@BOTVA_ITS6

Подготовка к экзамену

Объектно-ориентрованное программирование

Над файлом работали:
dimopster, fiixii, pluttan

Оглавление

1. Структура программы на C++. Пример простейшей программы. Препроцессор.

Структура программы на C++ состоит из нескольких частей:

1. **Директивы препроцессора** – директивы, начинающиеся со знака `#` , которые позволяют подключить библиотеки, определить макросы или изменить параметры компиляции. Например:
   - `#include <iostream>` (директива подключает стандартную библиотеку `iostream` для ввода-вывода)
   - `#define PI 3.1415` (определение макроса для дальнейшего использования в программе)
2. **Объявление пространства имен**: директива `using` или полное указание пространства имен для использования стандартных функций и объектов. Например:
   - `using namespace std` (объявление пространства имен `std` для использования функций и объектов стандартной библиотеки).

Главная функция `main()` : основная функция, которая выполняет все вычисления и формирует выходные данные. По стандартам эта функция должная быть прописана на самом верху, а при наличии других функций следует прописать прототипы этих функций и поместить в конец программы. При запуске программы, управление автоматически передается в функцию `main()` .

Пример простейшей программы:

```
int main(void){
    cout << “Hello world”! << endl; // выводим Hello world через потоки вводавывода
    return 0;                       // возвращаем 0 в качестве кода завершения программы
}
```

Препроцессор - это инструмент C++, который работает до компиляции кода и занимается обработкой директив препроцессора. Препроцессор выполняет подстановку макросов, подключает заголовочные файлы, удаление комментариев и т.д.

2. Скалярные типы данных C++. Определение констант и переменных. Пример.

Константа - это именованное значение, которое *нельзя изменить* после того, как оно было определено. В C++, чтобы определить константу, используется ключевое слово `const` .
Например:

```
const int MAX_VALUE = 100;
const double PI = 3.14159;
```

(по стандартам программирования константы записываются *заглавными буквами*).
Сначала указывается специальное слово `const` , а после него *тип переменной*, по которой будет хранить некоторое неизменяемое значение.
Переменная - это именованное значение, которое *может изменяться* в процессе выполнения программы. Чтобы определить переменную в C++, нужно указать ее *тип*, а затем *имя*.
Например:

```
int count = 0;
double price = 9.99;
```

Определение символьной переменной:

```
char letter = 'a';
```

Определение логической переменной:

```
bool is_finished = false; (true)
```

Типы данных в C++ В языке C++ есть несколько скалярных типов данных, которые представляют простые значения без составных элементов. Эти типы данных включают в себя *целочисленные типы*, *вещественные типы*, *символьный тип* и *логический тип*.
Целочисленные типы данных:

- `int` целочисленное значение – 4 байта во внутренней памяти компьютера
- `long` длинное целочисленное значение – 4/8 байтов в зависимости от компилятора – чаще 4 байта
- `long long` длинное целочисленное значение – 8 байт в памяти
- `short` короткое целочисленное значение – 2 байта
- `signed` знаковый тип – от конкретных отрицательных значений до конкретных положительных
- `unsigned` беззнаковый тип – значения от 0 и до положительных, умноженных на 2

Пример `signed<тип>` и `unsigned<тип>` (для примера взял `int`): `signed int` – от $-2.147.483.648$ и до $2.147.483.647$
`unsigned int` – от 0 и до $4.294.967.295$
Вещественные типы данных:

- `float` (вещественное значение с одинарной точностью – 6/7 знаков после запятой)

- `double` (вещественное значение с двойной точностью – 15/16 знаков после запятой)

Символьные тип данных:

- `char` (символ)
- `wchar_t` (представляет *“расширенные”* символы, которые могут быть представлены в формате Юникод)

Логический тип данных:

- `bool` (логическое значение: `true` или `false`)

3. Операции над скалярными данными C++. Приоритеты операций. Примеры.

В C++ скалярные данные представляются переменными, которые содержат только одно значение. Примерами скалярных данных являются типы `int` , `double` , `float` , `bool` , `char` и другие (см. 2 вопрос о типах данных).
Основные операции над переменными

- **Арифметические:**
    1. *Сложение*.
    2. *Вычитание*.
    3. *Умножение*.
    4. *Деление* (результат – вещественное число, если хотя бы одно из чисел *вещественное*; результат – целое число, если делимое и делитель – целые числа).
- **Логические:**
    1. `!` не
    2. `&&` и
    3. `||` или
- **Логические поразрядные:**
    1. ~ не
    2. `&` и
    3. `|` или
    4. ^ исключающее или
- **Отношения:**
    1. < меньше
    2. > больше
    3. <= меньше или равно
    4. >= больше или равно
    5. == равно
    6. != не равно
- **Сдвиги:**
    1. Сдвиг_вправо = Операнд >> Операнд
    2. Сдвиг_влево = Операнд << Операнд
- **Порядковые:**
    1. *Инкремент* (увеличение значения на 1) Следующее = (++ Операнд | Операнд ++)

2. *Декремент* (уменьшение значения на 1) Предыдущее = (`--` Операнд | `--` Операнд)

Ниже представлены основные операции над скалярными данными в порядке *приоритета* от наивысшего к наименьшему (некоторые операции выполняются раньше других):

1. `() [] -> :: .`
2. (`!` (не)) (`+` и `-` (*унарные знаки* – для изменения знака самого числа)) (инкремент/декремент (`++` и `--`)) (адрес (`&`)) (указатель (`*`)) `sizeof` , `new` , `delete` (специальные слова)
3. `.*` , `->*`
4. `*` , `/` , `%`
5. `+` , `-` (*бинарные* – для сложения и вычитания)
6. `<<` , `>>`
7. `<` , `<=` , `>` , `>=`
8. ((`==`) сравнение на равенство) (`!=` (сравнение на неравенство))
9. `&` (поразрядное и)
10. `^` (исключающее или)
11. `|` (поразрядное или)
12. `&&`
13. `||`
14. `?:` (тернарный оператор)
15. `=` , `*=` , `/=` , `%=` , `+=` , `-=` , `&=` , `^=` , `|=` , `<<=` , `>>=` (синтаксический сахар)
16. `,`

Примеры выражений:

```
int a = 10, b = 3, c = 1;
float division = a/b;       // division = 3

b = 0;
b = c++;                    // b = 1, c = 2
b = ++c;                    // b = 3, c = 3

a += b;                     // Альтернатива: a = a + b

c = (a = 5, b = a*a)        // Альтернатива: a = 5; b = a * a; c = b
```

4. Управляющие операторы if и switch C++. Примеры.

Операторы `if` и `switch` в языке программирования C++ используются для управления потоком выполнения программы *в зависимости от условия*.

Оператор условной передачи управления (if)

```
if Выражениеусловие(/) Оператор [else Оператор]`
```

Пример:

```
int a = 5;
if (a > 0) {
    cout << "a is positive" << endl;     //Вывод информации, что a > 0
} else if (a < 0) {
    cout << "a is negative" << endl;     //Проверка дополнительным условием)
                                         //Вывод информации, что a < 0
} else {
    cout << "a is zero" << endl;         // Если оба условия не выполняются тогда a = 0
}
```

Оператор выбора (switch)

```
Switch Выражение()
{
    {case Элемент : Оператор{}}
    [default : Оператор{}]
}
```

Пример:

```
int x = 2;
switch (x) {
    case 1:     //Если 'x' = 1, то:
        cout << "x is one" << endl;
        break;
    case 2:     //Если 'x' = 2, то:
        cout << "x is two" << endl;
        break;
    default:    //Если не выполнился ни один case
        cout << "x is neither one nor two" << endl;
        break;
}
```

(Если не написать оператор `break` в конструкции `switch` , то выполнение программы продолжится до тех пор, пока не встретится оператор `break` или конец конструкции)

`default` выполняется в том случае, если *ни один* из блоков *case* не выполняется.

5. Организация циклов в C++. Примеры.

Организация циклов в C++ осуществляется с помощью следующих конструкций:

Цикл while (цикл с предусловием)

```
while (if) {
    // код, который нужно повторять
}
```

Пример:

```
int i = 0;
while (i < 5) {
    cout << i << " ";
    i++;
}
```

(Программа выводит значения `i` от 0 до 4, т.к. цикл выполняется *5 раз*, а начало происходит с *0*)

В данном цикле, до тех пор, *пока условие истинно*, будет выполняться код, который находится *внутри* фигурных скобок.

Цикл do-while (цикл с постусловием)

```
do {
    // код, который нужно повторять
} while (if);
```

Пример:

```
int j = 0;
do {
    cout << j << " ";
    j++;
} while (j < 5);
```

(Программа также выводит значения *j от 0 до 4*)

В данном цикле сначала выполняется код *внутри фигурных скобок*, а затем *проверяется условие*. Если оно истинно, то цикл продолжается. Цикл `do-while` гарантирует, что внутренний блок кода будет выполнен *хотя бы один раз.*

Цикл for (итерационный цикл)

```
for (init; if; step) {
    // код, который нужно повторять
}
```

Пример:

```
for (int k = 0; k < 5; k++) {
    cout << k << " ";
}
```

(Программа также выводит значения *k от 0 до 4*)

В данном цикле в первом выражении происходит *инициализация*, второе выражение — *условие*, которое проверяется перед каждой итерацией цикла, а третье выражение — *шаг*, который выполняется после каждой итерации.

(В данном цикле мы *заранее знаем количечестко итераций*, которые надо выполнить)

Выражения в описании цикла можно не указывать

Например:

```
int k = 0;
for (;;k++) {
    cout << k << " ";
}
```

(В данном примере получаем бесконечный цикл)

6. Неструктурные операторы передачи управления в C++. Пример.

Неструктурные операторы передачи управления - это операторы, которые могут *изменить порядок выполнения программы*. Они могут использоваться для организации условных переходов и циклов, а также для *передачи управления* в другие части программы.
Оператор `goto` - это оператор безусловного перехода *на метку* в программе.
Пример:

```
int a = 1;
if (a == 1) {
    goto skip;
}
cout << "Переход не выполнен" << endl;
skip: cout << "Переход выполнен" << endl;
```

(Переход к метке с именем `skip`, т.к. `a == 1`, значит *условие выполнилось*)
Т.е. если значение переменной `a` равно 1, то выполнение программы переходит на метку `skip`, иначе выводится сообщение *"Переход не выполнен"*.
Оператор `break` - это оператор, который *прерывает выполнение цикла*
Например:

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        break;
    }
    cout << i << endl;
}
```

(В этом примере, если значение переменной `i == 3`, то *выполнение цикла прекращается* и программа выводит только числа от *0 до 2*)
Оператор `continue` - это оператор, который пропускает *текущую итерацию цикла* и переходит к следующей
Например:

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;
    }
    cout << i << endl;
}
```

(В этом примере, если значение переменной `i == 3`, то выполнение *текущей* итерации цикла *пропускается*, и программа выводит только числа *от 0 до 4, кроме 3*)

7. Указатели и ссылки. Примеры объявлений.

Указатели и ссылки - это специальные типы данных, которые используются для работы *с памятью* в программе.
Указатели - это переменные, которые хранят *адреса памяти* в компьютере. Они позволяют управлять данными в памяти. Операции с указателями включают в себя *разыменование* (получение конкретного значения по адресу в памяти), *получение адреса* и *выполнение арифметических операций с указателями*.
Ссылки - это “псевдонимы” переменных, которые используются для работы с ними, используя их *истинные значения*, а не копии. Операции с ссылками включают в себя *получение ссылки на переменную* и *выполнение разыменования ссылки*.
Пример программы, показывающей все аспекты, описанные выше:

```
int main(void) {
    int a = 5;
    int* ptr1;                    // объявление указателя на int
    ptr1 = &a;                    // присваивание указателю адреса переменной a
    *ptr1 = 10;                   // разыменование указателя и присваивание значения 10 переменной a
    cout << a << endl;            // выводится 10

    int& ref1 = a;                // объявление ссылки на int, присваивание ссылке переменной a
    ref1 = 20;                    // изменение значения переменной a через ссылку
    cout << a << endl;            // выводится 20

    const int& ref2 = a;          // объявление константной ссылки на int, присваивание ссылке
    переменной a

    int b = 15;
    int* ptr2 = &b;               // объявление указателя на int, присваивание указателю адреса
    переменной b
    ptr1 = ptr2;                  // присваивание указателю ptr1 адреса переменной b
    *ptr1 = 25;                   // изменение значения переменной b через указатель ptr1
    cout << b << endl;            // выводится 25

    int c = 30;
    int& ref3 = c;                // объявление ссылки на int, присваивание ссылке переменной c
    ref1 = ref3;                  // присваивание через ссылку ref1 значения переменной c
    cout << a << endl;            // выводится 30

    return 0;
}
```

8. Управление динамической памятью C++. Примеры.

В C++ управление *динамической памятью* осуществляется с помощью операторов `new` и `delete` .

Оператор `new` используется для *выделения блока памяти* заданного размера, который можно использовать для *хранения данных*.

Пример использования оператора `new` :

```
int* ptr = new int;        // выделение блока памяти для переменной типа int
*ptr = 42;                 // сохранение значения в выделенный блок памяти
```

Оператор `delete` используется для *освобождения памяти*, которая была выделена с помощью оператора `new` .

Пример использования оператора `delete` :

```
delete ptr;         // освобождение выделенной памяти
```

В данном примере мы *освобождаем блок памяти*, на который указывает *указатель ptr*.

Ещё один пример использования операторов `new` и `delete` :

```
int* arr = new int[10];         // выделение блока памяти для массива из 10 элементов
for(int i = 0; i < 10; i++) {
    arr[i] = i;
}
delete[] arr;                   // освобождение выделенной памяти стоит( указывать квадратные скобки,
     чтобы удалить все элементы массива, а не только первый).
```

9. Адресная арифметика C++. Примеры.

Адресная арифметика используется для работы *с указателями* на элементы массива или других структур данных. Она позволяет выполнять операции с адресами памяти, такие как *сложение*, *вычитание*, *инкремент* и *декремент*.
Пример использования адресной арифметики (инкремент и сложение):

```
int arr[] = {1, 2, 3, 4, 5};                    //создаем массив на несколько элементов
int* ptr = &arr[0];                              // получаем адрес первого элемента массива
cout << "Первый элемент: " << *ptr << endl;      // выводим первый элемент
ptr++;                                           // увеличиваем указатель на 1
cout << "Второй элемент: " << *ptr << endl;      // выводим второй элемент
ptr += 2;                                        // увеличиваем указатель на 2
cout << "Четвертый элемент: " << *ptr << endl;   // выводим четвертый элемент
```

Аналогично происходит работа с *декрементом* и *вычитанием*.
Еще один пример использования адресной арифметики (структура):

```
struct Person {
    string name;                // Поле "имя"
    int age;                    // Поле "возраст"
};

Person* arr = new Person[3];    // выделяем память для 3 объектов структуры Person
arr[0] = {"John", 25};          // инициализируем первый объект структуры
arr[1] = {"Sara", 23};          // инициализируем второй объект структуры
arr[2] = {"Jim", 30};            // инициализируем третий объект структуры

Person* ptr = arr;              // получаем адрес первого элемента массива
for(int i = 0; i < 3; i++) {
    cout << "Имя: "
         << ptr->name
         << ", Возраст: "
         << ptr->age << endl;
    ptr++;                      // переходим к следующему элементу
}

delete[] arr;                   // освобождаем выделенную память
```

Мы *выделяем память* (оператор `new`) для 3 объектов структуры `Person` и заполняем их данными. Затем мы получаем *адрес первого элемента* с помощью указателя и выводим с помощью цикла `for` *все три элемента* на экран, обращаясь к полям структуры через указатель и инкрементируя его в конце итерации. Затем мы *освобождаем выделенную память* (с помощью `delete[] arr`).

10. Массивы C++. Примеры объявлений и две технологии обработки. Пример.

Массивы - это структуры данных, которые позволяют хранить набор элементов *одного типа*. Индексация массивов начинается с *0*. Многомерные массивы в памяти расположены *построчно*.

Пример объявления массива:

```cpp
int numbers[5];    // объявление массива из 5 элементов типа int
float prices[10];  // объявление массива из 10 элементов типа float
char letters[26];  // объявление массива из 26 элементов типа char
```

Цикл for() / цикл foreach (цикл по коллекции) (это все один метод обработки массивов)

Обычный цикл `for()` :

```cpp
int numbers[5] = {1, 2, 3, 4, 5};   // объявление массива и инициализация его элементов
for (int i = 0; i < 5; i++) {       // цикл для обработки массива
    cout << numbers[i] << " ";      // вывод элементов массива на экран
}
```

Цикл `foreach` (цикл по коллекции):

```cpp
// Auto - автоопределение типа
// Ссылка (&) для изменения чисел в массиве
for (auto &i : numbers) {
    i *= 2;                 // Увеличение каждого элемента массива в 2 раза
    cout << i << endl;      // Вывод этих значений
}
```

Адресная арифметика

```cpp
int numbers[5] = {1, 2, 3, 4, 5};     // Объявление массива и инициализация его элементов
int* p = numbers;                     // Объявление указателя на первый элемент массива
for (int i = 0; i < 5; i++) {         // Цикл для обработки массива
    cout << *p << " ";                // Вывод элементов массива на экран
    p++;                              // Переход к следующему элементу массива
}
```

Пример работы с массивом (сумма элементов массива)

```cpp
int numbers[5] = {1, 2, 3, 4, 5};                    // Объявление массива и инициализация его
    элементов
int sum {0};                                         // Переменная для хранения суммы элементов
    массива
for (int i = 0; i < 5; i++)                          // Цикл для обработки массива
    sum += numbers[i];                               // Добавление элемента массива к сумме
cout << "Сумма элементов массива: " << sum << endl;  // Вывод результата на экран
```

11. Строки C++. Стандартные функции, работающие со строками. Примеры.

Строка в c++ - это *массив символов*, заканчивающийся нулевым символом. Строки в C++ представлены классом `std::string` . Для работы со строками в стандартной библиотеке есть *множество функций*.
Некоторые из них:
`getline` - функция для *считывания строки* из входного потока.
Пример:

```
#include <iostream>
#include <string>

int main() {
    string s;
    getline(cin, s);    // Ввод строки `s` через getline()
    cout << "Вы ввели: " << s << endl;
    return 0;
}
```

`strlen` - функция для *вычисления длины строки*.
Пример:

```
int main() {
    char s[] = "Hello, world!";
    cout << "Длина строки: " << strlen(s) << endl;
    return 0;
}
```

`strcpy` - функция для *копирования строки*.
Пример:

```
int main() {
    char s1[] = "Hello";
    char s2[10];
    strcpy(s2, s1);        // s1 - что копируем, s2 - куда копируем
    cout << s2 << endl;
    return 0;
}
```

`strcat` - функция для *объединения двух строк*.
Пример:

```
int main() {
    char s1[] = "Hello";
    char s2[] = " world!";
    strcat(s1, s2);             // конкатенация 2 строк
                                // запись полной строки производится в первый параметр strcat()
                                // поэтому выводим `s1`
    cout << s1 << endl;
    return 0;
}
```

`substr` - функция для *извлечения подстроки* из *строки*.
Пример:

```
int main() {
    string s = "Hello, world!";
    string sub = s.substr(7, 5);     // 7 - индекс, с которого будем получать подстроку
                                     // 5 - количество символов, которые мы скопируем
    cout << sub << endl;
    return 0;
}
```

find - функция для *поиска подстроки* в *строке*.

Пример:

```
int main() {
    string s = "Hello, world!";
    if (s.find("world") != '\0') {  //Если слово world найдено до символа конца строки \0
        cout << "Найдено" << endl;
    } else {
        cout << "Не найдено" << endl;
    }
    return 0;
}
```

Можно было написать другую реализацию: `!= string::npos`, *т.е.*

```
if (s.find("world") != string::npos)
```

12. Структурный тип C++. Пример.

Структурный тип данных позволяет объединять несколько переменных *разных типов* в одну структуру. Структурный тип данных описывается специальным словом `struct` , внутри которого описываются поля любого *скалярного типа данных*.
Пример объявления структуры:

```
struct Student {
    int group;          // Поле группа
    string name;        // Поле имя
    int age;            // Поле возраст
    float weight;       // Поле вес
};
```

Структуру можно создать и через пользовательский тип:

```
typedef struct {
    int group;              // Поле группа
        string name;        // Поле имя
        int age;            // Поле возраст
        float weight;       // Поле вес
} Student;                  // Имя структуры указывается в конце объявления
```

В данном примере создается структура `Student` , которая содержит четыре переменные разных типов: `group` (целое число), `name` (строка), `age` (целое число) и `weight` (вещественное число).
Чтобы создать объект нашей структуры, следует в основной программе написать тип `Student` перед идентификатором объекта
Пример:

```
int main(void){
    Student student;    // Создали объект student структуры Student
    ...
}
```

Чтобы обратиться к полям* созданного объекта, следует написать следующую конструкцию через *дот-нотацию**

```
int main(void){
    Student student;
    student.group = 10;     // Обращение к полю group объекта student
    student.name = “”John;  // Обращение к полю name объекта student
    student.age = 19;       // Обращение к полю age объекта student
    student.weight = 74.1;  // Обращение к полю weight объекта student
    return 0;
}
```

Обращение к полям можно осуществить и через указатель на структуру**

```
int main()
{
    Student *st = new Student;
    st->group = 10;
    st->name = ”John”;
    st->age = 19;
    st->weight = 74.1;
    delete st;
}
```

13. Функции C++. Передача параметров и возвращение результатов. Примеры.

Функция - это блок кода, который выполняет *определенную задачу*. Функции могут принимать параметры и возвращать результаты. Параметры в функцию могут передаваться *по значению* или *по ссылке*.

Передача параметров по значению означает, что в функцию передается *копия значения параметра*. При этом изменение значения параметра внутри функции *не повлияет* на значение переменной в вызывающей функции.

Пример:

```
void increment(int x) {
    x++;
}

int main() {
    int num = 5;
    increment(num);
    cout << num << endl; // выведет 5, тк значение переменной num не изменилось
    return 0;
}
```

Передача параметров по ссылке означает, что в функцию передается *ссылка на переменную*, а *не ее копия*. При этом изменение значения параметра внутри функции *повлияет* на значение переменной в вызывающей функции (передача по ссылке *осуществляется с помощью* знака `&` в объявлении функции).

Пример:

```
void increment(int& x) {
    x++;
}

int main() {
    int num = 5;
    increment(num);
    cout << num << endl; // выведет 6, тк значение переменной num было изменено внутри функции
    return 0;
}
```

Процедура – это *ФУНКЦИЯ*, которая *не возвращает никакого значения*. Перед объявлением процедуры надо написать тип возвращаемого значения – `void` (*пустота*). Процедура не может возвращать значения, но она может *печатать* значение через `cout` и менять параметры, переданные ей по ссылке.

Пример:

```
void Print()
{
    cout << "Hello world!" << endl;
}
int main()
{
    Print();
    return 0;
}
```

14. Параметры-массивы. Пример.

Параметры-массивы - это механизм передачи *массивов в функцию* в качестве *параметров*. Они позволяют передавать массивы *различных типов данных* и *размеров* в функцию, где они могут быть обработаны в соответствии с требованиями программы.

Пример:

```
void printArray(int arr[], int size) {   //Передаем массив как параметр и колво элементов
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr1[] = {1, 2, 3, 4, 5};
    int arr2[] = {10, 20, 30, 40, 50, 60, 70};

    printArray(arr1, 5);    // Печатаем первый массив
    printArray(arr2, 7);    // Печатаем второй массив

    return 0;
}
```

В C++ отсутствует контроль размера массива по первому индексу! Это означает, что если мы передаем *многомерный массив* в функцию, то функция должна знать *размерность массива по первому индексу*, чтобы правильно обрабатывать его элементы.

Пример:

```
void printArray(int arr[][4], int rows) {    //rows - отвечает за первый индекс массива
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 4; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}
```

15. Параметры-строки. Пример.

Параметры-строки - это механизм *передачи строк* в функцию в качестве *параметров*. Они позволяют передавать строки *любых размеров* в функцию, где они могут быть обработаны в соответствии с требованиями программы.

Пример:

```
string RemoveSpaces(string str)      // Функция, которая удаляет пробелы в строке
                                     // string str - передаваемый параметр строка()
{
    string result = "";
    for (int i = 0; i < str.length(); i++)
    {
        char ch = str[i];
        if (ch != ' ')
        {
            result += ch;
        }
    }
    return result;
}


int main()
{
    string str1 = "Hello, world!";
    string str2 = "This is a test string.";

    cout << RemoveSpaces(str1) << endl;     // Передаем строку str1 в функцию
    cout << RemoveSpaces(str2) << endl;     // Передаем строку str2 в функцию

    return 0;
}
```

16. Параметры структуры. Пример.

Параметры-структуры - это механизм передачи *объектов структур* в функцию или процедуру (структура может состоять из нескольких полей *разных типов*).
Для реализации этого механизма необходимо указать в качестве типа параметра идентификатор структуры.
Пример (передача по значению):

```
struct Person {                          //Описание структуры, в которой 3 поля
  string name;
  int age;
  float height;
};

void printPerson(Person p) {             //Передаем структуру типа Person в процедуру
  cout << "Name: " << p.name << endl;
  cout << "Age: " << p.age << endl;
  cout << "Height: " << p.height << endl;
}

int main(void){
    Person john;                         // Создание объекта структуры с именем john + прямая
    запись в поля объекта
    john.name = "John Smith";
    john.age = 35;
    john.height = 1.80;

    printPerson(john);                   // Вызов процедуры с передачей объекта
}
```

Пример (с передачей по ссылке):

```
struct Array                             // Описание структуры с одним полем - массив из 10
    элементов
{
    int mas[10];
};

void printArr(Array &a)                  // Передаем объект Array по ссылке, чтобы работать
    истинными элементами массива a( не копиями)
{
    for (int i = 0; i < 10; i++)
    {
        a.mas[i] = i + 2;                // Увеличиваем каждый элемент на 2
    }
}

int main()
{
    Array a;                             // Создаем объект структуры
    for (int i = 0; i < 10; i++)
    {
        a.mas[i] = i;                    // Инициализируем элементы массива
    }
    for (int i = 0; i < 10; i++)
    {
        cout << a.mas[i] << " ";         // Выводим элементы
    }
    cout << endl;

```

```
27      printArr(a);                            // Вызываем функцию, которая изменит значения массива
28      for (int i = 0; i < 10; i++)
29      {
30          cout << a.mas[i] << " ";            // Выводим измененные значения
31      }
32      cout << endl;
33      return 0;
34  }
```

17. Классы памяти переменных. Примеры.

В C++ существует несколько *классов памяти* переменных, в зависимости от того, как они *создаются* и *используются*.
Автоматическая память (`stack`) - это память, которая *выделяется и освобождается автоматически* при вызове и выходе из функции. Переменные (*локальные*), созданные внутри функции, находятся в *автоматической памяти*.
Пример:

```
void function() {
    int x = 10;            // переменная x находится в автоматической памяти
}
```

Статическая память (`static`) - это память, которая выделяется при *запуске программы* и освобождается только при *завершении* работы программы. Переменные, созданные с помощью ключевого слова `static`, находятся в *статической памяти*.
Пример:

```
void function() {
    static int x = 10;     // переменная x находится в статической памяти
}
```

Существует класс *внешних переменных* (`extern`). Они используются для доступа к переменным, *определенным в других файлах*. Данные переменные, объявленные как внешние, могут быть использованы сразу в нескольких файлах *без необходимости его повторного определения*.
Пример:

```
Файл
 "file1.cpp":

int globalVar = 10;          // определение глобальной переменной

int main() {

                             // Использование глобальной переменной

  return 0;
}Файл

 "file2.cpp":

extern int globalVar;        // объявление внешней переменной, полученной из "file1.cpp"

int main() {

                             // Описание внутри функции

  return 0;
}
```

Эти переменные можно использовать совместно со `static`:

```
extern static int b;
```

18. Параметры-функции. Пример.

Параметры-функции - это механизм передачи функции в другую функцию в качестве ее параметра.
Пример:

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

int add(int x, int y) { return x + y; }             // функция сложения двух чисел

int multiply(int x, int y) { return x * y; }        // функция умножения двух чисел

int invoke(int x, int y, int (*func)(int, int))     // функция, которая получает указатель на
    функцию
{
    return func(x, y);
}

int main()
{
    cout << "Addition of 20 and 10 is ";            // передаем ссылку на функцию add
    cout << invoke(20, 10, &add) << '\n';

    cout << "Multiplication of 20"
        << " and 10 is ";                           // передаем ссылку на функцию multiply
    cout << invoke(20, 10, &multiply) << '\n';

    return 0;
}
```

(в качестве последнего параметра функции `invoke` используется указатель на функцию. В него можно записать адрес функции указанного шаблона. В нашем случае, функция должна возвращать значение типа `int` , и принимать в качестве параметров два значения типа `int`)

19. Правила, определяющие видимость переменных в функциях. Пример.

В C++ *видимость переменных* в функциях определяется *блоками кода*. Переменные, объявленные *внутри блока кода*, могут быть использованы только *в этом блоке* и во внутренних блоках. Если переменная *объявлена* внутри функции, то она видна во всех блоках кода функции, но *невидима* за пределами функции.
Пример:

```
void myFunction() {
    int x = 5;
    cout << "x inside function: " << x << endl;

    {
        int y = 10;
        cout << "y inside inner block: " << y << endl; // x и y видны тут

    }

                                                      // y не виден здесь, за пределами
    внутреннего блока
                                                      // x виден тут
}

int main() {
    myFunction();
                                                      // x и y не видны тут, за пределами функции
    return 0;
}
```

Если снаружи функции прописана *глобальная переменная*, которая имеет *такой же идентификатор*, что и переменная *внутри функции*, то переменная, находящаяся *внутри функции*, будет *иметь приоритет* (локальная переменная *всегда перекрывает глобальную*).
Пример:

```
int x = 10;                                       // Глобальная переменная

void myFunction() {
    int x = 5;                                    // Локальная переменная
    cout << "x inside function: " << x << endl;   // Выведется 5
}

int main() {
    myFunction();
    cout << "x outside function: " << x << endl;  // Выведется 10
    return 0;
}
```

20. Пространства имен. Пример.

Пространство имен (`namespace`) в C++ - это *механизм*, который позволяет *группировать сущности* (*переменные*, *функции*, *классы* и т.д.) под *общим именем*, чтобы избежать конфликтов имен и упростить организацию кода.
Пример:

```
namespace MyNamespace {
    int x = 5;

    void myFunction() {
        cout << "Hello from MyNamespace" << endl;
    }
}

int main() {
    cout << MyNamespace::x << endl;
    MyNamespace::myFunction();
    return 0;
}
```

В этом примере создается *пространство имен* `MyNamespace`, в котором определены переменная `x` и функция `myFunction`. Переменная `x` и функция `myFunction` могут быть использованы *только внутри* пространства имен `MyNamespace`. Для доступа к этим сущностям вне пространства имен, нужно использовать *оператор разрешения контекста* `::`.
Пространства имен могут быть *вложенными друг в друга*. *Пример:*

```
namespace MyNamespace {
    namespace InnerNamespace {
        int x = 5;

        void myFunction() {
            cout << "Hello from InnerNamespace!" << endl;
        }
    }
}

int main() {
    cout << MyNamespace::InnerNamespace::x << endl;
    MyNamespace::InnerNamespace::myFunction();
    return 0;
}
```

21. Компоновка модулей C++. Защита от повторной компиляции. Пример.

Компоновка модулей - это процесс *разделения программы на отдельные файлы*, каждый из которых содержит определение *классов*, *функций* или *переменных*. Эти файлы могут быть скомпилированы отдельно и затем *объединены в единую программу*.
Защита от повторной компиляции - это механизм, который предотвращает *компиляцию одного и того же файла более одного раза*. Это важно, потому что если файл скомпилирован дважды, это может *привести к ошибкам* компоновки и неправильному поведению программы.
Пример:

```
//    Файл "foo."h:

#ifndef FOO_H
#define FOO_H

void foo();

#endif

//    Файл "foo.C"++:

#include "foo.h"

void foo() {
    cout << "Hello, world!" << endl;
}

//    Файл "main.C"++:

#include "foo.h"

int main() {
    foo();
    return 0;
}
```

Директива `#ifndef` проверяет, определен ли *макрос с именем* `FOO_H` . Если *макрос не определен*, то код между `#ifndef` и `#endif` *будет выполнен*, и *макрос* `FOO_H` *будет определен* с помощью директивы `#define` . Таким образом, *при следующем включении файла* `foo.h` в программу, *макрос* `FOO_H` уже *будет определен*, и код между `#ifndef` и `#endif` будет *пропущен*.

22. Перегрузка функций. Пример.

Перегрузка функций - это возможность определить *несколько функций с одинаковым именем*, но разными *параметрами*. Компилятор на основе переданных аргументов выбирает нужную функцию для вызова. Это удобно для упрощения кода и улучшения его читаемости.
Пример перегрузки функций:

```
// Прототипы функций
void print(int x);
void print(double x);
void print(char x);

int main()
{
    // В зависимости от того, чем мы инициализируем функцию, та функция и вызовется
    print(5);    // Значит вызвалась 1 функция
    print(3.14); // Значит вызвалась 2 функция
    print('a');  // Значит вызвалась 3 функция
    return 0;
}

void print(int x)    // 1 функция с идентификатором print()
{
    cout << "Integer value: " << x << endl;
}

void print(double x) // 2 функция с идентификатором print()
{
    cout << "Float value: " << x << endl;
}

void print(char x)   // 3 функция с идентификатором print()
{
    cout << "Character value: " << x << endl;
}
```

23. Функции с параметрами по умолчанию. Пример.

Функции с параметрами по умолчанию позволяют определить значения аргументов функции, которые будут *использоваться*, если при вызове функции *эти аргументы не указаны*.
Пример:

```
int addNumbers(int x, int y = 10) {
    return x + y;
}

int main() {
    cout << addNumbers(5) << endl;          // Вывод: 15 используется( значение y по умолчанию = 10)
    cout << addNumbers(5, 20) << endl;  // Вывод: 25 используется( инициализированное значение = 20)
    return 0;
}
```

В этом примере функция `addNumbers` принимает *два параметра*: `x` и `y` . Параметр `y` имеет значение по умолчанию, равное 10. Если при вызове функции *не указывается значение* для `y` , то используется значение *по умолчанию*.

24. Текстовые файлы. Отличие от Delphi Pascal. Пример.

Текстовый файл - это файл, содержащий текст, хранящийся на некотором накопителе в виде именованной последовательности байтов.
Работа с *текстовыми файлами* в C++ осуществляется с помощью *потоков ввода-вывода*.
Для работы с текстовыми файлами используются *классы* `fstream`, `ifstream` и `ofstream`, которые определены в *заголовочном файле* `fstream`.
`ifstream` - переход *от файла в программу* (открытие файла для чтения)
`ofstream` - переход *от программы к файлу* (открытие файла для записи)
В отличие от *Delphi Pascal*, в C++ текстовые файлы обрабатываются с помощью *потокового ввода-вывода*, а не с помощью *процедур чтения и записи*.
Пример открытия файла для записи:

```
ofstream foutF("F1.txt");          // Инициализируем открытие файла
char ch;
if (foutF.is_open())               // Если файл открыт, то вводим символы пока не напишут 0
{
    while (cin >> ch && ch != '0')
    {
        foutF << ch;               // Потоковый ввод в файл
    }
}
foutF.close();                     // Обязательное закрытие файла
```

Файл *всегда стоит закрывать* после работы. Если этого *не сделать*, то может пойти *учечка записанной информации* и некоторые данные могут *остаться в буфере* и *не переписаться* в наш файл.
Пример открытия файла для чтения:

```
ifstream finF_print("F1.txt");       // Инициализируем файловую переменную и сам файл для чтения
    данных
if (finF_print.is_open())            // Если файл открыт
{
    while (finF_print.get(ch))       // Цикл, который выполняется до тех пор, пока метод get()
    возвращает ненулевое значение
    {
        cout << ch;                  // Вывод в терминал
    }
}
finF_print.close();                  // Обязательное закрытие файла
```

25. Двоичные файлы. Отличие от Delphi Pascal. Пример.

Двоичные файлы в C++ и Delphi Pascal используются для хранения данных в *бинарном формате*. Однако, есть некоторые *отличия*:

1. В C++ для работы с *двоичными файлами* используются классы `fstream`, `ifstream` и `ofstream`, которые позволяют открывать файлы в различных режимах (*чтение*, *запись*, *добавление*) и производить операции *чтения и записи данных*.
2. В C++ при записи и чтении данных в бинарных файлах используется функция `write()` для записи и функция `read()` для чтения (В Delphi Pascal это `BlockRead()` и `BlockWrite()`).

Пример записи в двоичный файл:

```
int main() {
    string str = "Hello, world!";

    ofstream file("ex.bin", ios::binary);        // ios::binary - компонента, показывающая, что файл будет двоичным.
    if (file.is_open()) {
        file.write(str.c_str(), str.size());     // c_str() - это метод, передающий указатель на первый символ строки str; str.size() - размер строки str в байтах
    }
    file.close();
    return 0;
}
```

Пример чтения из файла:

```
int main()
{
    int arr[5];
    ifstream file("data.bin", ios::binary);
    file.read((char*)arr, sizeof(arr));         //Считываем данные из data.bin в массив из пяти элементов первый параметр - указатель на буфер, второй - размер одного элемента массива в байтах
    for (int i = 0; i < 5; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    file.close();
    return 0;
}
```

В данном случае данные будут *записаны в файл в бинарном формате* – каждый символ будет иметь свой *ASCII-код*. Каждый символ будет занимать *один байт памяти*. Вот как будет выглядеть эта строка в шестнадцатеричном представлении (*"Hello, world!"*):

`48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21`

Здесь каждый *двухзначный блок* представляет *один байт*. Например, первый блок "48" соответствует символу 'H' в *ASCII-кодировке*.

26. Определение класса, компоненты класса. Ограничение доступа. Пример.

Класс – это *пользовательский тип данных*, который может содержать данные (*поля/атрибуты*) и функции (*методы*) для работы с этими данными. Класс определяется ключевым словом `class` с последующим указанием имени класса и определением его компонентов (*полей и методов*) внутри фигурных скобок.
Компоненты класса – это непосредственно *поля* и *методы* класса. Обращение к ним допускается с использованием специальных слов, именуемых *“модификаторами доступа”*.
Ограничение доступа зависит от того, какой *модификатор доступа* мы используем внутри класса для описания его характеристик. *Всего их 3*:

1. `Private` – доступ только *внутри класса* (закрытый модификатор)
2. `Protected` – доступ только *внутри класса* и его *наследников* (защищенный модификатор)
3. `Public` – доступно *извне класса* (открытый модификатор), а также в *основной программе* (короче в любом месте программы).

Если не указать модификатор доступа в классе (*определяющий доступ метода/поля*), то он по умолчанию будет `private` .
При *наследовании*, если не указать *тип наследования*, он по умолчанию будет `protected` .
Пример:

```
class Person
{
public:
    string name;                                // Открытый доступ
    int age;
private:
    string password;                            // Закрытый доступ
protected:
    string address;                             // Защищенный доступ
};

int main(void)
{
    Person p1;                                  // Создание объекта
    p1.name = ”John”;                           // Открытый доступ
    p1.age = 30;                                // Открытый доступ
    cout << p1.age << ” ” << p1.name << endl;   // Вывод значений
    p1.password = ”123”;                        // Ошибка компиляции - закрытый доступ
    p1.address = ”123 Main St”;                 // Ошибка компиляции - защищенный доступ
    return 0;
}
```

27. Инициализация полей при отсутствии конструктора. Пример.

Инициализация полей класса осуществляется при помощи метода: *конкструктора* или *инициализирущего метода*.

Пример программы с инициализирущим методом:

```
//      1 Способ использование константного указателя базового класса (this)
class A
{
public:
    void Init(int Num, string Str)      //Инициализирующий метод
    {
        this->Num = Num;                // Инициализируем поле
        this->Str = Str;                // Инициализируем поле
    }

    void Print()                        // Метод, который напечатает поля
    {
        cout << Num << " " << Str << endl;
    }

    int Num;                            // Поле класса типа int
    string Str;                         // Поле класса типа string
};

int main()
{
    setlocale(LC_ALL, "ru");

    A a;                                // Создаем объект класса A
    a.Init(10, "Privet");               // Вызываем инициализирующий метод
    a.Print();                          // Выводим поля

    return 0;
}

//       2 способ использование стандартного присваивания полей класса

class B
{
public:
    void Init(int n, string s)          // Инициализирующий метод
    {
        Num = n;                        // Инициализируем поле
        Str = s;                        //Инициализируем поле
    }

    void Print()                        // Метод, который напечает поля
    {
        cout << Num << " " << Str << endl;
    }

    int Num;
    string Str;
};

int main()
{
    setlocale(LC_ALL, "ru");

```

```
55      B b;                                    // Создаем объект класса B
56      b.Init(10, "Privet");                   // Вызываем инициализирующий метод
57      b.Print();                              // Выводим поля
58
59      return 0;
60  }
```

Абсолютно *все поля и методы* 2-х классов находятся в `public` , значит через *объект* класса и *дот-нотацию* мы можем обратиться к полям *напрямую*. Следует *инкапсулировать* поля класса (сделать их *невидимыми* в основной программе) через `private` , чтобы нельзя было "*затереть*" наши *инициализированные данные*.
Есть 3-й пример *инициализации полей* через *геттеры* и *сеттеры*, но ГС про него *не говорила*, поэтому достаточно знать 2 способа.

28. Конструкторы. Инициализация полей при наличии конструктора. Пример.

Конструктор - это метод класса, который вызывается *автоматически* при создании объекта.
Конструкторов можем быть *сколько угодно* в классе, а вот *деструктор* - *один*.
Всего выделяют несколько *видов конструкторов*, которые выполняют *определенные функции*:

1. **Конструктор по умолчанию** - метод класса, который вызывается, когда *создается объект*.
2. **Конструктор с параметрами** - метод класса, который вызывается, когда *создается объект* класса с последующим *инициализированием параметров*.

```
class A
{
public:
    A() { cout << "Const" << endl; } // конструктор по умолчанию создается если любой следующий
    конструктор не будет инициализирован
    A(int a)                         // инициализирующий конструктор с параметром
    {
        this->a = a;                 // this константный указатель
    }

    int a;
};

int main(void)
{
    A a1;                            // создание объекта конструтор по умолчанию
    A a2(10);                        // создание объекта конструтор с параметром
}
```

Остальные конструкторы будут рассмотрены в следующих билетах.

29. Деструкторы. Пример.

Деструтор в C++ - это специальный метод класса, который *автоматически вызывается* при *уничтожении объекта класса*. *Деструктор* выполняет *очистку всех ресурсов*, которые были выделены объекту во время его жизни, включая память, выделенную динамически оператором `new` .
Деструктор имеет свое обозначение, которое начинается с символа (*тильда*), за которым следует *имя класса*. Он *не принимает аргументов* и *не возвращает значения*.
Деструктор вызывается, если конструирование объекта завершено!
Деструкторы тоже имеют несколько вариаций: 1. **Обычный деструктор** - осуществляет *основную очистку памяти* и ресурсов при *уничтожении объекта*.

```
class MyClass {
public:
    ~MyClass() {
        // освобождение ресурсов
    }
};
```

1. **Виртуальный деструктор** - используется в классах с *наследованием*, чтобы гарантировать правильную работу деструкторов при удалении объектов из *динамической памяти*. Виртуальный деструктор должен быть объявлен в *базовом классе*, чтобы все деструкторы *классов наследников* вызывали его *корректно*.

```
class Base {
public:
    virtual ~Base() {
        // освобождение ресурсов
    }
};

class Inherited : public Base {
public:
    ~Derived() {
        // освобождение ресурсов
    }
};
```

1. **«Пустой» деструктор** - используется для значительного сокращения объема кода, когда объект *не требует освобождения ресурсов*. (Данный деструктор используется в классах со *статическими полями*).

```
class MyClass {
public:
    ~MyClass() {}
};
```

Пример кода с деструтором:

```
class A
{
public:
    A() { cout << "Const" << endl; } // конструктор по умолчанию создается если любой следующий
    конструктор не будет инициализирован
```

```
5     A(int a)                          // инициализирующий конструктор
6     {
7         this->a = a;                  // this константный указатель
8     }
9
10    void Print()
11    {
12        cout << a << endl;
13    }
14
15    ~A()                               // деструктор по умолчанию надо его прописывать если есть
      конструктор даже если нет динамических полей
16    {
17        cout << "Destruktor enabled" << endl;
18    }
19
20    int a;
21};
22
23int main(void)
24{
25    A a(10);                           // создание объекта
26    a.Print();
27}
```

30. Инициализация полей объектов при наличии и отсутствии конструктора.

Примеры *инициализации полей* с *конструктором* и без него рассмотрены в билетах *27* и *28*. К тем примерам добавляется “*прямая запись*” в поле через *объект класса*. *Пример:*

```
class MyClass {
public:
    int age;                                // Поле, доступное в основной программе
    string name;                            // Поле, доступное в основной программе
};

int main() {
    MyClass obj;                            // Создаем объект

    obj.age = 10;                           // Прямая запись в поле
    obj.name = ”John”;                      // Прямая запись в поле

    cout << ”Age: ” << obj.age << endl;     // Выводим
    cout << ”Name: ” << obj.name << endl;
    return 0;
}
```

31. Простое и множественное наследование классов. Пример.

Простое наследование - это *механизм*, при котором *класс-наследник* конструируется из `public` и `protected` полей и методов *базового класса* (*от которого наследуется*), с возможностью добавления своих полей и методов. С помощью простого наследования класс может наследовать *все свойства базового класса*, включая его *поля* и *методы*. При простом наследовании у *класса-наследника* только один *родительский класс*.

Пример простого наследования:

```
class A
{
protected:                      // Поля доступны в классе наследнике
    int x;
    int y;

public:                         // Поля доступны в любом месте программы
    A() {}                      // Конструтор по умолчанию
    A(int x, int y)             // Инициализирующий конструтор
    {
        this->x = x;
        this->y = y;
    }
    int Sum()                   // Функция, возвращающая сумму x и y
    {
        return x + y;
    }
};

class B : public A              // Класс B наследуется от класса A
                                // про модицикаторы доступа при наследовании см. 32 билет
{
public:
    int z;                      // Добавляем еще одно поле
    B() {}                      // Конструтор по умолчанию
    B(int x, int y, int z) : A(x, y)
                                // Инициализирующий конструтор с вызовом инициализирующиего
    конструтора класса A с передачей параметров x и y
    {
        this->z = z;
    }
    int Mult()                  // Функция произведения 3 полей
    {
        return x * y * z;
    }
};

int main()
{
    setlocale(LC_ALL, "ru");

    A a(4, 10);                  // Объект класса A
    B b(4, 10, 7);               // Объект класса B

    cout << b.Sum() << endl;  // Вызов функции суммы
    cout << b.Mult() << endl; // Вызов функции произведения

    return 0;
}
```

Здесь класс `B` наследуется от класса `A` , следовательно, в классе `B` присутствуют все поля и методы класса `A` , и его собственные.

Множественное наследование - это *механизм конструирования* класса, при котором класс *наследует* свойства сразу *нескольких базовых классов*.

Пример множественного наследования:

```
class Car       // Класс машина
{
protected:
    Car()
    {
        cout << "I can drive!" << endl;
    }
};

class Plane     // Класс самолет
{
protected:
    Plane()
    {
        cout << "I can fly!" << endl;
    }
};

class FlyingCar : public Car, public Plane
// Класс летающая машина наследует поля и методы Car и Plane
{
public:
    FlyingCar() : Car(), Plane()
// Конструтор класса наследника + 2 конструтора базовых классов инициализация
    {
        cout << "I can drive & fly!" << endl;
    }
};

int main(void)
{
    setlocale(LC_ALL, "ru");

    FlyingCar fl;       // Создаем объект FlyingCar вызывается его конструктор
                        // Вывод: I can drive!
                        //        I can fly!
                        //        I can drive & fly!

    return 0;
}
```

В этом примере мы получили *методы* 2-х *базовых классов* в *производном*. При вызове *конструтора класса-наследника*, вызовятся и *наследуемые методы* базовых классов.

Обратите внимание, порядок вызова *контруторов базовых классов* определяется их порядковым числом в месте наследования.

То есть, в нашем примере:

```
class FlyingCar : public Car, public Plane
```

Класс `Car` стоит *первее*, чем `Plane` , следовательно, сначала вызовется *конструтор класса* `Car` , а потом `Plane` .

32. Наследование. Ограничение доступа при наследовании. Пример.

Наследование - это механизм, который позволяет создавать *новые классы* на основе уже *существующих*. Класс, от которого происходит наследование, называется *базовым классом*, а класс, который *наследует свойства* и методы *базового класса*, называется *производным классом*.
В производном классе можно *переопределять* методы *базового класса*, добавлять свои *методы* и *поля*, а также *вызывать методы* и использовать *поля базового класса*.
Ограничение доступа при наследовании:
`Public` - доступ к наследуемым полям и методам везде (в 2-х классах и основной программе `main()`)
`Protected` - доступ к наследуемым полям и методам только из самого класса и из производных классов (нет доступа в основной программе!)
`Protected` - доступ к наследуемым полям и методам только внутри самого класса!
Пример:

```
class A
{
    public:
       int x;
    protected:
       int y;
    private:
       int z;
};
class B : public A
{
    // x - public
    // y - protected
    // z недоступен из B
};
class C : protected A
{
    // x - protected
    // y - protected
    // z недоступен из C
};
class D : private A    // 'private' is default for classes
{
    // x - private
    // y - private
    // z недоступен из D
};
```

Для каждого *модификатора доступа* при наследовании меняется и способ передачи *полей/-методов* в производный класс, т.е. все определяется *"силой"* модификатора доступа:
От слабого к сильному (*по способу защищенности*):
`Public -> Protected -> Private`
Это значит, что при *наследовании* через `Protected` , каждый `Public` *метод/поле* станет `Protected` , а `Private` , если он имеется, останется *без изменений* (потому `Private` *сильнее* `Protected`). При наследовании через `Private` , абсолютно все поля/методы станут `Private` . Условно, *сильный модификатор доступа при наследовании* влияется только *на*

слабый модификатор доступа внутри класса.

33. Конструкторы и деструкторы производных классов. Пример.

```
class MyClass
{
public:
    MyClass()
    {
        cout << "Constructor Class 1" << endl;
    }
    ~MyClass()
    {
        cout << "Destructor Class 1" << endl;
    }
};

class Class2 : public MyClass   // Свободное наследование
{
public:
    Class2()
    {
        cout << "Constructor Class 2" << endl;
    }
    ~Class2()
    {
        cout << "Destructor Class 2" << endl;
    }
};

int main()
{
    Class2 cl1;                 // Создание объекта

    return 0;
}
```

При наличии *конструторов* и *деструторов* в 2-х классах, один из которых *наследуется от другого*, конструторы вызываются в *прямой последовательности* (сначала для *базового класса*, потом для *производного*), а *деструкторы наоборот* (сначала деструктор *производного класса*, потом *базового*). То есть:
Отладка верхней программы: - Constructor Class 1 - Constructor Class 2 - Destructor Class 2 - Destructor Class 1
Конструкторы производных классов вызываются при *создании объекта* производного класса. Они могут использовать конструкторы *базового класса* для инициализации *унаследованных* полей. Если конструктор *не определен* в производном классе, компилятор автоматически вызовет *конструктор по умолчанию* базового класса.
Деструкторы производных классов вызываются при удалении объекта производного класса. Они могут использовать деструкторы *базового класса* для *освобождения унаследованных ресурсов*.

34. Композиция. Пример.

Композиция - это механизм, который позволяет *создавать объекты*, содержащие *другие объекты в качестве своих частей*. Это достигается путем включения *одного класса в другой класс* в качестве его полей.

Пример композиции:

```
class Human
{
public:
    void Think()
    {
        brain.Think();
    }

private:
    class Brain          // Класс Brain внутри класса Human
    {
    public:
        void Think()
        {
            cout << "I think" << endl;
        }
    };

    Brain brain;         // Создаем объект класса Brain, чтобы вызвать его в методе Think()
};

int main(void)
{
    setlocale(LC_ALL, "ru");

    Human *human = new Human();
    human->Think();

    delete human;

    return 0;
}
```

1. Класс `Brain` без нашего человека *никак существовать не может*.
2. Класс `Brain` (еще называют "*иннер*" классом) мы больше *нигде не можем использовать*, т.к. мозг жестко *инкапсулирован* в человека (находится внутри секции `Private`).
3. *Композиция* - объект класса не может существовать без другого объекта класса (*жесткая привязка*).

35. Наполнение. Пример.

Агрегация / Наполнение - это отношение между объектами, при котором *один объект* является *частью другого объекта* и при этом *может существовать независимо* от него (*не жесткое включение*).

Пример наполнения:

```
class Cap // Объект класса кепка может быть на голове человека, а может и не быть, поэтому прописывается вне основного класса и не включается жестко в класс Human, как это было с классом Brain
{
public:
    string Get_Color()
    {
        return color;
    }

private:
    string color = "red";
};

class Human
{
public:
    void Think()
    {
        brain.Think();
    }

    void InspectTheCap()
    {
        cout << "My cap is " << cap.Get_Color() << endl;
                        // Вызываем метод Get_Color() класса Cap внешний
    }

private:
    class Brain         // Иннер класс обязательно включается в человека
    {
    public:
        void Think()
        {
            cout << "I think" << endl;
        }
    };

    Brain brain;         // Объект класса Brain иннер класс
    Cap cap;             // Объект класса Cap внешний класс
};

int main(void)
{
    setlocale(LC_ALL, "ru");

    Human human;
    human.Think();
    human.InspectTheCap();

    return 0;
}
```

Наполнение или агрегация - явление *не жесткого включения класса*, т.е. включения объекта одного класса в другой может и не быть.
`Brain` у человека *есть всегда* (не у всех xd), поэтому его включение *обязательно* (количество объектов *варьируется* от 1 до $+\infty$). Что нельзя сказать про `Cap` , ведь кепка у человека *может быть*, а может и *не быть*, поэтому она включается *не жестко в класс* (количество объектов *варьируется* от 0 до $+\infty$). Это главное отличие *композиции* от *наполнения*.
Класс `Cap` можно использовать *с другими классами*, т.к. он *не привязан жестко* к какому-либо классу.

36. Полиморфное наследование. Простой полиморфизм. Пример.

Простой полиморфизм - механизм *переопределения методов базового класса* при *наследовании* в *производном классе* (в примере - это метод `Bark()`).
То есть эти методы имеют *одинаковые идентификаторы*, но *разную реализацию метода* для конкретного класса.
Пример программы с простым полиморфизмом:

```
class Animal
{
public:
    virtual void Bark()     // Виртуальный метод так как метод Bark() будет переопределяться в производном классе
    {
        cout << "WOUF!" << endl;
    }
};

class Dog : public Animal
{
public:
    void Bark() override    // Директива 'override' переопределяет метод базового класса и проверяет правильность ввода параметров в скобки
    {
        cout << "WOUF! WOUF! WOUF!" << endl;
    }
};

int main(void)
{
    setlocale(LC_ALL, "ru");

    Dog denji;
    Animal ani;
    ani.Bark();
    denji.Bark();

    return 0;
}
```

Простой полиморфизм происходит на этапе *компиляции программы* (*раннее связывание*), т.е. компилятор определяет *тип объекта* и вызывает соответствующую *реализацию метода*.

37. Полиморфное наследование. Сложный полиморфизм. Пример.

Сложный полиморфизм - это механизм *переопределения* методов, используемый для создания иерархии классов в *различной функциональностью*. (чаще используется вместе с *абстрактными классами*).

Пример программы со сложным полиморфизмом:

```
class Shape {                                   // Абстрактный класс его методы переопределяются в классахнаследниках-
public:
    virtual double getArea() = 0;               // Чистый виртуальный метод
};

class Circle : public Shape {                   // Наследуем абстрактный класс
public:
    Circle(double radius) {                     // Конструтор класса
        this->radius = radius;
    }

    double getArea() override {                 // Переопределение метода абстрактого класса
        return 3.14 * radius * radius;
    }
private:
    double radius;
};

class Rectangle : public Shape {                // Наследуем абстрактный класс
public:
    Rectangle(double length, double width) {    // Конструтор класса
        this->length = length;
        this->width = width;
    }

    double getArea() override {                 // Переопределение метода абстрактого класса
        return length * width;
    }
private:
    double length;
    double width;
};

int main() {
    Shape* shapes[2];                           // Массив указателей
    shapes[0] = new Circle(5);                  // 1 элемент массива
    shapes[1] = new Rectangle(4, 6);            // 2 элемент массива


    for(int i = 0; i < 2; i++) {                // Проходим циклом по элементам динамического массива и вызываем метод getArea() для вывода площади фигуры
        cout << "Area of shape " << (i+1) << " is: " << shapes[i]->getArea() << endl;
    }

    for (int i = 0; i < 2; i++)
    {
        delete shapes[i];                       // Освобождаем память
    }

    return 0;
}
```

В этом примере используется *абстрактный базовый класс* `Shape` , который определяет *общий интерфейс для классов-наследников* `Circle` и `Rectangle` . Классы-наследники переопределяют метод `getArea()` , который возвращает *площадь фигуры*.
В C++ нельзя создать объект абстрактного класса!* Это логично, потому что *методы абстрактных классов не имеют реализации* и используются *в других классах*, в которых они *переопределяются* и имеют свой *собственный код-реализацию*.

38. Статические компоненты классов. Пример.

Статические поля в классах - это *переменные*, которые принадлежат *классу в целом*, а *не конкретному объекту класса*. Они объявляются с помощью *ключевого слова* `static` .
`static` переменная является *общей для всех объектов класса*.
Инициализация *статической переменной* осуществляется *вне класса*!. Для этого надо использовать *следующую конструкцию*:

```
<Variable_Type> Class_Name::Variable_Name = {Value}
```

Статические методы - это методы класса, определенные *при помощи* того же слова `static` .
Статические методы *не получают параметра* `this` , потому что они *не оперируют конкретным объектом класса*, а работают *на уровне класса* в целом.
Пример программы со статическими компонентами:

```
class Apple
{
public:
    static int count;       // Статическая переменная

    Apple(int weight, string color)
    {
        this->weight = weight;
        this->color = color;
        count++;
    }

    static void printCount()  // Статический метод
    {
        cout << "Number of apples created: " << count << endl;
    }

private:
    int weight;
    string color;
};

int Apple::count = 0;          // Инициализация статического поля вне класса

int main(void)
{
    setlocale(LC_ALL, "ru");

    Apple apple1(150, "Red");
    Apple apple2(200, "Green");
    Apple apple3(200, "Green");

    cout << apple1.count << endl;   // 3
    cout << apple2.count << endl;   // 3
    cout << apple3.count << endl;   // 3

    cout << Apple::count << endl;   // 3
    Apple::printCount();            // 3

    return 0;
}
```

39. Особенности работы с динамическими объектами. Пример.

Динамические объекты - способ конструирования программы, при котором *динамически выделяется память* под *создание объектов*.
Основные особенности работы с динамическими объектами в C++:

1. **Создание динамических объектов** происходит с помощью оператора `new` . Оператор `new` возвращает *указатель на выделенную область памяти*, которую необходимо освободить с помощью оператора `delete` .
2. **Освобождение памяти**, занятой динамическим объектом, происходит с помощью оператора `delete` . Если *не освободить память*, занятую динамическим объектом, это приведет к *утечке памяти*.
3. **Освобождение динамической памяти**, выделенной для *полей* класса, происходит внутри деструктора.

Пример работы с динамическими объектами:

```
class MyClass
{
public:
    MyClass()
    {
        cout << "Constructor called" << endl;
    }

    MyClass(int num)
    {
        this->num = num;
    }

    void Print()
    {
        cout << num << endl;
    }

    ~MyClass()
    {
        cout << "Destructor called" << endl;
    }

private:
    int num;
};

int main()
{
    MyClass *obj = new MyClass(5);      // Указатель на объект
    obj->Print();                       // Вызов метода через указатель

    delete obj;

    return 0;
}
```

Создание динамического массива объектов класса:

```
class MyClass
{
public:
    MyClass()
    {
        cout << "Constructor called" << endl;
    }

    void init(int num)
    {
        this->num = num;
    }

    void Print()
    {
        cout << num << endl;
    }

    ~MyClass()
    {
        cout << "Destructor called" << endl;
    }

private:
    int num;
};

int main()
{
    MyClass *object = new MyClass[5];   //  динамический массив из 5 объектов класса MyClass

    object[0].init(5);                  // Инициализация 5 объектов
    object[1].init(10);
    object[2].init(15);
    object[3].init(20);
    object[4].init(25);

    for (auto i = 0; i < 5; i++)
    {
        object[i].Print();
    }

    delete[] object;                    // Пишем [] чтобы освободить память всего массива, а не его
    первого элемента

    return 0;
}
```

Отладка программы: - Вызываем конструктор - Вызываем конструктор - Вызываем конструктор - Вызываем конструктор - Вызываем конструктор - 5 - 10 - 15 - 20 - 25 - Вызываем деструктор - Вызываем деструктор - Вызываем деструктор - Вызываем деструктор - Вызываем деструктор

40. Правило Пяти. Конструктор перемещения, операция присваивания перемещением.

Если *класс* или *структура* определяет *один из следующих методов*, то они должны явным образом определять *все виды методов*:

1. *Копирующий конструктор.*
2. *Конструктор перемещения.*
3. *Оператор присваивания.*
4. *Оператор присваивания перемещением.*
5. *Деструктор - если не используются “умные указатели”.*

Конструктор перемещения вызывается, если параметр - *временный объект*:

```
Имякласса
_ Имя(класса_ && Имяобъекта_) {...}
```

Оператор присваивания перемещением вызывается, если присваиваемый объект - *временный*:

```
Имякласса
_ Имякласс_::operator= Имя(класса_ && Имяобъекта_) {...}
```

Если объект имеет *физический адрес* (не является временным), а требует организовать *вызов конструктора перемещения* или *оператор присваивания перемещением*, то используют *функцию* `move()`

```
Имякласса
_ && std::move Имя(класса_ & Имяобъекта_)
```

Пример кода с привилом Пяти:

```
class Number
{
private:
    int *pnum;

public:
    Number(int Num) : pnum(new int(Num))              // Инициализация конструктора
    {
        cout << "New, Constructor" << endl;
    }
    Number(const Number &R) : pnum(new int(*R.pnum))  // Инициализация копирующего конструктора (1)
    {
        cout << "New, Constructor copy" << endl;
    }
    Number() : pnum(nullptr) {}                       // Конструктор без параметров
    Number &operator=(const Number &R)                // Оператор присваивания
    {
        if (pnum != nullptr)
        {
            delete pnum;
            cout << "Free" << endl;
        }
```

```
        pnum = new int(*R.pnum);
        cout << "New Operator= copy" << endl;
        return *this;
    }
    ~Number()                                           // Деструктор
    {
        if (pnum != nullptr)
        {
            delete pnum;
            cout << "Free" << endl;
        }
        cout << "Destructor" << endl;
    }
    Number(Number &&R) : pnum(R.pnum)                   // Конструктор перемещения
    {
        R.pnum = nullptr;
        cout << "Constructor move" << endl;
    }
    Number &operator=(Number &&R)                       // Оператор присваивания перемещением
    {
        if (pnum != nullptr)
        {
            delete pnum;
            cout << "Free move" << endl;
        }
        pnum = R.pnum;
        R.pnum = nullptr;
        cout << "Operator= move" << endl;
        return *this;
    }
};

Number f(int a, int b)
{
    Number temp(a + b);
    return Number(move(temp));
}

int main()
{
    Number A(5);
    Number B(A);
    Number C(move(A));
    Number D(6);
    D = move(A);
    Number F = f(6, 7);
    return 0;
}
```

41. Объекты с динамическими полями. Копирующий конструктор. Пример.

Динамические поля - это *поля*, память для которых *выделяется* динамически. Память для них выделяется с помощью `new` (как правило, в инициализирующих методах), освобождается с помощью `delete` (как правило, в деструкторе).

Пример программы:

```
class MyClass
{
public:
    MyClass(int size)
    {
        this->size = size;
        this->data = new int[size];
                                            // Создаем динамический массив

        for (auto i = 0; i < size; i++)
        {
            this->data[i] = i;
        }
    }

                                            //Внизу копирующий конструтор, принимающий в качестве параметра объект переменную типа MyClass
                                            //Параметр передаем по ссылке, а не по значению. Это нужно для
                                            //1. Передачи оригинального объекта нашего класса, а не его копии.
                                            //2. Безопасного копирования динамической памяти.

    MyClass(const MyClass &other)
    {
        this->size = other.size;
        this->data = new int[size];

        for (auto i = 0; i < size; i++)
        {
            this->data[i] = other.data[i];
        }
    }

    void Print()
    {
        for (auto i = 0; i < size; i++)
        {
            cout << data[i] << " ";
        }
        cout << endl;
    }

    ~MyClass()
    {
        delete[] data;
    }

private:
    int *data;
    int size;
```

```
};

int main(void)
{
    setlocale(LC_ALL, "ru");

    MyClass class1(5);
    MyClass class2 = class1;                //Вызов копирующего конструктора.

    class1.Print();
    class2.Print();

    return 0;
}
```

42. Дружественные функции, методы и классы. Пример.

Дружественные *функции*, *методы*, *классы* - это *механизмы*, которые позволяют обойти *обычные правила доступа* к членам класса.
Дружественная функция - это функция, которая объявляется *дружественной* внутри определения класса (указывается только *сигнатура функции*). Такая функция имеет доступ к *закрытым членам класса* и может быть *вызвана извне класса*, как *обычная функция*.
Пример программы с дружественной функцией:

```
class Point
{
public:

    Point(int x, int y)                      // Конструктор инициализации полей класса через
    константный указатель.
    {
        this->x = x;
        this->y = y;
    }

    void Print()
    {
        cout << x << " " << y << endl;
    }

    friend void ChangeX(Point &other);      // Пишем сигнатуру внешней функции с добавлением спец
    слова friend

private:
    int x;
    int y;
};

                                            // Передаем оригинальный объект по ссылке для работы с
    его полями.
void ChangeX(Point &other)
{
    other.x = -1;
}

int main(void)
{
    setlocale(LC_ALL, "ru");

    Point point(5, 1);
    point.Print();

    ChangeX(point);                          // Вызываем дружественную функцию
    point.Print();

    return 0;
}
```

Дружественный метод одного класса имеют доступ к `private` и `protected` полям другого класса, в котором прописана *сигнатура этого метода*. Он также объявляется *дружественным* внутри класса.
Пример кода с дружественным методом:

```
class Apple;                                        // Определение класса.

class Human
{
public:
    void TakeApple(Apple &other);                   // Прописываем сигнатуру метода. Мы не имеем
    доступа к private полям класса выносим класс наружу.
};

class Apple
{
public:
    Apple(int weight, string color)
    {
        this->weight = weight;
        this->color = color;
    }

    friend void Human::TakeApple(Apple &other);      // Дружественный метод

private:
    int weight;
    string color;
};

void Human::TakeApple(Apple &other)
{
    cout << "TakeApple " << other.weight << " " << other.color << endl;
}

int main(void)
{

    Apple apple(150, "Red");
    Human human;
    human.TakeApple(apple);

    return 0;
}
```

Если прописать класс `Human` *под классом* `Apple` , сам класс `Apple` ничего *не будет знать* о классе `Human` . Перенесем данный класс *наверх* и допишем *определение класса* `Apple` .
Дружественный класс - это *класс*, который имеет *доступ к закрытым полям и методам другого класса*. Для этого он также объявляется *дружественным внутри определения этого класса*. В данном случае класс, который объявляется *дружественным*, видит *все члены класса*, включая *приватные*.
Пример кода с дружественным классом:

```
class Apple;

class Human
{
public:
    void TakeApple(Apple &other);
};

class Apple
{
```

```
11      friend Human; // Объявляем дружественный класс все private поля Apple становятся доступны для
        класса Human
12
13  public:
14      Apple(int weight, string color)
15      {
16          this->weight = weight;
17          this->color = color;
18      }
19
20  private:
21      int weight;
22      string color;
23  };
24
25  void Human::TakeApple(Apple &other)
26  {
27      cout << "TakeApple " << other.weight << " " << other.color << endl;
28  }
29
30  int main(void)
31  {
32
33      Apple apple(150, "Red");
34      Human human;
35      human.TakeApple(apple);
36      cin.get();
37
38      return 0;
39  }
```

43. Переопределение операций. Пример.

Переопределение операций - это возможность определить *свою реализацию стандартных операций* для пользовательских типов данных, таких как *классы*.
Для *переопределения операций* в C++ используются *специальные методы*, называемые *операторами перегрузки*.
Например, оператор *“+”* может быть переопределен для класса, чтобы позволить *складывать объекты этого класса*. Если *не определить* операцию для объектов, то *будет ошибка*, т.к. по умолчанию данную операцию *нельзя использовать* для *операндов* типа “класс”.
Пример программы, преобразующей некоторые операции:

```
class A
{
public:
    A() {}
    A(int num)
    {
        this->num = num;
    }

    // Тип перегрузки операторов определяется типом вызываемого значения в основной программе, те
    // Снизу оператор имеет тип bool, тк в основной программе result тоже имеет тип bool
    // Аналогично для оператора типа A в( основной программе мы передаем объект того же типа – A a3)

    bool operator==(const A &other)
    // Перегрузка оператора сравнения, тк изначально он не определен для операндов типа класс.
    {
        return this->num == other.num;
    }

    A operator+(const A &other)
    // Перегрузка оператора суммирования значений экземпляров объектов класса.
    {
        A temp(this->num + other.num);
        return temp;
    }

    void Print()
    {
        cout << num << endl;
    }

private:
    int num;
};

int main(void)
{
    setlocale(LC_ALL, ”ru”);

    A a1(2);
    A a2(2);
    bool result = a1 == a2;
    cout << result << endl; // 1 те True
    A a3 = a1 + a2;
    a3.Print(); // 4

    return 0;
```

```
48 }
```

Перегрузка операторов *инкремента* и *декремента*.
различают *префиксную* и *постфиксную* записи:

```
1 Point & operator ++() {}          //префиксная запись a++
2 Point & operator ++(int) {}       //постфикская запись a++
```

Аргумент типа `int` во втором случае означает постфиксную форму оператора.

1. Можно переопределять только *операции*, параметры которых – *объекты*.
2. *Не разрешается переопределение:* `sizeof` , `? :` (*тернарные операторы*), `#` , `##` , `::` (*пространство имен*), `<класс> ::` (*область видимости*)
3. При переопределении операций *нельзя изменить* ее *приоритет* и *ассоциативность*.

44. Шаблоны классов. Пример.

Шаблон класса – обобщенное *описание класса*, содержащее *параметры*, позволяющие задавать типы *используемых полей* или других данных. Шаблоны классов *определяются* с использованием *ключевого* слова `template` .

Описание шаблона класса:

```
template Списокпараметров_ Описаниекласса_
```

Операция *создания описания класса* из *шаблона* называется **инстанцированием*.

Пример программы с шаблоном класса:

```
template <typename T>                    // Создание шаблона с произвольным типом
class MyClass
{
public:
    MyClass(T value)
    {
        this->value = value;
    }

    void DataTypeSize()
    {
        cout << sizeof(value) << endl;    // Выводит размер переменной в байтах
    }

private:
    T value;
};

int main()
{
    int a = 5;
    MyClass<int> myclass(a);
    myclass.DataTypeSize();               // 4 размер int в байтах

    long long b = 10;
    MyClass<long long> myclass2(b);
    myclass2.DataTypeSize();              // 8 размер long long в байтах

    return 0;
}
```

45. Шаблоны функций. Пример.

Шаблоны функций в C++ - это обобщенный *механизм*, который позволяет *создавать функции*, работающие с *различными типами данных* (прописывают со *спец. словом* `template`).
Перегрузка функций: (НАПОМИНАНИЕ)
Компилятор *автоматически определяет*, какую функцию *выбрать*, если они имеют *одинаковые идентификаторы* (в нашем случае - это функция `Sum()`)

```
int Sum(int a, int b)               // 1 функция суммирования целые числа
{
    return a + b;
}

double Sum(double a, double b)      // 2 функция суммировния вещественные числа
{
    return a + b;
}

int main()
{
    cout << Sum(4, 10) << endl;     // Инициализируем поля целыми числами, значит компилятор
    выбирает 1 функцию
    cout << Sum(4.6, 9.5) << endl;  // Инициализируем поля вещественными числами, значит компилятор
    выбирает 2 функцию
    return 0;
}
```

Для чего нужно было напоминание про перегрузку функций? Дело в том, что *шаблоны функций заменяют перегрузку этих функций* и нам не нужно писать *большое количество функций* для различных *типов* передаваемых значений.
Пример программы с шаблоном функции:

```
template <typename T>   // Описание шаблона с общим типом T
T Sum(T a, T b)         // Описание шаблонной функции
{
    return a + b;
}

int main()
{
    cout << Sum(4, 10) << endl;
    cout << Sum(4.6, 9.5) << endl;

    return 0;
}
```

В этой программе компилятор *автоматически определяет тип* передаваемого значения `T` и выводит *соответствующее значение*.
Шаблоны функций в C++ позволяют создавать *обобщенный код*, который может работать с *различными типами данных*, *НЕ требуя* явного указания типа данных, потому что он *определится автоматически* при компиляции.
Пример программы с шаблоном и разными передаваемыми типами:

```
template <typename T1, typename T2>
T1 Sum(T1 a, T2 b)
{
```

```
    return a + b;
}

int main()
{
    cout << Sum(4, 10.9) << endl;

    return 0;
}
```

Здесь в функцию `Sum()` передается сначала параметр типа `int` , а потом `double` , то есть в шаблоне `T1` будет иметь тип `int` , а `T2` тип `double` .

Интересный факт:

Вместо `typename` можно написать слово `class` (они *ничем не отличаются*):

```
template <class T>
```

46. Организация библиотеки ввода/вывода C++. Операции извлечения и вставки.

Стандартная библиотека ввода/вывода C++ (`iostream`) предоставляет средства для работы с *потоками ввода и вывода данных*. Она предоставляет *операции извлечения* (`>>`) и *вставки* (`<<`) для работы *с потоками*.

1. **Операция извлечения** (`>>`) позволяет считать данные из *потока ввода* и *сохранить* их в *переменные*. Она может использоваться для чтения различных типов данных, таких как *целые числа*, *вещественные числа*, *символы*, *строки* и т.д.
2. **Операция вставки** (`<<`) позволяет *вывести данные* в *поток вывода*. Она может использоваться для вывода *различных типов данных*, таких как *целые числа*, *вещественные числа*, *символы*, *строки* и т.д.
3. **Операции сдвигов** `<<`, `>>` в классах потоков *переопределены для обозначения операций ввода-вывода*.

Пример программы с потоками ввода/вывода:

```
#include <iostream>                // Подключение потоков ввода вывода
#include <string>

using namespace std;

int main()
{
    int age;
    string name;

    cout << "Enter your name: ";    // Вставка в поток

    cin >> name;                   // Извлечение из потока

    cout << "Enter your age: ";
    cin >> age;

    cout << "Hello, " << name << "! You are " << age << " years old." << endl;

    return 0;
}
```

47. Организация контейнеров на классах. Пример диаграммы классов.

Контейнер в C++ - это класс, который позволяет управлять *коллекцией* объектов *одного типа*.
2 признака контейнеров:

1. *Контейнер* - это объект класса, который содержит в себе *объекты других классов*.
2. Все классы *связаны между собой*.

Стандартная библиотека C++ предоставляет *множество контейнеров*, каждый из которых имеет свои *уникальные свойства* и *методы*. Некоторые из наиболее *распространенных контейнеров*:

1. `std::vector` - это *динамический массив*, который может хранить элементы *любого типа*.
2. `std::list` - это *двусвязный список*, который предоставляет *быструю вставку* и *удаление элементов* в *любом месте списка*.
3. `std::map` - это *ассоциативный контейнер*, который представляет собой отображение *ключей* на *значения*. Каждый ключ может иметь *только одно соответствующее ему значение*.
4. `std::set` - это контейнер, который содержит *уникальные элементы* в отсортированном порядке.

Пример программы с использыванием контейнера:

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

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

    string GetName() const                          // Передаем через const чтобы не изменять
    состояние объекта
    {
        return name;
    }

    int GetAge() const                              // Передаем через const чтобы не изменять
    состояние объекта
    {
        return age;
    }

```

```
private:
    string name;
    int age;
};

int main()
{
    vector<Person> persons;                          // Создание вектора объектов класса Person

    persons.push_back(Person("Alice", 25));          // Добавление объектов в вектор
    persons.push_back(Person("Bob", 30));
    persons.push_back(Person("Charlie", 35));

    for (const auto &person : persons)               // Вывод информации о каждом объекте вектора
    {
        cout << "Name: " << person.GetName() << ", Age: " << person.GetAge() << endl;
    }

    return 0;
}
```

48. Организация контейнеров на шаблонах. Пример диаграммы классов.

Организация контейнеров на шаблонах позволяет создавать *универсальные контейнеры*, которые могут хранить элементы *разных типов*. Шаблон контейнера представляет собой *обобщенный класс*, который использует *параметр типа* для определения типа элементов, которые *содержит контейнер*.

Пример шаблонного контейнера на основе вектора в C++:

```cpp
#include <iostream>
#include <vector>
#include <string>

template <typename T>                       // Объявляем шаблон класса контейнера

class MyContainer {
private:
    vector<T> v;                            // Вектор для хранения элементов
public:
    void add(T x) {
        v.push_back(x);                     // Добавление элемента в вектор
    }
    void show() {                           // Вывод элементов контейнера
        cout << "Elements in container: ";
        for (int i = 0; i < v.size(); i++) {
            cout << v[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    MyContainer<int> c1;    // Создаем контейнер для хранения целых чисел
    c1.add(1);              // Для объекта c1 шаблон будет инициализирован только int типом
    c1.add(2);
    c1.show();

    MyContainer<string> c2;  // Создаем контейнер для хранения строк
    c2.add("Hello");         // Для объекта c1 шаблон будет инициализирован только string типом
    c2.add("world");
    c2.show();

    return 0;
}
```

49. Организация интерфейса с использованием виджетов Qt. Пример.

Виджет в QT - это графический элемент *интерфейса пользователя*, который может быть размещен *на окне приложения*. Виджетами могут быть *кнопки*, *поля ввода*, *метки*, *таблицы*, *графики* и т.д.

Пример простой программы в QT с использованием виджетов:

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("My App");

    QLabel *label = new QLabel("Hello, World!", window);
    label->setGeometry(50, 50, 200, 50);

    QPushButton *button = new QPushButton("Click me!", window);
    button->setGeometry(50, 100, 200, 50);

    QObject::connect(button, &QPushButton::clicked, [=](){
        label->setText("Button clicked!");
    });

    window->show();

    return app.exec();
}
```

50. Сигналы, слоты и события Qt. Пример.

Сигналы, *слоты* и *события* являются *основными механизмами взаимодействия* между объектами в Qt.

1. **Сигналы** - это события, которые генерируются объектом при *определенных условиях*. Они могут быть переданы другим объектам, называемым *слотами*, для выполнения *определенных действий*.

2. **Слоты** - это *функции*, которые вызываются *при получении сигнала*. Они могут использоваться для *выполнения любых действий*, включая *изменение состояния объекта* или *вызов других функций*.

3. **События** - это действия, которые происходят в приложении, такие как *нажатие клавиши* или *щелчок мыши*. Они могут быть обработаны объектами, которые *подписались на определенные события*.

Пример использования сигналов, слотов и событий в QT
Допустим, у нас есть *класс* `MainWindow` , который содержит *кнопку* и *метку*. Мы хотим, чтобы при нажатии на кнопку, текст в метке *изменился* на *“Hello, world!”*.
Сначала мы создаем *слот* в классе `MainWindow` :

```
public slots:
    void onButtonClicked();
```

Затем мы подключаем *сигнал “clicked()”* кнопки к *слоту* `onButtonClicked()` :

```
connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(onButtonClicked()));
```

Наконец, мы реализуем *слот*, который *изменит текст в метке:*

```
void MainWindow::onButtonClicked()
{
    ui->label->setText(”Hello, world!”);
}
```