены класса, определенного с помощью ключевого слова `struct` или `union`, по умолчанию являются `public`.

При отсутствии спецификатора доступа (т.е. `private/protected/public`) у базового класса, базовый класс будет `public` если класс определен с помощью `struct` и `private` если класс определен с помощью `class`.

9. Встраиваемая функция.

Основная идея в том, чтобы ускорить программу ценой занимаемого места. После того как вы определите встроенную функцию с помощью ключевого слова `inline`, всякий раз когда вы будете вызывать эту функцию, компилятор будет заменять вызов функции фактическим кодом из функции.

Встроенные функции очень хороши для ускорения программы, но если вы используете их слишком часто или с большими функциями, у вас будет чрезвычайно большая программа. Иногда большие программы менее эффективны, и поэтому они будут работать медленнее, чем раньше. Встроенные функции лучше всего подходят для небольших функций, которые часто вызываются.

```

1  #include <iostream>
2
3  using namespace std;
4
5  inline void hello()
6  {
7      cout<<"hello";
8  }
9  int main()
10 {
11     hello(); //Call it like a normal function...
12     cin.get();
13 }

```

10. Для чего используется ключевое слово `protected`?

Protected — определяет элемент как закрытый для доступа, обратиться к данной переменной или метода можно только в рамках текущего класса или дружественной функции, а также в классе наследнике.

`Protected` поля и методы наследуются.

Также `protected` используется как тип наследования. При защищенном наследовании открытые и защищенные члены становятся защищенными, а закрытые члены остаются недоступными.

Спецификатор доступа в базовом классе	Спецификатор доступа при защищенном наследовании
<code>public</code>	<code>protected</code>
<code>protected</code>	<code>protected</code>
<code>private</code>	не доступен

11. Дружественная функция.

Дружественные функции — это функции, которые не являются членами класса, однако имеют доступ к его закрытым членам — переменным и функциям, которые имеют спецификатор `private`. Для определения дружественных функций используется ключевое слово **friend**.

```

friend void drive(Auto &);
friend void setPrice(Auto &, int price);
//...

```

Дружественные функции не являются членами класса и не наследуются

12. Дружественный класс.

Дружественный класс — класс, имеющий доступ к закрытым полям данного класса.

```

class Storage
{
private:
    int m_nValue;
    double m_dValue;
public:
    Storage(int nValue, double dValue)
    {
        m_nValue = nValue;
        m_dValue = dValue;
    }

    // Сделаем класс Display другом Storage
    friend class Display;
};

```

Кстати, то, что Display - друг Storage, не делает Storage другом Display

13. Защищенные члены класса.

Спецификатор доступа **protected** позволяет получить доступ к членам класса следующим товарищам: классу, к которому принадлежит член, друзьям и производным от данного классам.


14. Жизненный цикл объекта.

1. **План (описание)** - любой объект должен до его появления быть описан (до появления объекта требуется его описание). **План** - описание объекта, также содержащий четкие требования относительно ресурсов для создания объекта (тут путур конкретно уже хуйню несет. я бы не стал это писать)
2. **Создание (конструирование)** - исходя из описания объекта бла-бла-бла какую же он хуйню несет
3. **Старт объекта** - появление объекта (старт объекта в физическом, материализованном смысле).
4. **Функционирование** - начало работы объекта
5. **Остановка** - окончание ресурса объекта
6. **Демонтаж**



15. Жизненный цикл виртуального объекта и его реализация на языке C++.

Язык обеспечивает жизненный цикл виртуального объекта по аналогии жизненного цикла объекта из предметной области. Схемы этих жизненных циклов совпадают.



Реализация жизненного цикла
виртуального объекта на языке C++

Описание	Класс
Создание	Отработка конструктора объекта. Выделение памяти и исходная инициализация.
Объект	Завершение работы конструктора объекта.
Старт	Начало функционирования.
Функционирование	Участие объекта в работе (в алгоритме) приложения
Остановка	Начало отработки деструктора объекта.
Демонтаж	Отработка деструктора объекта.
Завершение	Завершение работы деструктора объекта. Освобождение выделенной памяти.

Центр дистанционного обучения
Образование в стиле hi tech

МИРЭА, Институт Информационных технологий, кафедра Вычислительной техники

online.mirea.ru 8

16. Инкапсуляция.

Все программы состоят из двух основных элементов: **инструкций (кода) и данных**.

Код – это часть программы, которая выполняет действия, а **данные** представляют собой информацию, на которую направлены эти действия.

Инкапсуляция – это такой механизм программирования, который связывает воедино код и данные, которые он обрабатывает, чтобы обезопасить их как от внешнего вмешательства, так и от неправильного использования.

На базе одного класса могут быть созданы множество объектов. Объект созданный согласно описанию класса поддерживает инкапсуляцию. У каждого объекта есть свои границы («рамки»). Эта граница определяется согласно описанию пространства класса. Внутри объекта код, данные или обе эти составляющие могут быть закрытыми в «рамках» этого объекта или открытыми.

Закрытый код (или данные) известен и доступен только другим частям того же объекта. К закрытому коду или данным не может получить доступ та часть программы, которая существует вне этого объекта.

Открытый код (или данные) доступны любым другим частям программы, даже если они определены в других объектах. Обычно открытые части объекта используются для предоставления управляемого интерфейса с закрытыми элементами объекта.

17. Имеются два способа сделать функцию встраиваемой. Что это за способы?

Встраиваемая функция – это небольшая (по объему кода) функция, код которой подставляется вместо ее вызова.

Причиной существования встраиваемой функции является эффективность (скорость выполнения). Вместо инструкции реализации вызова, подставляется код реализации встраиваемой функции. Если код большой, то транслятор организует вызов стандартным образом.

При описании класса встраиваемую функцию можно определить двумя способами.

1. Первый способ определение кода реализации в заголовочной части класса.
2. Второй способ относится к части реализации. Синтаксис представления

**`inline «тип метода» «имя класса» :: «имя метода»
([«параметры»]) { // код реализации }`**

Пример

```
#include <iostream>
using namespace std;
class cl_1 {
    int i;
    int j;
public:
    int get_i ( ) { return i; }
    int get_j ( );
```

```
};  
inline cl_1 :: get_j ( )  
{  
    return j;  
}
```

18. Исключительные ситуации.

Исключительная ситуация – событие при выполнении программы, которое приводит к ее ненормальному или неправильному поведению. Выделяют два вида исключительных ситуаций:

- **Аппаратные**, которые генерируются процессором:
 - Деление на 0
 - Выход за границы массива
 - Обращение к невыделенной памяти
 - Переполнение разрядной сетки
- **Программные** – генерируемые операционной системой и прикладными программами. Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или **возбудить**, исключение (вызываются самим программистом с помощью `throw`)

Идея **обработки исключительных ситуаций** состоит в том, что функция, обнаружившая проблему, но не знающая как её решить, генерирует исключение в надежде, что вызвавшая её функция сможет решить возникшую проблему. Функция, которая может решать проблемы данного типа, указывает, что она перехватывает такие исключения.

Для реализации обработки исключений в C++ используйте выражения `try`, `throw` и `catch`.

Блок `try {...}` позволяет включить один или несколько операторов, которые могут создавать исключение.

Выражение `throw` используется только в программных исключениях и означает, что исключительное условие произошло в блоке `try`.

В качестве операнда выражения `throw` можно использовать объект любого типа.

Для обработки исключений, которые могут быть созданы, необходимо реализовать один или несколько блоков `catch` сразу после блока `try`. Каждый блок `catch` указывает тип исключения, которое он может обрабатывать.

Сразу за блоком `try` находится **защищенный раздел кода**. Выражение `throw` вызывает исключение, т.е. создает его.

Блок кода после `catch` является **обработчиком исключения**. Он перехватывает исключение, вызываемое, если типы в выражениях `throw` и `catch` совместимы.

19. Класс. Назначение и синтаксис описания.

Классы в C++ — это абстракция описывающая методы, свойства, ещё не существующих объектов. (Объекты — конкретное представление абстракции, имеющее свои свойства и методы.)

Классы в C++ предназначены для описания объектов. Их применение позволяет группировать для каждого объекта данные и методы, которые оперируют этими данными, что делает код более читаемым.

```
3 class base {  
4     public:  
5         base(){}  
6     private:  
7         int number;  
8 };
```

20. Класс `vector`.

Введем несколько доп определений

Контейнер — объект, предназначенный для хранения других объектов

Массив — пронумерованное множество элементов одного типа

Вектор — также динамический массив — контейнер для хранения элементов одного типа, способный расширяться по мере использования программы (=динамическая структура)

Из того, что это динамический массив, следует следующее: все элементы вектора расположены в памяти подряд (можно пользоваться арифметикой указателей).

Добавление нового элемента реализовано следующим образом: Изначально создается динамический массив некоторой размерности. Когда добавляются новые элементы, происходит проверка, помещаются ли они в текущий массив. Если да, то они записываются в массив. Если нет, то создается новый массив, размеров в 2 раза больше, все старые элементы копируются в начало в том же порядке, затем добавляются новые элементы.

Из-за того, что при увеличении размера вектора массив может переиздаваться, нужно работать с арифметикой указателей максимально аккуратно.

Синтаксис объявления:

vector <тип данных> <имя объекта>

Пример:

```
vector <string> vect
```

Обходить элементы вектора можно как при помощи доступа по индексу, так и при помощи итератора

Итератор – объект, предоставляющий доступ к элементам определенного контейнера

Синтаксис итератора:

vector<тип данных>::iterator<имя итератора>

Пример:

```
vector<string>::iterator vect
```

Примеры методов данного класса:

Метод	Описание
at (i)	Возвращает ссылку на элемент, заданный параметром i
back ()	Возвращает ссылку на последний элемент вектора
begin ()	Возвращает итератор первого элемент вектора
capacity ()	Возвращает текущую емкость вектора
clear ()	Удаляет все элементы вектора
empty ()	Возвращает истинное значение если вектор пуст, иначе ложь
end ()	Возвращает итератор для конца вектора
erase (iterator it)	Удаляет элемент, на который указывает итератор it. Возвращает итератор элемента, который расположен следующим за удаленным
erase (iterator start, iterator end)	Удаляет элементы, заданные между итераторами start и end. Возвращает итератор элемента, который расположен следующим за последним удаленным
front ()	Возвращает ссылку на первый элемент вектора
insert (iterator it, const T & val = T ())	Вставляет параметр val перед элементом, заданным итератором it. Возвращает итератор элемента
pop_back ()	Удаляет последний элемент вектора
push_back (const T & val)	Добавляет в конец вектора элемент, значение которого равно параметру val

21. Класс string.

String – класс с методами и переменными для организации работы со строками в языке программирования C++. Он включён в стандартную библиотеку C++

Строка – это последовательная коллекция символов, которая используется для представления текста.

Объект **String** — это последовательная коллекция char объектов, представляющих строку

Максимальный String размер объекта в памяти составляет 2 ГБ или около 1 миллиарда символов.

Из всего арсенала строк мы юзаем...erase? и все... но я накидаю вам только конструкторов штук 5, чтобы было

Конструкторы строк

Строки можно создавать с использованием следующих конструкторов:

string() - конструктор по умолчанию (без параметров) создает пустую строку.

string(string & S) - копия строки S

string(size_t n, char c) - повторение символа c заданное число n раз.

string(size_t c) - строка из одного символа c.

string(string & S, size_t start, size_t len) - строка, содержащая не более, чем len символов данной строки S, начиная с символа номер start.

size

Метод size() возвращает длину строки. Возвращаемое значение является беззнаковым типом (как и во всех случаях, когда функция возвращает значение, равное длине строке или индексу элемента - эти значения беззнаковые). Поэтому нужно аккуратно выполнять операцию вычитания из значения, которое возвращает size(). Например, ошибочным будет запись цикла, перебирающего все символы строки, кроме последнего, в виде for (int i = 0; i < S.size() - 1; ++i).

Кроме того, у строк есть метод length(), который также возвращает длину строки.

Подробнее о методе [size](#).

resize

S.resize(n) - Изменяет длину строки, новая длина строки становится равна n. При этом строка может как уменьшиться, так и увеличиться. Если вызвать в виде S.resize(n, c), где c - символ, то при увеличении длины строки добавляемые символы будут равны c.

Подробнее о методе [resize](#).

clear

S.clear() - очищает строчку, строка становится пустой.

Подробнее о методе [clear](#).

empty

S.empty() - возвращает true, если строка пуста, false - если не пуста.

Подробнее о методе [empty](#).

push_back

S.push_back(c) - добавляет в конец строки символ c, вызывается с одним параметром типа char.

Подробнее о методе [push_back](#).

append

Добавляет в конец строки несколько символов, другую строку или фрагмент другой строки. Имеет много способов вызова.

S.append(n, c) - добавляет в конец строки n одинаковых символов, равных c. n имеет целочисленный тип, c - char.

S.append(T) - добавляет в конец строки S содержимое строки T. T может быть объектом класса string или C-строкой.

S.append(T, pos, count) - добавляет в конец строки S символы строки T начиная с символа с индексом pos количеством count.

Подробнее о методе [append](#).

erase

S.erase(pos) - удаляет из строки S с символа с индексом pos и до конца строки.

S.erase(pos, count) - удаляет из строки S с символа с индексом pos количеством count или до конца строки, если pos + count > S.size().

Подробнее о методе [erase](#).

insert

Вставляет в середину строки несколько символов, другую строку или фрагмент другой строки. Способы вызова аналогичны способам вызова метода append, только первым параметром является значение i - позиция, в которую вставляются символы. Первый вставленный символ будет иметь индекс i, а все символы, которые ранее имели индекс i и более сдвигаются вправо.

S.insert(i, n, c) - вставить n одинаковых символов, равных c. n имеет целочисленный тип, c - char.

S.insert(i, T) - вставить содержимое строки T. T может быть объектом класса string или C-строкой.

S.insert(i, T, pos, count) - вставить символы строки T начиная с символа с индексом pos количеством count.

Подробнее о методе [insert](#).

substr

S.substr(pos) - возвращает подстроку данной строки начиная с символа с индексом pos и до конца строки.

S.substr(pos, count) - возвращает подстроку данной строки начиная с символа с индексом pos количеством count или до конца строки, если pos + count > S.size().

22. Какая инструкция catch перехватывает все типы исключительных ситуаций?

Если оператор `catch` задает многоточие (...) вместо типа, блок `catch` обрабатывает все типы исключений.

Поскольку блоки `catch` обрабатываются в порядке программы для поиска подходящего типа, обработчик с многоточием должен быть последним обработчиком для соответствующего блока `try`. Как правило, блок `catch(...)` используется для ведения журнала ошибок и выполнения специальной очистки перед остановкой выполнения программы.

23. Какова основная форма конструктора копий?

```
NaruTooClass(const NaruTooClass &other){}
```

Имя_класса (const имя_класса & obj) { // тело конструктора }.

Здесь `obj` - это ссылка на объект или адрес объекта.

24. Конструктор копии.

Помимо стандартного конструктора по умолчанию, каждый класс также имеет неявно определенный конструктор копирования (его можно переопределить). Конструктор копирования получает в качестве константного параметра другой объект этого класса, который будет копировать. По умолчанию происходит побитовое копирование. То есть копируется значение элементов объекта. Если переменная хранит адрес (является указателем), то копируется указатель, А НЕ ЭЛЕМЕНТ ПО УКАЗАТЕЛЮ, это может вызывать исключительные ситуации.

Общий вид конструктора по умолчанию:

```
NaruTooClass(const NaruTooClass &other){}
```

Конструктор копирования вызывается всякий раз, когда создается копия объекта

25. Контейнеры и итераторы.

Контейнер - объект, предназначенный для хранения других объектов

Примеры контейнеров:

1. `arr` (array, обычный массив)
2. `vector` (динамический массив)
3. `map/multimap` (ассоциативный массив, также массив "ключ" - "значение")
4. `list` (в плане C++ - именно двунаправленный список)

5. **set** (множество, хранящее только уникальные элементы, в более простом смысле – просто множество)

Для примеров более чем хватит массива, вектора и карты
Итератор – объект, предоставляющий доступ к элементам
определенного контейнера

Итератор	Описание
Произвольного доступа (random access)	Используется для считывания и записи значений. Доступ к элементам произвольный
Двухнаправленный (bidirectional)	Используется для считывания и записи значений. Может проходить контейнер в обоих направлениях
Однонаправленный (forward)	Используется для считывания и записи значений. Может проходить контейнер только в одном направлении
Ввода (input)	Используется только для считывания значений. Может проходить контейнер только в одном направлении
Вывода (output)	Используется только для записи значений. Может проходить контейнер только в одном направлении

Синтаксис итератора:

<тип контейнера><тип данных>::iterator<имя итератора>

Пример:

```
vector<string>::iterator vect
```

26. Контейнер – динамический массив.

Вектор – также динамический массив – контейнер для хранения элементов одного типа, способный расширяться по мере использования программы (=динамическая структура)

Синтаксис объявления:

```
vector <тип данных> <имя объекта>
```

Пример:

```
vector <string> vect
```

Обходить элементы вектора можно как при помощи доступа по индексу, так и при помощи итератора

Примеры методов данного класса:

Метод	Описание
<code>at (i)</code>	Возвращает ссылку на элемент, заданный параметром <code>i</code>
<code>back ()</code>	Возвращает ссылку на последний элемент вектора
<code>begin ()</code>	Возвращает итератор первого элемент вектора
<code>capacity ()</code>	Возвращает текущую емкость вектора
<code>clear ()</code>	Удаляет все элементы вектора
<code>empty ()</code>	Возвращает истинное значение если вектор пуст, иначе ложь
<code>end ()</code>	Возвращает итератор для конца вектора
<code>erase (iterator it)</code>	Удаляет элемент, на который указывает итератор <code>it</code> . Возвращает итератор элемента, который расположен следующим за удаленным
<code>erase (iterator start, iterator end)</code>	Удаляет элементы, заданные между итераторами <code>start</code> и <code>end</code> . Возвращает итератор элемента, который расположен следующим за последним удаленным
<code>front ()</code>	Возвращает ссылку на первый элемент вектора
<code>insert (iterator it, const T & val = T ())</code>	Вставляет параметр <code>val</code> перед элементом, заданным итератором <code>it</code> . Возвращает итератор элемента
<code>pop_back ()</code>	Удаляет последний элемент вектора
<code>push_back (const T & val)</code>	Добавляет в конец вектора элемент, значение которого равно параметру <code>val</code>

27. Какое условие является обязательным для присвоения одного объекта другому?

Они должны принадлежать одному классу; (либо объект, принимающий значение, должен принадлежать классу, наследуемому от класса присваиваемого объекта (выше по иерархии)). ГРАЧ СКАЗАЛ, ЧТО ЭТО НЕ ТАК)

28. Контейнер – ассоциативный список.

Ассоциативный список (с хуя ли он список?), также известный как **map – структура данных** (построенная на красно-черном дереве (число для понтов, если захотите путура удивить)), **хранящая не просто элемент, а пару ключ-значение, где для каждого ключа ровно одно значение.**

Методы данного чуда:

Метод	Описание
<code>assign (m, const T & значение = T())</code>	Присваивает списку m элементов, каждый элемент равно параметру значение
<code>back ()</code>	Возвращает ссылку на последний элемент списка
<code>begin ()</code>	Возвращает итератор первого элемент списка
<code>clear ()</code>	Удаляет все элементы списка
<code>empty ()</code>	Возвращает истинное значение если список пуст, иначе ложь
<code>end ()</code>	Возвращает итератор конца списка
<code>Erase (iterator it)</code>	Удаляет элемент, на который указывает итератор it. Возвращает итератор, указывающий на элемент, который расположен после удаленного
<code>erase (iterator start, iterator end)</code>	Удаляет элементы, заданные между итераторами start и end. Возвращает итератор, указывающий на элемент, который расположен за последним удаленным
<code>front ()</code>	Возвращает ссылку на первый элемент списка
<code>insert (iterator it, const T & val = T())</code>	Вставляет параметр val перед элементом, заданным итератором it. Возвращает итератор элемента
<code>pop_back ()</code>	Удаляет последний элемент списка
<code>push_back (const T & val)</code>	Добавляет в конец списка элемент, значение которого равно параметру val

В связи с тем, что данный контейнер `.v` отличие от вектора, построен не на массиве, а на дереве, время выполнения многих операций отличается

операция добавления: $O(\log(N))$

поиска: $O(\log(N))$

удаления: $O(\log(N))$

Эти свойства вытекают из устройства бинарного дерева – мы не можем обратиться к элементу за $O(1)$ по индексу, так как такового индекса у нас не существует.

Также существует контейнер, который позволяет для каждого ключа хранить не обязательно ровно одно значение. Такой контейнер называется **multimap**.

Также в связи с устройством данных контейнеров не представляется возможным получить напрямую доступ к какой-либо паре – для прохождения по `map/multimap` используется только итератор.

Синтаксис объявления самих структур:

`map<тип данных> <имя структуры>`

`multimap<тип данных> <имя структуры>`

Синтаксис объявления итератора:

`map<тип данных>::iterator <имя итератора>`

`multimap<тип данных>::iterator <имя итератора>`

А ТЕПЕРЬ, ДЕТИ МОИ, ПРИГОТОВЬТЕСЬ СТРАДАТЬ. СЕЙЧАС МЫ БУДЕМ РАССМАТРИВАТЬ СТРОЕНИЕ ДАННОЙ ХЕРНИ.

Будем вводить одно понятие за другим. пока не наступит катарсис

Граф - совокупность множеств $\{V, E\}$, где V - непустое множество вершин графа, E - множество ребер графа.

Дерево - связный ациклический граф

Двоичное дерево - структура данных, в которой каждый узел дерева может иметь не более двух потомков (0, 1, 2). Двоичное дерево уже является ориентированным графом, т.е. каждое ребро в ней - дуга строго от одного объекта (строго более высокого в уровне иерархии), до другого (строго более низкого в уровне иерархии)

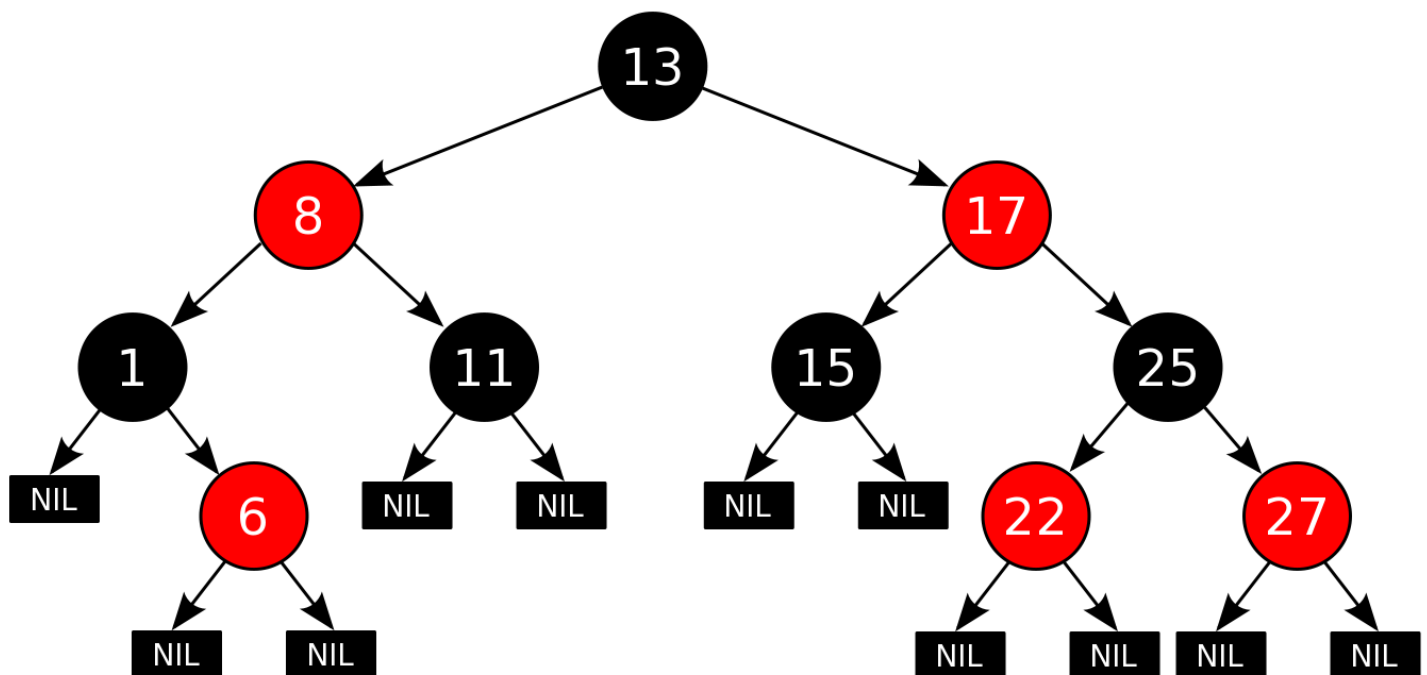
Двоичное дерево поиска - такое двоичное дерево, где выполняется условие:

1. Для любого узла X выполняется условие $key[left[X]] < key[X] \leq key[right[X]]$, то есть для любого родительского узла ключ узла слева будет меньше ключа род узла, а для ключа узла справа ключ будет больше родительского ключа.

Короче, дерево кидает все значения. меньшие род значения, влево, а остальные вправо

Самобалансированное дерево - такое двоичное дерево поиска, для которого высота его правого поддерева примерно равна высоте левого поддерева (примерно - то есть отличаются по высоте не более, чем на 1)

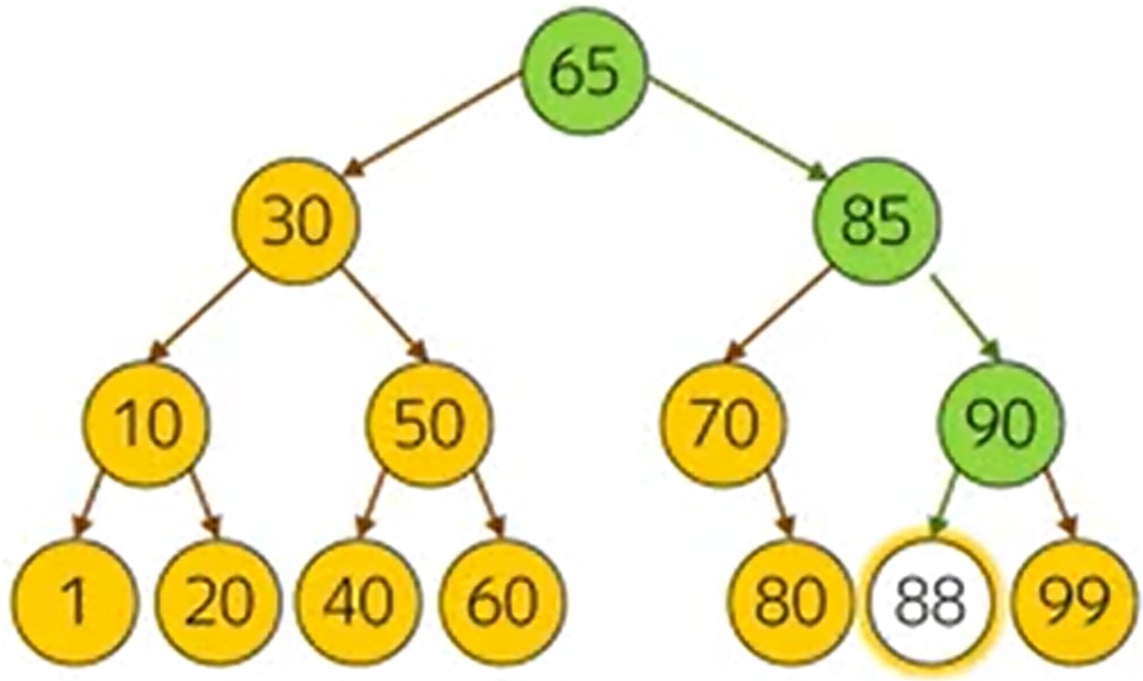
Красно-черное дерево, также **RB-tree** (бля неужели я дошел до сюда) - такое двоичное дерево поиска, которое обладает механизмом балансировки благодаря введению нового параметра для узла - цвета.



Рассматривать подробно что и как не будем, в пизду. Просто запомните, что дерево балансируется исходя из цвета узлов.

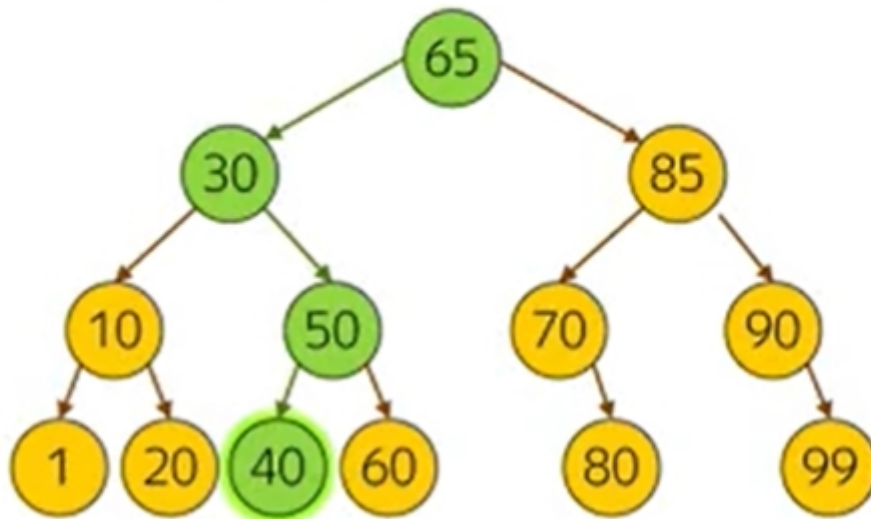
insert - сложность операции $O(\log N)$, N - количество узлов в дереве

Вставка числа 88



find - сложность операции $O(\log N)$, N - количество узлов в дереве

Поиск числа 40



delete работает по сути на find, так что сложность такая же, а картинки не нашел

29. Какой тип операций ведет к вызову конструктора копий?

Создание нового объекта на основе старого. Передача объекта в метод или функцию в качестве параметра создает копию, НО БЕЗ КОНСТРУКТОРА КОПИРОВАНИЯ (по путиридзе).

30. Конструктор и деструктор объекта.

Наряду с функциями-членами (методами) классов, которые пользователь определяет сам. в классе всегда присутствуют два метода вне зависимости от того, были ли они явно объявлены пользователем или нет. Эти методы носят названия конструктор и деструктор

Чтобы настроить, как класс инициализирует его члены или вызывать функции при создании объекта класса, возможно определение конструктора. **Конструктор** – это особый метод класса, который выполняется автоматически в момент создания объекта класса

Деструктор – также особый метод класса, который срабатывает во время уничтожения объектов класса. Чаще всего его роль заключается в том, чтобы освободить динамическую память, которую выделял конструктор для объекта. Если деструктор не определен, компилятор предоставит деструктор по умолчанию (деструктор по умолчанию – деструктор без параметров)

Отличие конструктора и деструктора от любых других методов:

1. всегда называются одинаково – именем класса и ~ + имя класса в случае деструктора
2. Всегда присутствуют – в случае. когда нет явной инициализации в классе, вызывается конструктор и деструктор без параметров

31. Класс map и multimap.

См 28 билет

32. Можно ли адрес объекта передать функции в качестве аргумента?

Да, можно – передача доступа при помощи предоставления в качестве аргумента функции указателя на объект

33. Множественное наследование.

Наследование – принцип, согласно которому класс может использовать переменные и методы другого класса как свои собственные. Класс. от которого наследуются методы. называется

родительским; класс, который наследует методы, называется дочерним

Множественное наследование происходит, когда дочерний класс имеет два или более родительских класса

```
1  #include <iostream>
2
3  class Computer {
4  public:
5      void turn_on() {
6          std::cout << "Welcome to Windows 95" << std::endl;
7      }
8  };
9
10 class Monitor {
11 public:
12     void show_image() {
13         std::cout << "Imagine image here" << std::endl;
14     }
15 };
16 class Laptop : public Computer, public Monitor {};
17
18 int main() {
19     Laptop Laptop_instance;
20     Laptop_instance.turn_on();
21     Laptop_instance.show_image();
22     return 0;
23 }
```

Множественное наследование связано с проблематикой неопределенного наследования - случай, когда в обоих род классах есть метод с одинаковым названием и он вызывается из дочернего - дочерний класс не может понять, к какому именно классу обращаться и выдается compile error

```
1  #include <iostream>
2
3  class Computer {
4  private:
5      void turn_on() {
6          std::cout << "Computer is on." << std::endl;
7      }
8  };
9
10 class Monitor {
11 public:
12     void turn_on() {
13         std::cout << "Monitor is on." << std::endl;
14     }
15 };
16
17 class Laptop : public Computer, public Monitor {};
18
19 int main() {
20     Laptop Laptop_instance;
21     Laptop_instance.turn_on();
22     return 0;
23 }
```

вод

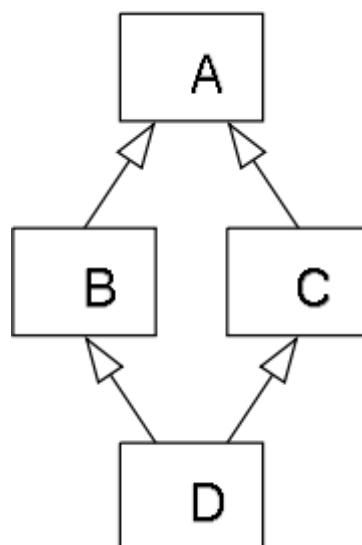
показать выходные данные из: Система управле

открытие репозитория:

inline void Monitor::turn_on()
Поиск в Интернете

"Laptop::turn_on" не является однозначным
Поиск в Интернете

Next level problem - ромбовидное наследование



Эта проблема возникает, когда классы B и C наследуют A, а класс D наследует B и C.

К примеру, классы А, В и С определяют метод `print_letter()`. Если `print_letter()` будет вызываться классом D, неясно какой метод должен быть вызван — метод класса А, В или С

Методы решения:

- 1.Сходить нахуй
- 2.вызвать метод конкретного родительского класса (например, через приведение указателя)
- 3.переопределить проблематичный метод в последнем дочернем классе (хуяк хуяк виртуальное наследование)

34. Можно ли использовать инструкцию `throw`, если ход выполнения программы не затрагивает инструкции, расположенные в блоке `try`?

Нет. Любая инструкция, которая возбуждает исключительную ситуацию, должна выполняться внутри блока `try`.

35. Может ли быть инициализирован массив, память для которого выделяется динамически?

Оказывается, можно

```
int len;  
cin >> len;  
char* c = new char[len]();
```

Инициализирует массив из `'\0'`

36. Наследование. Реализация наследования на языке C++.

Реальные объекты постоянно развиваются, претерпевают определенные изменения. Появляются новые модели, версии объектов. Но при их разработке новая модель содержит множество характеристик и функционала предыдущей версии. Пример: постоянное совершенствование коммуникационной и вычислительной техники, всевозможных гаджетов, популярного программного обеспечения, автомобилей и т.д.

В языке C++ возможность разработать новые классы на базе уже существующих реализован посредством механизма наследования.

Для наследования одного класса другим, используется следующая основная форма записи:

```
class «имя производного класса» : «спецификатор доступа» «имя базового класса»
```

Здесь «спецификатор доступа» – это одно из трех ключевых слов: `public`, `private` или `protected`. Рассмотрим спецификаторы `public` и `private`.

Спецификатор доступа определяет, как элементы базового класса (base class) наследуются производным классом (derived class).

Пример

```
#include <iostream>
using namespace std;
class base {
    int x;
public:
    void setx ( int n ) { x = n; }
    void showx ( ) { cout << x << "\n"; }
}; // Класс наследуется как открытый
class derived: public base
{
    int y;
public:
    void sety ( int n ) { y = n; }
    void showy ( ) { cout << y << "\n"; }
};
int main ( )
{
    derived ob;
    ob.setx ( 10 ); // доступ к члену базового класса
    ob.sety ( 20 ); // доступ к члену производного класса
    ob.showx ( );   // доступ к члену базового класса
    ob.showy ( );   // доступ к члену производного класса
    return 0;
}
```

37. Объявление элементов класса спецификацией `static`.

Статические свойства (поля, переменные).

Статические поля и методы объявляются с помощью модификатора `static`. Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

Статические поля

Статические свойства применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля

существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

Свойства статических полей:

1) память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора (определение должно быть записано вне класса):

```
#include <iostream>
class Example {
public:
    static int value; //объявление в классе
};
int Example::value; //определение статического поля в
//глобальной области, по умолчанию инициализируется //нулем.
// int Example::value=10; // пример инициализации
//произвольным значением
```

2) статические поля доступны как через имя класса, так и через имя объекта

```
Example object1, *object2;
...
cout<<Example::value<<object1.value<<object2->value;
```

3) на статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как `private`, нельзя изменить с помощью операции доступа к области действия; это можно сделать только с помощью статических методов;

4) память, занимаемая статическим полем, не учитывается при определении размера с помощью операции `sizeof`.

Статические методы

Это можно рассматривать как самостоятельный объект имеющий интерфейс ко всем объектам данного класса.

38. Объявление элементов класса спецификацией `const`.

Переменные, объявленные с использованием спецификатора `const`, не могут изменять свои значения во время выполнения программы. Любой `const` – переменной можно присвоить некоторое начальное значение.

Спецификатор сообщает компилятору о недопустимости изменения значения данного элемента. Таким образом еще на этапе разработки кода предотвращается модификация переменной при выполнении программы.

39. Объявление объекта и доступ к его элементам.

Объект – прототип класса, которому могут быть заданы свойства и у которого можно вызывать методы

В дальнейшем будем считать, что все методы могут быть вызваны, а ко всем свойствам можно обратиться (крч с модификаторами доступа все в порядке)

Объект может быть объявлен как статически, так и динамически
Статическое объявление объекта

Доступ к объектам, объявленным разными способами, также используют различные оператор. Различают два оператора доступа к членам – `.` и `->`. Второй оператор отличается от первого тем, что также дополнительно содержит операцию разыменования указателя. Также отличие состоит в том, что второй оператор можно перегрузить, в то время как первый оператор является не перегружаемым.

Статической объявление объекта: "Имя класса" "Имя объекта"

Динамической объявление: "Имя класса" * "Имя объекта" = new "имя класса"

40. Объединение. Назначение и синтаксис описания.

В C++ объединение также представляет собой тип класса, в котором функции и данные могут содержаться в качестве его членов. Объединение похоже на структуру тем, что в нем по умолчанию все члены открыты до тех пор, пока не указан спецификатор `private`. Главное же в том, что в C++ все данные, которые являются членами объединения, находятся в одной и той же области памяти (точно так же, как и в C). Объединения могут содержать конструкторы и деструкторы. Объединения C++ совместимы с объединениями C. Если в отношениях между структурами и классами существует, на первый взгляд, некоторая избыточность, то об объединениях этого сказать нельзя. В объектно ориентированном языке важна поддержка инкапсуляции.

Поэтому способность объединений связывать воедино программу и данные позволяет создавать такие типы классов, в которых все данные находятся в общей области памяти. Это именно то, чего нельзя сделать с помощью классов. Применительно к C++ имеется несколько ограничений, накладываемых на использование объединений. Во-первых, они не могут наследовать какой бы то ни было класс и не могут быть базовым классом для любого другого класса. Объединения не могут иметь статических членов. Они также не должны содержать объектов с конструктором или деструктором, хотя сами по себе объединения могут иметь конструкторы и деструкторы.

В C++ имеется особый тип объединения — это анонимное объединение (anonymous union). Анонимное объединение не имеет имени типа и следовательно нельзя объявить переменную такого типа. Вместо этого анонимное объединение просто сообщает компилятору, что все его члены будут находиться в одной и той же области памяти. Во всех остальных отношениях члены объединения действуют и обрабатываются как самые обычные переменные. То есть, доступ к членам анонимного объединения осуществляется непосредственно, без использования оператора точка (.). Например, рассмотрим следующий фрагмент:

```
union { // анонимное объединение
    int i;
    char ch [4];
};

// непосредственный доступ к переменным i и ch
i = 10;
ch[0] = 'X';
```

41. Объекты в качестве возвращаемого значения функции.

Так же как объект может быть передан функции в качестве аргумента, он может быть и возвращаемым значением функции. Для этого, во-первых, в объявлении функции указать тип класса в качестве типа возвращаемого значения. Во-вторых, объект типа класса возвращается с помощью обычной инструкции return. Имеется одно важное замечание по поводу объекта в качестве возвращаемого значения функции: если функция возвращает объект, то для хранения возвращаемого значения автоматически создается временный объект. После того как значение возвращено, этот объект удаляется. Во многих учебниках указано, что **в момент возврата объекта происходит вызов конструктора копии и деструктора, что может исказить данные**

объекта (пример – динамический массив в объекте, копируется ссылка на массив, а не сам массив, при вызове деструктора копии массив удаляется. См билет про конструктор копирования).

42. Определение адреса перегруженной функции.

Так же, как и в С, вы можете присвоить адрес функции указателю и получить доступ к функции через этот указатель. Адрес функции можно найти, если поместить имя функции в правой части инструкции присваивания без всяких скобок или аргументов. Например, если `zap()` – это функция, причем правильно объявленная, то корректным способом присвоить переменной `p` адрес функции `zap()` является инструкция:

```
p = zap;
```

В языке С любой тип указателя может использоваться как указатель на функцию, поскольку имеется только одна функция, на которую он может ссылаться. Однако в С++ ситуация несколько более сложная, поскольку функция может быть перегружена. Способ объявления указателя и определяет то, адрес какой из перегруженных версий функции будет получен. Уточним, объявления указателей соответствуют объявлениям перегруженных функций. Функция, объявлению которой соответствует объявление указателя, и является искомой функцией.

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

```
// вывод заданного в переменной count числа пробелов
void space (int count) {
    for ( ; count; count - ) cout << ' ';
}
```

```
// вывод заданного в переменной count числа символов,
// вид которых задан в переменной ch
void space (int count, char ch) {
    for (; count; count - ) cout << ch;
}
```

```
int main () {
    // Создание указателя на функцию с одним целым
    // параметром.
    void (*fp1) (int) ;
    // Создание указателя на функцию с одним целым и
    // одним символьным параметром.
    void (*fp2) (int, char);
```



```

fp1 = space; // получение адреса функции space (int)
fp2 = space; // получение адреса функции space (int, char)
fp1 (22); // выводит 22 пробела
cout << " | \n";
fp2 (30, 'x'); // выводит 30 символов x
cout << " \n";
return 0;
}

```

43. Определение системы и три примера систем.

Система – множество взаимосвязанных и взаимодействующих объектов для решения одной или множества задач (достижения одной или множества целей) .

Примеры систем:

- Самолет
- Компьютер
- Человек

44. Перегрузка бинарных операторов.

Бинарный оператор - оператор, производящий действие с двумя объектами.

Перегрузка операторов в [программировании](#) — один из способов реализации [полиморфизма](#), заключающийся в возможности одновременного существования в одной [области видимости](#) нескольких различных вариантов применения операторов, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются (где параметры - объекты, с которыми производятся операции)

Любая **бинарная** операция @ может быть определена **двумя способами**: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром.

Переопределяемые бинарные операторы

Оператор	Имя
,	Запятая
!=	Неравенство
%	Модуль
%=	Модуль/назначение
&	Побитовое И
&&	Логическое И
&=	Побитовое И/назначение
*	Умножение
*=	Умножение/назначение
+	Сложение
+=	Сложение/назначение
-	Вычитание
-=	Вычитание/назначение
->	Выбор члена
->*	Выбор указателя на член
/	Отдел
/=	Деление/назначение
<	Меньше чем
<<	Сдвиг влево
<<=	Сдвиг влево/назначение
<=	Меньше или равно
=	Назначение
==	Равенство
>	Больше
>=	Больше или равно
>>	Сдвиг вправо
>>=	Сдвиг вправо/назначение
^	Исключающее ИЛИ
^=	Исключающее ИЛИ/назначение
	Побитовое ИЛИ
=	Побитовое включающее ИЛИ/назначение
	Логическое ИЛИ

```

struct Complex {
    Complex( double r, double i ) : re(r), im(i) {}
    Complex operator+( Complex &other );
    void Display( ) {    cout << re << ", " << im << endl; }
private:
    double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
    return Complex( re + other.re, im + other.im );
}

```

При реализации дружественную функцию подается не один параметр, а два.

45. При наследовании одного класса другим, когда вызываются конструкторы классов? Когда вызываются их деструкторы?

Конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. Деструкторы вызываются в обратном порядке. В итоге получается что-то по типу кольца – конструктор род класса начинает, деструктор род класса заканчивает (как кольцевая рифма)

46. Полиморфизм.

При конструировании новых версии изделий, многие элементы сохраняют форму (наименование, способ активизации), но меняют функционал поведения.

Для реализации данного свойства новой версии изделия, в объектно-ориентированных языках реализован посредством механизма полиморфизма.

Изменение функциональности поведения объекта (изменения алгоритма) отображается в части реализации метода в описании класса. Сохранению формы соответствует неизменность наименования метода, при различии алгоритмов реализации. Полиморфизм иногда характеризуется фразой «один интерфейс, много методов».

При проектировании иерархии классов, некоторые методы сохраняя названия, изменяются по сути (меняется алгоритм реализации). Полиморфизм – механизм обеспечивающий возможность определения различных реализации метода одним названием для классов различных уровней иерархии.

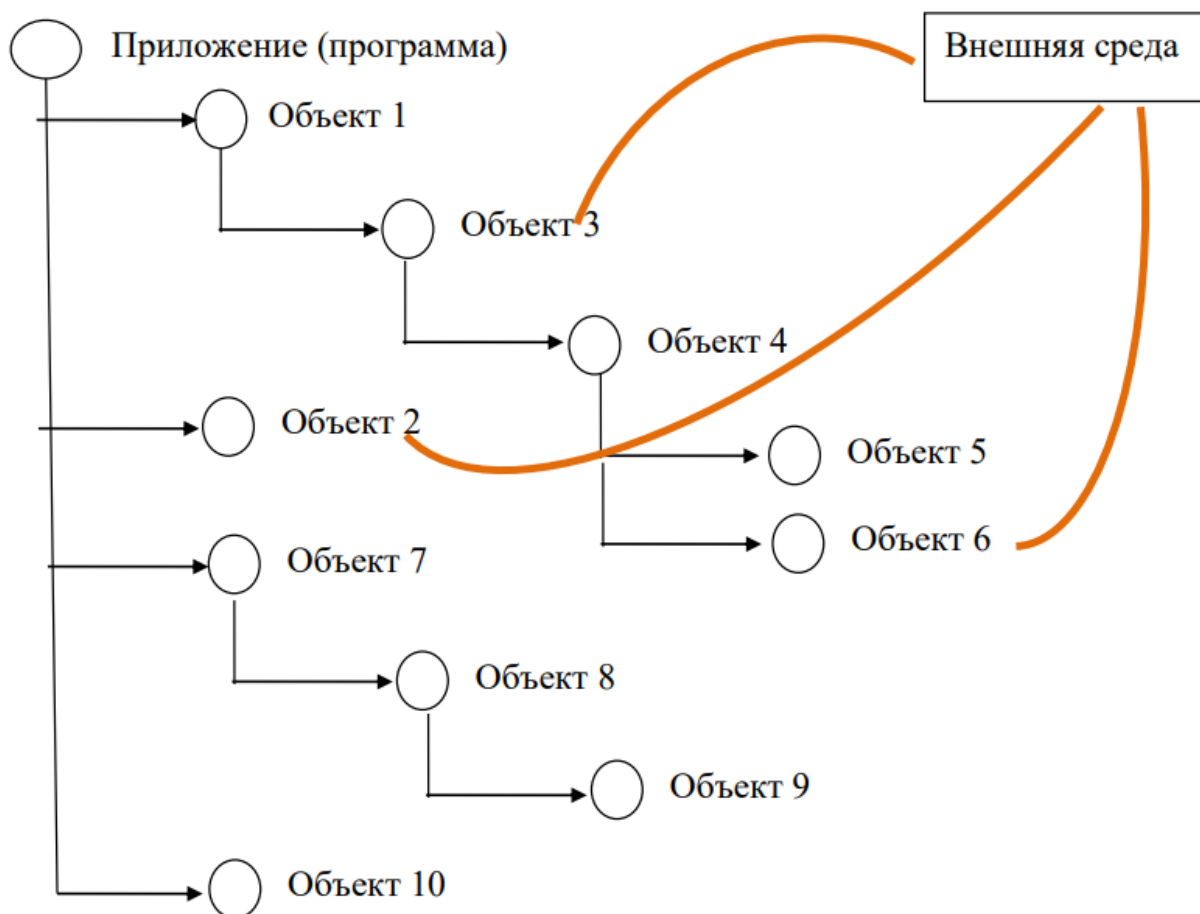
47. Параметризованные конструкторы.

Каждый класс имеет базовый конструктор по умолчанию. Такой конструктор не принимает параметров, что не всегда удобно. Мы можем перегрузить конструктор по умолчанию новым параметризованным конструктором. Имя параметризованного конструктора совпадает с именем класса.

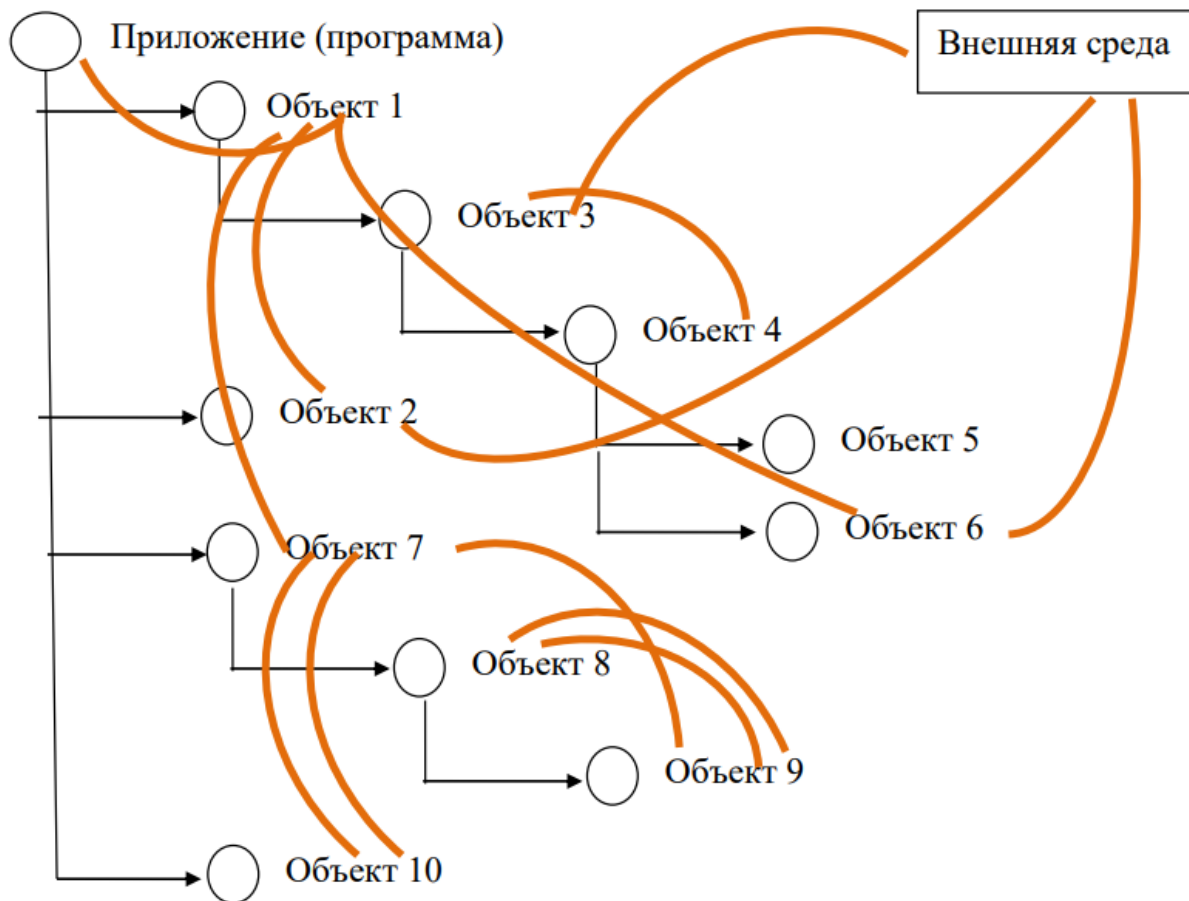
48. Программа – система.

Любое приложение, разработанное с соблюдением требований объектно ориентированного подхода, является системой. Программирование на объектно-ориентированном языке соответствует работе конструктора (лего, а не параметризованного). Когда из элементов (объектов) собирается система (программа). При запуске этой программы реализованная система начинает функционировать, решать задачи. Разработка программ становится естественным процессом отображения реальной деятельности.

Выделим два уровня сложности построения приложения. Первый уровень сложности: взаимодействие допустимо между конструктивно взаимосвязанными объектами и внешней средой.



Второй уровень сложности: взаимодействие допустимо между любыми объектами системы и внешней средой.



После запуска программы выполняется создание объектов, сопровождение их жизненного цикла и обеспечение функционирования программы как системы.

50. Перегрузка оператора индексации массивов []

Оператор индексации `[]` позволяет получить элемент объекта с индексом `i`.
Перегружается ТОЛЬКО как функция-член

```

1  #include <iostream>
2  class IntList{
3      private:
4          int m_list[10] = {1,2,3,4,5,6,7,8,9,0};
5      public:
6          int& operator[] (int index);
7  };
8
9  int& IntList::operator[] (int index){
10     return m_list[index];
11 }
12 int main (){
13     IntList intList;
14     std::cout << intList[7];
15 }

```

51. Присвоение объектов.

Объекты одного класса можно присваивать друг другу. Операция выполняется посредством оператора присвоения. Синтаксис выражения:

«имя объекта 1» = «имя объекта 2»;

Объекту «имя объекта 1» побитно (поразрядно) присваивается содержимое всех свойств (элементов данных) объекта «имя объекта 2».

Пример

```

#include <iostream>
using namespace std;

// ----- Заголовочная часть.
class cl_1 {
private:
    int i;
public:
    void set_i ( int k );
    void show_i ( );
};

// ----- Часть реализации.
void cl_1 :: set_i ( int k ) {
    i = k;
}

```

```

}
void cl_1 :: show_i ( ) {
    cout << "i = " << i << "\n";
}

// ----- Основная программа
int main ( ) {
    cl_1 ob_1, ob_2;          // объявление объектов
    ob_1.set_i ( 11 );        // инициализация свойства i в ob_1.
    ob_2.set_i ( 15 );        // инициализация свойства i в ob_2.
    ob_1.show_i ( );          // вывод значения свойства объекта
ob_1.
    ob_1 = ob_2;              // присвоение объекту ob_1 объекта
ob_2.
    ob_1.show_i ( );          // вывод значения свойства объекта
ob_1.
    return 0;
}

```

Программа выведет на консоль следующее:

```

i = 11
i = 15

```

52. Приведение типов.

Понимаю, что написано много, но я не нашёл, что здесь можно сократить

Оператор **dynamic_cast** реализует приведение типов в динамическом режиме, что позволяет контролировать правильность этой операции во время работы программы. Если при выполнении оператора **dynamic_cast** приведения типов не произошло, будет выдана ошибка приведения типов. Основная форма оператора **dynamic_cast**:

```
dynamic_cast <целевой_тип> ( выражение )
```

целевой_тип — это тип, которым должен стать тип параметра «выражение» после выполнения операции приведения типов.

Целевой тип должен быть типом указателя или ссылки и результат выполнения параметра выражение тоже должен стать указателем или ссылкой. Таким образом, оператор **dynamic_cast** используется для приведения типа одного указателя к типу другого или типа одной ссылки к типу другой. Основное назначение оператора **dynamic_cast** заключается в реализации операции приведения полиморфных типов.

Например, пусть дано два полиморфных класса В и D, причем класс D является производным от класса В, тогда оператор `dynamic_cast` всегда может привести тип указателя D* к типу указателя В*. Это возможно потому, что указатель базового класса всегда может указывать на объект производного класса. Оператор `dynamic_cast` может также привести тип указателя В* к типу указателя D*, но только в том случае, если объект, на который указывает указатель, действительно является объектом типа D.

При неудачной попытке приведения типов результатом выполнения оператора `dynamic_cast` является нуль, если в операции использовались указатели. Если же в операции использовались ссылки, возбуждается исключительная ситуация `bad_cast`.

Рассмотрим простой пример. Предположим, что `Base` – это базовый класс, а `Derived` – это класс, производный от класса `Base`.

```
Base * bp, b_ob;  
Derived * dp, d_ob;  
bp = &d_ob; // указатель базового класса  
        // указывает на объект производного класса  
dp = dynamic_cast < Derived * > ( bp )  
if ( ! dp ) cout << "Приведение типов прошло успешно";
```

Здесь приведение типа указателя `bp` базового класса к типу указателя `dp` производного класса прошло успешно, поскольку указатель `bp` на самом деле указывает на объект производного класса `Derived`. Таким образом, после выполнения этого фрагмента программы на экране появится сообщение

Приведение типов прошло успешно.

Доступны еще три операторы приведения типов, их основные формы:

```
const_cast < целевой_тип > ( выражение )  
reinterpret_cast < целевой_тип > ( выражение )  
static_cast < целевой_тип > ( выражение )
```

«целевой_тип» – это тип, которым должен стать тип параметра выражение после выполнения операции приведения типов. Как правило, указанные операторы обеспечивают более безопасный и интуитивно понятный способ выполнения некоторых видов операций преобразования, чем оператор приведения 86 типов, более характерный для языка С.

Оператор **const_cast** при выполнении операции приведения типов используется для явной подмены атрибутов const (постоянный) и/или volatile (переменный). Целевой тип должен совпадать с исходным типом, за исключением изменения его атрибутов const или volatile. Обычно с помощью оператора const_cast значение лишают атрибута const.

Оператор **static_cast** предназначен для выполнения операций приведения типов над объектами неpolиморфных классов.

Например, его можно использовать для приведения типа указателя базового класса к типу указателя производного класса. Кроме этого, он подойдет и для выполнения любой стандартной операции преобразования, но только не в динамическом режиме (т. е. не во время выполнения программы).

Оператор **reinterpret_cast** дает возможность преобразовать указатель одного типа в указатель совершенно другого типа. Он также позволяет приводить указатель к типу целого и целое к типу указателя. Оператор reinterpret_cast следует использовать для выполнения операции приведения внутренне несовместимых типов указателей.

1. В следующей программе демонстрируется использование оператора reinterpret_cast.

```
// Пример использования оператора reinterpret_cast
#include <iostream>
using namespace std;
int main ( ) {
    int* i;
    char * p = "Это строка";
    // приведение типа указателя к типу целого
    i = reinterpret_cast < int* > ( p );
    cout << *i;
    return 0;
}
```

В данной программе с помощью оператора reinterpret_cast указатель на строку превращен в целое. Это фундаментальное преобразование типа и оно хорошо отражает возможности оператора reinterpret_cast.

2. В следующей программе демонстрируется оператор const_cast.

```
// Пример. использования оператора const_cast
#include <iostream>
using namespace std;
```

```

void f ( const int * p ) {
    int * v;
    // преобразование типа лишает указатель p атрибута const
    v = const_cast < int * > ( p );
    * v = 100; // теперь указатель v может изменить объект
}
int main ( ) {
    int x = 99;
    cout << "Объект x перед вызовом функции равен: " << x <<
endl;
    f ( & x );
    cout << "Объект x после вызова функции равен: " << x <<
endl;
    return 0;
}

```

Ниже представлен результат выполнения программы:

```

Объект x перед вызовом функции равен: 99
Объект x после вызова функции равен: 100

```

Несмотря на то что параметром функции `f ()` задан постоянный указатель, вызов этой функции с объектом `x` в качестве параметра изменил значение объекта.

53. Перегрузка функций.

Перегрузка функций – это создание нескольких функций с одним именем, но с разными параметрами. Под разными параметрами понимают, что должно быть разным *количество аргументов* функции и/или их *тип*.

Перегрузка функций также называется *полиморфизмом функций*. "Поли" означает много, "морфе" – форма, то есть полиморфическая функция – это функция, отличающаяся многообразием форм.

Под полиморфизмом функции понимают существование в программе нескольких перегруженных версий функции, имеющих разные значения. Изменяя количество или тип параметров, можно присвоить двум или нескольким функциям одно и тоже имя. При этом никакой путаницы не будет, поскольку нужная функция определяется по совпадению используемых параметров. Это позволяет создавать функцию, которая сможет работать с целочисленными, вещественными значениями или значениями других типов без необходимости создавать отдельные имена для каждой функции.

Таким образом, благодаря использованию перегруженных функций, не следует беспокоиться о вызове в программе нужной функции, отвечающей типу передаваемых переменных. При вызове перегруженной функции компилятор автоматически определит, какой именно вариант функции следует использовать.

54. Перегрузка унарных операторов.

Перегрузка операторов в **программировании** — один из способов реализации **полиморфизма**, заключающийся в возможности одновременного существования в одной **области видимости** нескольких различных вариантов применения операторов, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются (где параметры - объекты, с которыми производятся операции)

Перегрузке могут быть подвергнуты следующие унарные операторы.

1. **!** (логическое НЕ)
2. **&** (адрес)
3. **~** (дополнение)
4. ***** (разыменовыватель указателя)
5. **+** (унарный плюс)
6. **-** (унарное отрицание)
7. **++** (приращение)
8. **--** (декремент)
9. операторы преобразования

Операторы **++** приращения и **--** декремента постфикса (и **--**) обрабатываются отдельно приращении и уменьшении, тк , имеется два варианта каждого из них:

- преинкрементный и постинкрементный операторы;
- преддекрементный и постдекрементный операторы.

Любая **унарная** операция **@** может быть определена **двумя способами**: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром.

```

class Person {
public:
    void operator ++()
    {
        ++age;
    }
protected:
    int age;
    ...
};

int main(void)
{
    Person John;
    ++John;
    ...
}

```

```

class Person {
protected:
    int age;
    ...
friend void operator ++(Person &);
};

void operator ++(Person &ob)
{
    ++ob.age;
}

int main (void)
{
    Person John;
    ++John;
    ...
}

```

55. Структура. Назначение и синтаксис описания.

Синтаксически класс похож на структуру. Класс и структура имеют фактически одинаковые свойства. В С++ определение структуры расширили таким образом, что туда, как и в определении класса, удалось включить функции члены, в том числе конструкторы и деструкторы. Таким образом, единственным

отличием между структурой и классом является то, что члены класса, по умолчанию, являются закрытыми, а члены структуры – открытыми. Расширенный синтаксис описания структуры:

```
struct имя_типа {  
    // открытые функции и данные – члены класса.  
private:  
    // закрытые функции и данные – члена класса  
} список_объектов
```

56. Операторы new и delete.

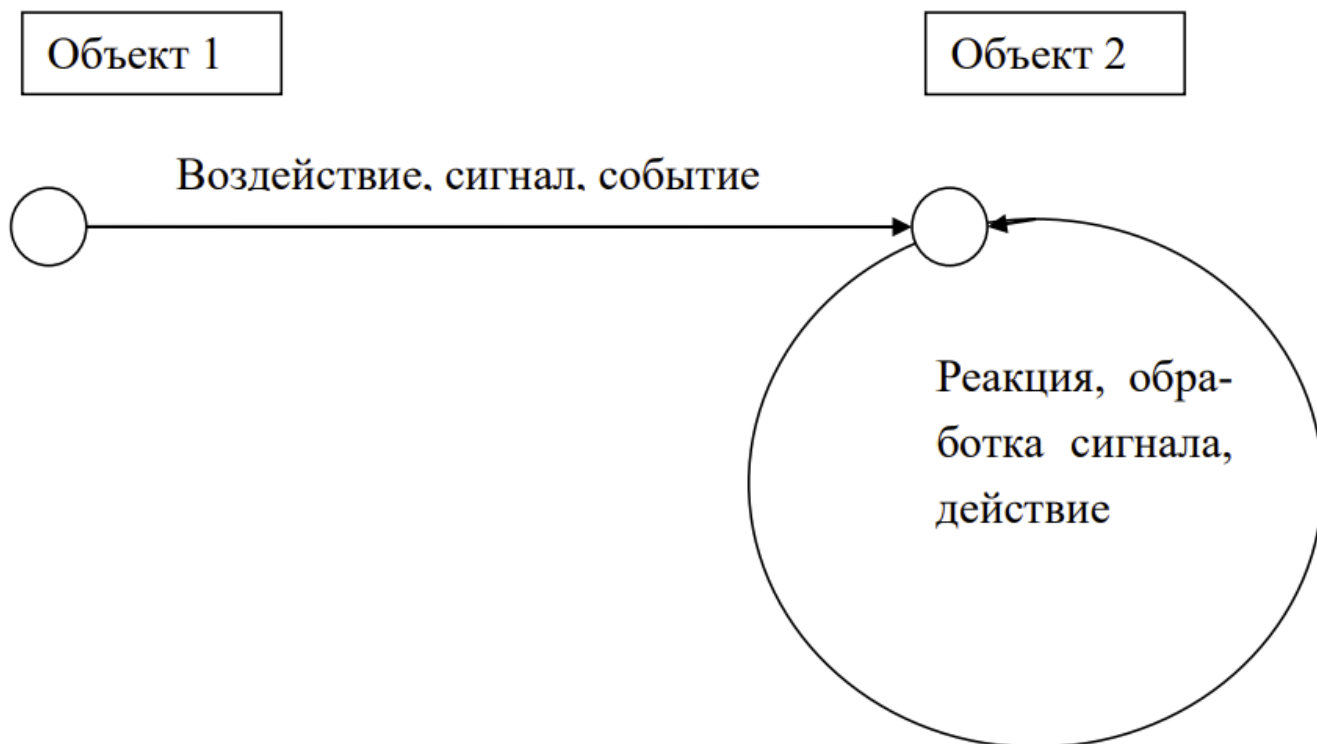
C++ поддерживает динамическое выделение и освобождение объектов с помощью операторов new и delete .

Оператор new – динамически выделяет память, достаточную для хранения объекта данного класса, обрабатывает конструктор класса, возвращает указатель на объект данного класса. Если свободной памяти недостаточно для выполнения запроса, произойдет одно из двух: либо оператор new возвращает нулевой указатель, либо будет сгенерирована исключительная ситуация.

Оператор delete освобождает память выделенную для объекта, предварительно выполнив код деструктора. Вызов оператора delete с неправильным указателем может привести к разрушению системы динамического выделения памяти и возможному краху программы.

57. Сигналы и обработчики.

Сигналы и обработчики – механизм взаимодействия объектов системы как между собой, так и с внешней средой.



Очевидное

Метод сигнала – метод объекта, “посылающий сигнал” (например, изменяя значение, переданной по ссылке переменной), другому объекту

Метод обработчика – метод объекта, “принимающий сигнал” и, определенным образом обрабатывающий его.

Чуть менее очевидное

Если говорить о реализации, то вспоминаем KL_3_3. Вводим структуру, хранящую указатель на метод сигнала объекта-эмитента, метод обработчика целевого объекта и указатель на этот самый объект, чтобы было от кого вызывать метод обработчика. Передача сигнала от объекта А к объекту В означает последовательный вызов метода сигнала и метода обработчика с одним и тем же набором аргументов.

ЗАПОМНИТЕ ДЕТИ

Параметр – переменная в функции (методе), которая будет содержать переданное снаружи значение

Аргумент – переменная или выражение, чьи значения передаются в функцию (метод) при ее (его) вызове

58. Форматированный ввод-вывод данных.

Форматированный ввод-вывод данных в C++ осуществляется за счет следующих вещей:

Форматирующих функций-членов

Флагов

Манипуляторов

```
1 //Основные форматирующие функции-члены:
2 cout.fill('/*symbol*'); // устанавливает символ заполнитель
3 // где symbol - символ заполнитель, символ передаётся в одинарных кавычках
4
5 cout.width(/*width_field*/); // задает ширину поля
6 // где width_field - количество позиций(одна позиция вмещает один символ)
7
8 cout.precision(/*number*/); // задает количество знаков после десятичной точки
9 // где number - количество знаков после десятичной точки
```

Флаги форматирования позволяют включить или выключить один из параметров ввода/вывода.

```
// установка нескольких флагов
cout.setf( ios::/*name_flag1*/ | ios::/*name_flag2*/ | ios::/*name_flag_n*/ );

// снятие нескольких флагов
cout.unsetf( ios::/*name_flag1*/ | ios::/*name_flag2*/ | ios::/*name_flag_n*/ );
```

Флаг	Назначение	Пример	Результат
boolalpha	Вывод логических величин в текстовом виде (<code>true</code> , <code>false</code>)	<code>cout.setf(ios::boolalpha); bool log_false = 0, log_true = 1; cout << log_false << endl << log_true << endl;</code>	false true
oct	Ввод/вывод величин в восьмеричной системе счисления (сначала снимаем флаг <code>dec</code> , затем устанавливаем флаг <code>oct</code>)	<code>cout.unsetf(ios::dec); cout.setf(ios::oct); int value; cin >> value; cout << value << endl;</code>	Ввод: 99 ₁₀ Вывод: 143 ₈
dec	Ввод/вывод величин в десятичной системе счисления (флаг установлен по умолчанию)	<code>cout.setf(ios::dec); int value = 148; cout << value << endl;</code>	148
hex	Ввод/вывод величин в шестнадцатеричной системе счисления (сначала снимаем флаг <code>dec</code> , затем устанавливаем флаг <code>hex</code>)	<code>cout.unsetf(ios::dec); cout.setf(ios::hex); int value; cin >> value; cout << value << endl;</code>	Ввод: 99 ₁₀ Вывод: 63 ₁₆
showbase	Выводить индикатор основания системы счисления	<code>cout.unsetf(ios::dec); cout.setf(ios::oct ios::showbase); int value; cin >> value; cout << value << endl;</code>	Ввод: 99 ₁₀ Вывод: 0143 ₈
uppercase	В шестнадцатеричной системе счисления использовать буквы верхнего регистра(по умолчанию установлены буквы нижнего регистра)	<code>cout.unsetf(ios::dec); cout.setf(ios::hex ios::uppercase); int value; cin >> value; cout << value << endl;</code>	Ввод: 255 ₁₀ Вывод: FF ₁₆
showpos	Вывод знака плюс <code>+</code> для положительных чисел	<code>cout.setf(ios::showpos); int value = 15; cout << value << endl;</code>	+15
scientific	Вывод чисел с плавающей точкой в экспоненциальной форме	<code>cout.setf(ios::scientific); double value = 1024.165; cout << value << endl;</code>	1.024165e+003
fixed	Вывод чисел с плавающей точкой в фиксированной форме(по умолчанию)	<code>double value = 1024.165; cout << value << endl;</code>	1024.165
right	Выравнивание по правой границе(по умолчанию). Сначала необходимо установить ширину поля(ширина поля должна быть заведомо большей чем, длина выводимой строки).	<code>cout.width(40); cout << «cppstudio.com» << endl;</code>	__cppstudio.com
left	Выравнивание по левой границе. Сначала необходимо установить ширину поля(ширина поля должна быть заведомо большей чем, длина выводимой строки).	<code>cout.setf(ios::left); cout.width(40); cout << «cppstudio.com» << endl;</code>	cppstudio.com__

Манипулятор — объект особого типа, который управляет потоками ввода/вывода, для форматирования передаваемой в потоки информации. Отчасти манипуляторы дополняют функционал, для

форматирования ввода/вывода. Но большинство манипуляторов выполняют точно, то же самое, что и функции с флагами форматирования.

Манипулятор	Назначение	Пример	Результат
endl	Переход на новую строку при выводе	<code>cout << «website:» << endl << «cppstudio.com»;</code>	website: cppstudio.com
boolalpha	Вывод логических величин в текстовом виде (<code>true</code> , <code>false</code>)	<code>bool log_true = 1; cout << boolalpha << log_true << endl;</code>	true
noboolalpha	Вывод логических величин в числовом виде (<code>true</code> , <code>false</code>)	<code>bool log_true = true; cout << noboolalpha << log_true << endl;</code>	1
oct	Вывод величин в восьмеричной системе счисления	<code>int value = 64; cout << oct << value << endl;</code>	100₈
dec	Вывод величин в десятичной системе счисления (по умолчанию)	<code>int value = 64; cout << dec << value << endl;</code>	64₁₀
hex	Вывод величин в шестнадцатеричной системе счисления	<code>int value = 64; cout << hex << value << endl;</code>	40₈
showbase	Выводить индикатор основания системы счисления	<code>int value = 64; cout << showbase << hex << value << endl;</code>	0x40
noshowbase	Не выводить индикатор основания системы счисления (по умолчанию).	<code>int value = 64; cout << noshowbase << hex << value << endl;</code>	40
uppercase	В шестнадцатеричной системе счисления использовать буквы верхнего регистра (по умолчанию установлены буквы нижнего регистра).	<code>int value = 255; cout << uppercase << hex << value << endl;</code>	FF₁₆
nouppercase	В шестнадцатеричной системе счисления использовать буквы нижнего регистра (по умолчанию).	<code>int value = 255; cout << nouppercase << hex << value << endl;</code>	ff₁₆
showpos	Вывод знака плюс <code>+</code> для положительных чисел	<code>int value = 255; cout << showpos << value << endl;</code>	+255

noshowpos	Не выводить знак плюс + для положительных чисел (по умолчанию).	<pre>int value = 255; cout <<noshowpos<< value << endl;</pre>	255
scientific	Вывод чисел с плавающей точкой в экспоненциальной форме	<pre>double value = 1024.165; cout << scientific << value << endl;</pre>	1.024165e+003
fixed	Вывод чисел с плавающей точкой в фиксированной форме (по умолчанию).	<pre>double value = 1024.165; cout << fixed << value << endl;</pre>	1024.165
setw(int number)	Установить ширину поля, где number — количество позиций, символов (выравнивание по умолчанию по правой границе). Манипулятор с параметром.	<pre>cout << setw(40) << «cppstudio.com» << endl;</pre>	__cppstudio.com
right	Выравнивание по правой границе(по умолчанию). Сначала необходимо установить ширину поля(ширина поля должна быть заведомо большей чем, длина выводимой строки).	<pre>cout << setw(40) << right << «cppstudio.com» << endl;</pre>	__cppstudio.com
left	Выравнивание по левой границе. Сначала необходимо установить ширину поля(ширина поля должна быть заведомо большей чем, длина выводимой строки).	<pre>cout << setw(40) << left << «cppstudio.com» << endl;</pre>	cppstudio.com__
setprecision(int count)	Задаёт количество знаков после запятой, где count — количество знаков после десятичной точки	<pre>cout << fixed << setprecision(3) << (13.5 / 2) << endl;</pre>	6.750
setfill(int symbol)	Установить символ заполнитель. Если ширина поля больше, чем выводимая величина, то свободные места поля будут наполняться символом symbol — символ заполнитель	<pre>cout << setfill('0') << setw(4) << 15 << ends << endl;</pre>	0015

59. Что такое родовой класс и какова его основная форма?

ЭТО ТО ЖЕ САМОЕ, ЧТО И ШАБЛОН КЛАССА, СМ. БИЛЕТ 72

60. Что происходит, когда открытые члены базового класса наследуются как открытые? Что происходит, когда они наследуются как закрытые?

Когда открытые члены базового класса наследуются как открытые, они наследуются без изменений и остаются открытыми, когда они наследуются как закрытые, они становятся закрытыми.

61. Чисто виртуальные функции и абстрактные классы.

Чисто виртуальная (абстрактная) функция – функция, которая не имеет тела. Это заполнитель, который должен быть переопределен производными классами

Синтаксис:

```
// чисто виртуальная функция
virtual int getValue() const = 0;
```

Любой класс, где есть 1+ абстрактная функция становится абстрактным классом. Это значит, что нельзя создать его экземпляр (действительно, если бы у нас был экземпляр такого класса, это значило бы, что можно вызывать функцию, не имеющую реализации)

Кроме того, производные классы должны переопределять тело этой функции, иначе производный класс тоже будет считаться абстрактным

АХТУНГ:

Можно определять чисто виртуальные функции, у которых есть тело

```
3  class Animal // Animal - абстрактный базовый класс
4  {
5  protected:
6      std::string m_name;
7
8  public:
9      Animal(const std::string& name)
10         : m_name{ name }
11     {
12     }
13
14     std::string getName() { return m_name; }
15
16     // = 0 означает, что эта функция чисто виртуальная
17     virtual const char* speak() const = 0;
18
19     virtual ~Animal() = default;
20 };
```

```
2  const char* Animal::speak() const // даже если у нее есть тело
3  {
4      return "buzz";
5  }
```

Что важно: При предоставлении тела для чисто виртуальной функции тело должно быть предоставлено отдельно.

Эта парадигма может быть полезна, когда вы хотите, чтобы ваш базовый класс предоставлял для функции реализацию по умолчанию, но при этом заставлял любые производные классы предоставлять свои собственные реализации. Однако, если производный класс удовлетворен реализацией по умолчанию, предоставленной базовым классом, он может просто вызвать реализацию базового класса напрямую. То есть мы переопределяем абстрактную функцию в наследнике, но в переопределении вызываем реализацию метода из родителя:

```
class Dragonfly: public Animal
{
public:
    Dragonfly(const std::string& name)
        : Animal{name}
    {
    }

    // этот класс больше не является абстрактным,
    // потому что мы определили эту функцию
    const char* speak() const override
    {
        // использовать реализацию по умолчанию в Animal
        return Animal::speak();
    }
};
```

62. Что такое родовая функция и какова ее основная форма?

ЭТО ТО ЖЕ САМОЕ, ЧТО И ШАБЛОН ФУНКЦИИ. СМ БИЛЕТ 73

63. Что такое встраиваемая функция? В чем ее преимущества и недостатки?

Основная идея в том, чтобы ускорить программу ценой занимаемого места. После того как вы определите встроенную функцию с помощью ключевого слова `inline`, всякий раз когда вы будете вызывать эту функцию, компилятор будет заменять вызов функции фактическим кодом из функции.

Вроде как у каждого компилятора есть свои эвристические приемы, которые помогают определить, какую функцию он будет встраивать а какую нет. Не встраиваются рекурсии (или встраивается их кусочек, а остальное заменяется вызовом

функции), большие функции, функции с операторами ввода вывода и функции с циклами (что спорно, да)

Встроенные функции очень хороши для ускорения программы, но если вы используете их слишком часто или с большими функциями, у вас будет чрезвычайно большая программа. Иногда большие программы менее эффективны, и поэтому они будут работать медленнее, чем раньше. Встроенные функции лучше всего подходят для небольших функций, которые часто вызываются.

```
1 #include <iostream>
2
3 using namespace std;
4
5 inline void hello()
6 {
7     cout<<"hello";
8 }
9 int main()
10 {
11     hello(); //Call it like a normal function...
12     cin.get();
13 }
```

Встроенные функции обеспечивают следующие преимущества:

1. Затраты на вызов функции не происходят.
2. Затрат времени на возврат не происходит.

Недостатки встроенной функции:

1. Если они слишком большие и вызываются слишком часто, объем ваших программ сильно возрастает. Из-за этого применение встраиваемых функций обычно ограничивается короткими функциями.
2. Встроенная функция может увеличить накладные расходы времени компиляции, если кто-то изменит код внутри встроенной функции, тогда все вызывающие места должны быть перекомпилированы, потому что компилятору потребуется заменить весь код еще раз, чтобы отразить изменения.

64. Чем действие дружественной оператор-функции отличается от действия оператор-функции — члена класса?

Немного теории:

Операторы можно перегружать в двух вариантах:

как функцию-член

как свободную (не-член) функцию.

Четыре оператора можно перегрузить только как функцию-член — это =, ->, [], (). Для того, чтобы перегрузить оператор как

функцию-член необходимо объявить нестатическую (то есть принадлежащую не классу а объекту) функцию-член с именем `operator@`, где @ символ(ы) оператора.

В случае перегрузки унарного оператора эта функция не должна иметь параметров, а в случае бинарного должна иметь ровно один параметр.

Для того, чтобы перегрузить оператор как свободную (не-член) функцию, необходимо объявить функцию с именем `operator@`, где @ символ(ы) оператора.

В случае перегрузки унарного оператора, эта функция должна иметь один параметр, а в случае бинарного должна иметь два параметра. В случае перегрузки бинарного оператора – по крайней мере один из двух параметров, а в случае унарного единственный параметр должен быть того же типа (или типа ссылки), что и тип, для которого реализуется перегрузка.

```
N::X x, y;  
// инфиксная форма  
N::X z = x + y;  
N::X v = x - y;  
N::X w = +x;  
N::X u = -x;  
x(1,2);  
char p = x[4];  
// функциональная форма  
N::X z = x.operator+(y);  
N::X v = operator-(x, y);  
N::X w = x.operator+();  
N::X u = operator-(x);  
x.operator()(1,2);  
char p = x.operator[] (4);
```

А теперь собственно ответ на вопрос:

Дружественная оператор-функция имеет доступ к закрытым полям класса, поэтому, например, можно не писать геттеры для полей, использующихся при операциях

Если оператор перегружен для некоторого класса X с использованием свободных функций, то этот оператор

автоматически становится перегруженным для любого класса, имеющего неявное преобразование к X. Это справедливо как для унарных, так и для бинарных операторов.

Звучит очень умно но абсолютно непонятно. Надеюсь, кто-то разберется в этом вместо меня

Одна из причин по которой для бинарных операторов свободные функции могут оказаться предпочтительными — это симметрия. Часто желательно, чтобы если корректным выражением является $x@y$, то корректным выражением было бы и $y@x$ для любых допустимых типов. Для свободных функций мы можем выбирать произвольный тип первого операнда, когда как в случае функции-члена мы этого лишены. В качестве примера можно привести оператор `+` для `std::string`, когда один из операндов имеет тип `const char*`.

Перегрузка бинарных операторов с использованием свободных функций позволяет расширять интерфейс класса без добавления новых функций-членов. (Напомним, что интерфейс класса включает не только функции-члены, но и свободные функции с параметрами тип которых определяется этим классом.) В качестве примера можно привести перегрузку операторов вставки и извлечения из потока. Если бы мы для перегрузки этих операторов использовали функции-члены, то нам бы пришлось для каждого нового типа, вставляемого в поток или извлекаемого из потока, добавлять в потоковые классы соответствующие функции-члены, что понятное дело невозможно.

65. Что такое объект?

Объект — то, что может быть индивидуально рассмотрено и описано (лекции Путуридзе)

Объект — прототип класса, которому могут быть заданы свойства и у которого можно вызывать методы

66. Что происходит с защищенным членом класса, когда класс наследуется как открытый? Что происходит, когда он наследуется как закрытый?

как открытый: остается защищенным

как закрытый: становится закрытым

67. Управление доступом к элементам класса.

Спецификатор **public** является спецификатором доступа (access specifier), то есть определяет параметры доступа к членам класса – переменным и функциям. В частности, он делает их доступными из любой части программы.

С помощью спецификатора **private** мы можем скрыть реализацию членов класса, то есть сделать их закрытыми, сделать поля или методы видимыми только внутри самого класса (или внутри дружественной функции). Элементы класса с этим модификатором доступа не наследуются.

Спецификатор **protected** запрещает обращение к полю/методу извне класса (исключение – друж функция), однако эти элементы класса будут наследоваться.

68. Указатели и ссылки на объект.

Ну э. Мы можем объявить указатель на объект (или получить его, от оператора new), можем объявить ссылку (**S& s**). Потом через указатель и ссылку можем обращаться к членам класса через **->** для указателя и **.** для ссылки.

69. Указатель на объект производного класса

Указатели на базовый и производный классы связаны друг с другом, чего мы не наблюдаем для указателей других типов. Как правило, указатель одного типа не может указывать на объект другого типа. Однако указатели базовых и производных классов выпадают из этого правила. В C++ указатель базового класса может быть использован в качестве указателя на объект любого другого класса, производного от этого базового. Предположив например, что у нас есть базовый класс B и производный от него класс D. Любой указатель, объявленный как указатель на B, можно использовать для указания на объект типа D. Таким образом, если имеются объявления:
B *p; // указатель на объект типа B

B B_ob; // объект типа B

D D_ob; // объект типа D

то следующие предложения вполне правильны:

p = &B_ob; // p указывает на объект типа B

p = &D_ob; /* p указывает на объект типа D,

который является производным от B. */

Базовый указатель можно использовать только для обращения к тем частям производного объекта, которые наследованы от базового класса. Так, в этом примере p можно использовать для обращения ко всем элементам D_ob, унаследованным от B_ob. Однако к элементам специфическим для D_ob, посредством p обратиться нельзя

70. Управление доступом при наследовании.

Различают 3 спецификатора доступа при наследовании:

Спецификатор доступа в базовом классе	Спецификатор доступа при открытом наследовании
<code>public</code>	<code>public</code>
<code>protected</code>	<code>protected</code>
<code>private</code>	не доступен

Открытое наследование (`public` - наследование) - открытые поля класса остаются открытыми, защищенные - защищенными, `private` поля не доступны.

Защищенное наследование - открытые поля и защищенные - защищенные, закрытые поля - закрытые

Закрытое наследование - все поля становятся `private`

71. Указатель `this`.

Указатель `this` - указатель на объект класса, для которого выполняется метод

Указатель - переменная, в которой хранится адрес объекта.

Таким образом, указатель `this` хранит адрес объекта класса, для которого выполняется метод (функция-член класса)

72. Шаблон класса.

По аналогии обобщенной функцией определяются обобщенные классы. Для этого создается класс, в котором определяются все используемые им алгоритмы, при этом реальный тип обрабатываемых в нем данных будет задан как параметр при создании объектов этого класса.

Компилятор автоматически сгенерирует корректный тип объекта на основе типа, заданного при создании объекта.

Синтаксис:

```
template <class Type> class «имя класса»
```

где `Type` представляет имя типа, который будет задан при создании объекта класса, используя следующий формат:

```
«имя класса» < тип > «имя объекта»
```

```

// member_function_templates1.cpp
template<class T, int i> class MyStack
{
    T* pStack;
    T StackBuffer[i];
    static const int cItems = i * sizeof(T);
public:
    MyStack( void );
    void push( const T item );
    T& pop( void );
};

template< class T, int i > MyStack< T, i >::MyStack( void )
{
};

template< class T, int i > void MyStack< T, i >::push( const T item )
{
};

template< class T, int i > T& MyStack< T, i >::pop( void )
{
};

int main()
{
}

```

73. Шаблон функции.

В связи с тем, что в С++ статическая типизация данных (прием, при котором переменная, связывается с **типом** в момент объявления и **тип** не может быть изменен позже), то есть необходимость порой для каждого типа данных реализовывать свою функцию, однако вне зависимости от типов данных сама по себе функция делает одно и то же

Пример:

```

5  string gt(string a, string b) {
6      return a + b; //конкатенация строк
7  }
8
9  double gt(double a, double b) {
10     return a + b; //сложение двух чисел
11 }

```

Для того, чтобы не страдать такой хуйней как введение ручками перегруженных функций, придумали шаблоны функций

Шаблоны функций -- это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов. Другими словами, шаблон функции (шаблонная функция, обобщенная функция) представляет собой семейство разных функций (или описание алгоритма). По описанию шаблон функции похож на

обычную функцию: разница в том, что некоторые элементы не определены (типы, константы) и являются параметризованными. Все шаблоны функций начинаются со слова `template`, после которого идут угловые скобки, в которых перечисляется список параметров. Каждому параметру должно предшествовать зарезервированное слово `class` или `typename`.

Синтаксис:

```
template<typename <название типа> >
<название типа> <имя функции>(<параметры>)
```

```
template<typename T>
T max(T a, T b){
    return a > b ? a : b;
}
```

Ключевое слово `typename` говорит о том, что в шаблоне будет использоваться встроенный тип данных, такой как: `int`, `double`, `float`, `char` и т. д. А ключевое слово `class` сообщает компилятору, что в шаблоне функции в качестве параметра будут использоваться пользовательские типы данных, то есть классы. Когда выполняется вызов функции, компилятор анализирует параметр шаблонизированной функции и создает экземпляр для соответствующего типа данных: `int`, `char` и так далее.

74. Почему следующие две перегруженные функции внутренне неоднозначны?

`int f (int a);` - принимает значение (чиллит в своей зоне видимости)
`int f (int & a);` - принимает ссылку

75. Почему следующая функция может не компилироваться как встраиваемая?

```
void fl ( ) {
    int i;
    for( i = 0; i < 10; i++ ) cout << i;
}
```

Наличие цикла и вывода (в реальности цикл тоже может встроиться, но Путуридзе лучше так не говорить).

76. Что неправильно в данном фрагменте?

```
int main ( ) {  
    . . . .  
    throw 12.23;  
}
```

throw допустим только в try

77. Что неправильно в следующем прототипе функции?

```
char * f ( char * p, int x = 0, char * q );
```

Функция со значением по умолчанию должна идти в самом конце, иначе ошибка компиляции

78. Что неправильно в конструкторе, показанном в следующем фрагменте?

```
class sample {  
    double a, b, c;  
public:  
    double sample ( );  
}
```

Ему не нужно указывать возвращаемое значение. (не нужно, у конструктора в принципе типа нет)

79. Правильен ли следующий фрагмент?

```
union {  
    float f;  
    unsigned int bits;  
}
```

Путур, верти точку с запятой в после union`а !

Ну и еще юнионы живут только в теле функций (по крайней мере анонимные)

80. Какой будет результат после отработки данной программы?

```

1  #include <iostream>
2  #include <list>
3  using namespace std;
4  int main ( ) {
5      list < char > lst;
6      list < char > :: iterator it_lst;
7
8      int i;
9
10     for ( i = 0; i < 10; i ++ ) lst.push_back ( 'A' + i );
11     cout << "Size = " << lst.size ( ) << endl;
12
13     cout << "lst: ";
14     while ( ! lst.empty ( ) ) {
15         it_lst = lst.end ( );
16         it_lst --;
17         cout << * it_lst;
18         lst.pop_back ( );
19     }
20     return 0;
21 }

```

```

Size = 10
lst: JIHGFEDCBA

```

81. Какой будет результат после отработки данной программы?

```

1  #include <iostream>
2  using namespace std;
3  class static_func_demo {
4      static int i;
5  public:
6      static void init ( int x ) { i = x; }
7      void show ( ) { cout << i; }
8  };
9
10 int static_func_demo :: i;
11
12 int main (){
13     static_func_demo :: init ( 100 );
14     static_func_demo x;
15     x.show ( );
16     return 0;
17 }

```

ВЫВОД: 100

82. Какой будет результат после отработки данной программы?

```

1  #include <iostream>
2  using namespace std;
3  class samp {
4      int i;
5  public:
6      samp ( int n ) { i = n; }
7      void set_i ( int n ) { i = n; }
8      int get_i ( ) { return i; }
9  };
10 void sqr_it ( samp ob ) {
11     ob.set_i ( ob.get_i ( ) * ob.get_i ( ) ) ;
12     cout << ob.get_i() << "\n";
13 }
14 int main ( ) {
15     samp a ( 10 );
16     sqr_it ( a );
17     cout << a.get_i ( );
18     return 0;
19 }

```



```

100
10

```

ВЫВОД:

Объект копируется, поэтому значение в a неизменно.

83. Исследуйте следующую конструкцию:

```

#include <iostream>

using namespace std;

class cl_base {
    int a, b;
public:
    int c;
    void setab ( int i, int j ) { a = i; b = j; }
    void getab ( int & i, int & j ) { i = a; j = b; }
};

class derived_1 : public cl_base { };
class derived_2 : private cl_base { };

int main ( ) {
    derived_1 ob_1;
    derived_2 ob_2;
    int i, j;

    // . . . . .
    return 0;
}

```

Какая из следующих инструкций правильна внутри функции main ()?

- A. ob_1.getab (i, j);
- B. ob_2.getab (i, j);
- C. ob_1.c = 10;
- D. ob_2.c = 10;

C, так как они наследуются как публик, в отличие от остальных. А вообще public, а не pablic, так что оно не скомпилируется вообще.

А, так как ошибку компилятор не выдаст, но выводить будет мусор (дискуссионный момент, зависит от постановки вопроса)

84. Объясните, что в следующей программе неправильно, и исправьте ее.


```

#include <iostream>

using namespace std;

class cl_1 {
    int * p;
public:
    cl_1 ( int i );
    ~cl_1 ( ) { delete p; }
    friend int getval ( cl_1 ob );
};

cl_1 :: cl_1 ( int i ) {
    p = new int;
    if ( ! p ) {
        cout << "Error 1\n";
        exit ( 1 );
    }
    * p = i;
}

int getval ( cl_1 ob ) { return * ob.p; }

int main ( ) {
    cl_1 a ( 1 ), b ( 2 );

    cout << getval ( a ) << " " << getval ( b );
    cout << "\n";
    cout << getval ( a ) << " " << getval ( b ) ;

    return 0 ;
}

```

```

class cl_1 {
    int * p;
public:
    cl_1 (const cl_1&);
    cl_1 ( int i );
    ~cl_1 ( ) { delete p; }
    friend int getval ( cl_1 ob );
};

cl_1::cl_1 (const cl_1& ref) {
    p = new int;
    *p = *(ref.p);
}

```

В чем идея: при передаче объекта по значению, то есть копированием, скопируются все поля, в том числе и указатель. После отработки дружественной функции копия объекта удаляется, тем самым очищается память, на которую ссылается указатель (убивая значения, валяющиеся в а и b). То есть после первого вызова указатели в объектах а и b становятся мусорными.

Как это решается: выделяем новый кусок памяти, закидываем туда значение, хранящееся по указателю в переданном объекте

85. Какой будет результат после отработки данной программы?

```

#include <iostream>

using namespace std;

class A {
public: A ( ) { cout << "Constructor A\n"; }
      ~A ( ) { cout << "Destructor A\n"; }
};

class B {
public:
      B ( ) { cout << "Constructor B\n"; }
      ~B ( ) { cout << "Destructor B\n"; }
};

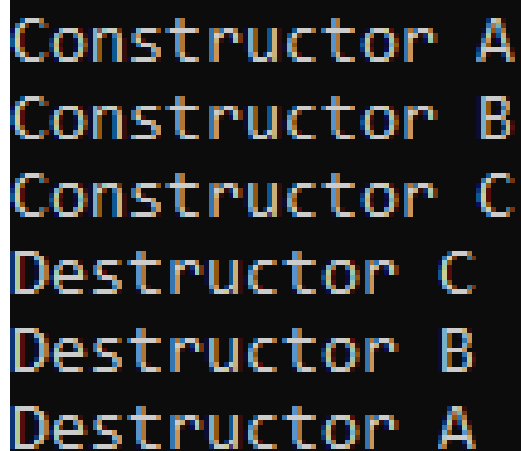
class C : public A, public B {
public:
      C ( ) { cout << "Constructor C\n"; }
      ~C ( ) { cout << "Destructor C\n"; }
};

int main ( ) {
      C ob;

      return 0;
}

```

ВЫВОД:



```

Constructor A
Constructor B
Constructor C
Destructor C
Destructor B
Destructor A

```

86. В следующей программе имеется ошибка. Исправьте ее с помощью оператора `const_cast`.

```

1  #include <iostream>
2  using namespace std;
3
4  void f ( const double & i ){
5      const_cast<double &>(i) = 100;
6  }
7
8  int main (){
9      double x = 98.6;
10     cout << x << endl;
11     f ( x );
12     cout << x << endl;
13     return 0;
14 }

```

Идея в том, что мы пытаемся изменить константную переменную, что ну как бэ нельзя. А преобразование `const_cast` делает переменную неконстантной

87. Ниже приведены две перегруженные функции. Покажите, как получить адрес каждой из них.

```

1  #include <iostream>
2  using namespace std;
3
4  int dif ( int a, int b ) { return a - b; }
5  float dif ( float a, float b ) { return a - b; }
6
7  int main ( ) {
8      int (*func1) (int a, int b);
9      func1 = dif; // первая перегрузка
10
11     float (*func2) (float a, float b);
12     func2 = dif; // первая перегрузка
13
14     return 0;
15 }
16

```

88. Какой будет результат после отработки данной программы?

```

#include <iostream>
using namespace std;
union bits {
    bits ( short n );
    void show_bits ( ) ;
    short d;
    unsigned char c [ sizeof ( short ) ];
};
bits :: bits ( short n ) { d = n; }
void bits :: show_bits ( ) {
    int i, j;
    for ( j = sizeof ( short ) - 1; j >= 0; j -- ) {
        cout << "Byte " << j << " : ";
        for( i = 128; i; i >>= 1 )
            if ( i & c [ j ] ) cout << "1";
            else cout << "0";
        cout << "\n";
    }
}
int main ( ) {
    bits ob ( 5 );
    ob.show_bits ( );
    return 0;
}

```

Byte 1:00000000

Byte 0:00000101

но вопрос я в рот ебал, да

А я знаю, как это решается, но вам не скажу

89. Какой будет результат после отработки данной программы?

```

#include <iostream>
using namespace std;
template < class T >
T & inc_value ( T & val ) {
    ++val;
    return val;
}
int main ( )
{
    int x = 64;
    char c = 64;
    x = ( int ) inc_value < int > ( x );
    cout << x << endl;
    c = ( char ) inc_value < char > ( c );
    cout << c << endl;
    printf ( "%02X", c );
    return 0;
}

```

А это как вообще решать

65 //просто увеличили число на 1

А //65 в ASCII это 65

41 //65 в 16-ричном формате

это говно не надо решать. оно не решается

90. Какой будет результат после отработки данной программы?

```
#include <iostream>
using namespace std;
int main ( ) {
    union {
        unsigned char bytes [ 4 ];
        int value;
    };
    int i;
    value = 128;
    for ( i = 3; i >= 0; i -- )
        cout << ( int ) bytes [ i ] << " ";
    return 0;
}
```

0 0 0 128

Union дает нам памяти по размеру наибольшего члена - четыре байта. Первая ячейка забивается 128, остальные нулями. Потом с конца выводим, явно приводя к int

91. Какой будет результат после отработки данной программы?

```

#include <iostream>

using namespace std;

class samp {
    int i;
public:
    samp ( int n ) { i = n; }
    void set_i ( int n ) { i = n; }
    int get_i ( ) { return i; }
};

void sqr_it ( samp * ob ) {

    ob -> set_i ( ob -> get_i ( ) * ob -> get_i ( ) );

    cout << ob -> get_i ( ) << "\n";
}

int main ( ) {
    samp a ( 10 );

    sqr_it ( & a );
    cout << a.get_i ( );

    return 0;
}

```

100

100, ТК ПЕРЕДАЕТСЯ ПО ССЫЛКЕ

92. Какой будет результат после отработки данной программы?

тут вопросик короче по поводу того, что последний символ чаровского массива тупа не используется

Он используется и выводится пустой символ

Array a:

1 2 3 4 5

Array b:

1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array c:

HELLO

93. Дана следующая программа, переделайте все соответствующие обращения к членам класса так, чтобы в них явно присутствовал указатель `this`.

```
1  #include <iostream>
2  using namespace std;
3  class cl_1 {
4      int a, b;
5  public:
6      cl_1 ( int n, int m ) { this->a = n; this->b = m; }
7      int add ( ) { return this->a + this->b; }
8      void show ( );
9  };
10 void cl_1 :: show ( ) {
11     int t;
12     t = this->add ( );
13     cout << t << "\n";
14 }
15 int main() {
16     cl_1 ob ( 10, 14 );
17     ob.show ( ) ;
18     return 0;
19 }
20
```

94. Какой будет результат после отработки данной программы?


```

#include <iostream>

using namespace std;

class base {
public:
    base ( ) { cout << "Constructor base\n"; }
    ~base ( ) { cout << "Destructor base\n"; }
};

class derived : public base {
public:
    derived ( ) { cout << "Constructor derived\n"; }
    ~derived ( ) { cout << "Destructor derived\n"; }
};

int main ( ) {

    derived ob;

    return 0;
}

```

```

Constructor base
Constructor derived
Destructor derived
Destructor base

```

95. Что неправильно в следующей программе?

```
#include <iostream>

using namespace std;

void triple ( double & num );

int main ( ) {
    double d = 7.0;

    triple ( & d );
    cout << d;
```

```
    return 0;
}
```

Есть только объявление но нет определения функции. Кроме того, значение принимается по ссылке, не нужно брать адрес.

96. Какой будет результат после отработки данной программы?

```

#include <iostream>

using namespace std;

union swapbytes {
    unsigned char  c [ 2 ];
    unsigned short i;

public:
    swapbytes ( unsigned short x );
    void swp ( );
};

swapbytes :: swapbytes ( unsigned short x ) { i = x; }

void swapbytes :: swp ( ) {
    unsigned char temp;
    temp      = c [ 0 ];
    c [ 0 ] = c [ 1 ];
    c [ 1 ] = temp;
}

int main ( ) {
    swapbytes ob ( 1 );
    ob.swp ( );
    cout << ob.i;
    return 0;
}
Short: 00000000 00000000
Char: 00000000

i = 1: 00000000 00000001
поменяли значения в массиве чаров, память для i выглядит теперь так: 00000001 00000000

2 в 8 это 256.
256

```

97. Какой будет результат после отработки данной программы?


```

1  #include <iostream>
2  using namespace std;
3  class cl_1 {
4  public:
5      static int i;
6      void seti ( int n ) { i = n; }
7      int geti ( ) { return i; }
8  };
9  int cl_1 :: i;
10 int main ( ) {
11     cl_1 o1, o2;
12     cl_1 :: i = 100;
13     cout << "o1.i: " << o1.geti ( ) << '\n';
14     cout << "o2.i: " << o2.geti ( ) << '\n';
15     return 0;
16 }
17

```

```

o1.i: 100
o2.i: 100

```

мы увеличиваем публичное статик поле, усе.

99. Какой будет результат после отработки данной программы?

```

#include <iostream>

using namespace std;

int rotate ( int i ) {
    int x;
    if ( i & 0x80000000 ) x = 1;
    else                  x = 0;
    i = i << 1;
    i += x;
    return i;
}

int main ( ) {
    int a;
    a = 0x80000000;
    cout << rotate ( a );
    return 0;
}

```

вывод: 1

Почему ?

0x80000000=10000000000000000000000000000000

0x80000000 и 0x80000000 побитово дает истину, так как в двоичном представлении чисел 32-й бит равен 1. Потом сдвигаем 0x80000000 на 1 влево, крайний единичный бит теряется, справа дописывается ноль и получается число, полностью состоящее из нулей, прибавляем 1, получаем 1.

100. Какой будет результат после отработки данной программы?

```

#include <iostream>
using namespace std;
union swapbytes {
    unsigned char c [ 2 ];
    unsigned short i;
public:
    swapbytes ( unsigned short x );
    void swp ( );
};
swapbytes :: swapbytes ( unsigned short x ) { i = x; }
void swapbytes :: swp ( ) {
    unsigned char temp;
    temp = c [ 0 ];
    c [ 0 ] = c [ 1 ];
    c [ 1 ] = temp;
}
int main ( ) {

```

```
swapbytes ob ( 256 );  
ob.swp ( );  
cout << ob.i;  
return 0;  
}
```

Задача обратная билету 96, ответ, следовательно 1

101. Какой будет результат после отработки данной программы?

```
#include <iostream>  
using namespace std;  
int main() {  
    unsigned char c = 0x1A;  
    c <<= 4; // умножить на 16  
    c >>= 4; // разделить на 16  
    printf ( "c = %02x", c );  
    return 0;  
}
```

Вывод: c = 0A

102. Какой будет результат после отработки данной программы?

```

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

class BaseClass {
    virtual void f ( ) { }
};

class Derived1 : public BaseClass { };
class Derived2 : public BaseClass { };

int main ( ) {
    int i ;
    BaseClass * p,   baseob;
    Derived1   ob1;
    Derived2   ob2;

    cout << "Type - " << typeid ( i ).name ( ) << endl;
    p = & baseob;
    cout << "Type - " << typeid ( * p ).name ( ) << endl;
    p = & ob1;
    cout << "Type - " << typeid ( * p ).name ( ) << endl;
    p = & ob2;
    cout << "Type - " << typeid ( * p ).name ( ) << endl;

    return 0;
}

```

```

Type - i
Type - 9BaseClass
Type - 8Derived1
Type - 8Derived2

```

У стандартным типом, кажись, только первая буква типа пишется

103. Какой будет результат после отработки данной программы?

dynamic_cast работает только вверх по иерархии, но никак не вниз


```

#include <iostream>
using namespace std;
class Base {
public:
    virtual void f ( ) { }
};
class Derived: public Base {
public:
    void derived_only ( ) {
        cout << "It's object of class Derived\n";
    }
};
int main ( ) {
    Base * bp, b_ob;
    Derived * dp, d_ob;
    bp = & b_ob;
    dp = dynamic_cast < Derived * > ( bp );
    if ( dp ) dp -> derived_only ( );
    else cout << "Error 1\n";
    bp = & d_ob;
    dp = dynamic_cast < Derived * > ( bp );
    if ( dp ) dp -> derived_only ( );
    else cout << "Error 2\n";
    return 0;
}

```

Вывод:

Error 1

It's object of class Derived

104. Что неправильно в следующем фрагменте

```
#include <iostream>
using namespace std;
class cl1 {
    int i , j;
```

```
public:
    cl1 ( int a, int b ) { i = a; j = b; }
};
class cl2 {
    int i, j;
public:
    cl2 ( int a, int b ) { i = a; j = b; }
};

int main ( ) {
    cl1 x ( 10, 20 );
    cl2 y ( 0, 0 );
    x = y;
    . . . . .
}
```

Присваиваем друг другу объекты, которые не состоят в одной иерархии.