Постановка задачи

Вывод иерархического дерева объектов на консоль

Внутренняя архитектура (вид иерархического дерева объектов) в большинстве реализованных программах динамически меняется в процессе отработки алгоритма. Вывод текущего дерева объектов является важной задачей, существенно помогая разработчику, особенно на этапе тестирования и отладки программы.

Построить модель иерархической системы. Реализовать вывод на консоль иерархического дерева объектов в следующем виде:

```
                    root



ob_1

ob_2

ob_3

ob_4

ob                      _5



ob
_6



ob
_7
```

где: root - наименование корневого объекта (приложения).

Состав и иерархия объектов строиться посредством ввода исходных данных. Ввод организован как в контрольной работе № 1.

Система содержит объекты пяти классов, не считая корневого. Номера классов: 2,3,4,5,6.

Описание входных данных

Множество объектов, их характеристики и расположение на дереве иерархии. Структура данных для ввода согласно изложенному в фрагменте методического указания в контрольной работе № 1.

Описание выходных данных

Вывести иерархию объектов в следующем виде:
Object tree
«Наименование корневого объекта» «Наименование объекта 1»
«Наименование объекта 2» «Наименование объекта 3»
. . . . .

Отступ каждого уровня иерархии 4 позиции.

Метод решения

К классу Base добавлен метод:

public:

- print_tree_view(int deep_lvl) - метод печати поддерева данной вершины в красивом виде

Добавлены два новых класса:

D

public:

- D(Base * parent, string name, int status): Base(paremt, name, status) - конструктор класса

E

public:

- E(Base * parent, string name, int status): Base(paremt, name, status) - конструктор класса

№	Имя класса	Имя классанаследника	Модификатор доступа при наследовании	Описание	Переход
1	Base			Базовый класс для всех остальных	
		App	public		2
		A	public		3
		B	public		4
		C	public		5
		D	public		6
		E	public		7
2	app			Класс-приложение	
3	B				
4	B				
5	C				
6	D				
7	E				

Описание алгоритма

Класс объекта: App

Модификатор доступа: public

Метод: build_tree

Функционал: Функция считывания дерева

Параметры: отсутствуют

Возвращаемое значение: отсутствует

№	Предикат	Действия	№ перехода
1		Инициализация переменных строкового типа s1, s2	2
2		Инициализация переменных целочисленного типа class_type и status со значением 0	3
3		Считывание s1	4
4		Вызов метода set_name у данного объекта с аргументом s1	5
5 6		Считывание s1 6 ∅ ∅	
	s1 не равно "endtree"	Считывание s2, class_type, status	7
7		Вызов метода get_by_name у данного объекта с аргументом s1	8
8		Присваивание результата работы get_by_name переменной b	9
9	class_type равен 2	Вызов метода add_child у данного объекта с аргументом новым объектом класса A проинициализированным с аргументами: указателем на данный класс, s2 и status	5
	class_type равен 3	Вызов метода add_child у данного объекта с аргументом новым объектом класса B проинициализированным с аргументами: указателем на данный класс, s2 и status	5
	class_type равен 4	Вызов метода add_child у данного объекта с аргументом новым объектом класса C проинициализированным с аргументами: указателем на данный класс, s2 и status	5

	class_type равен 5	Вызов метода add_child у данного объекта с аргументом новым объектом класса D проинициализированным с аргументами: указателем на данный класс, s2 и status	5	
	class_type равен 6	Вызов метода add_child у данного объекта с аргументом новым объектом класса E проинициализированным с аргументами: указателем на данный класс, s2 и status	5	
	class_type не равен ничему из перечисленного		5	

Класс объекта: Base

Модификатор доступа: public

Метод: print_tree_view

Функционал: Выводит информацию о вершинах в виде дерева

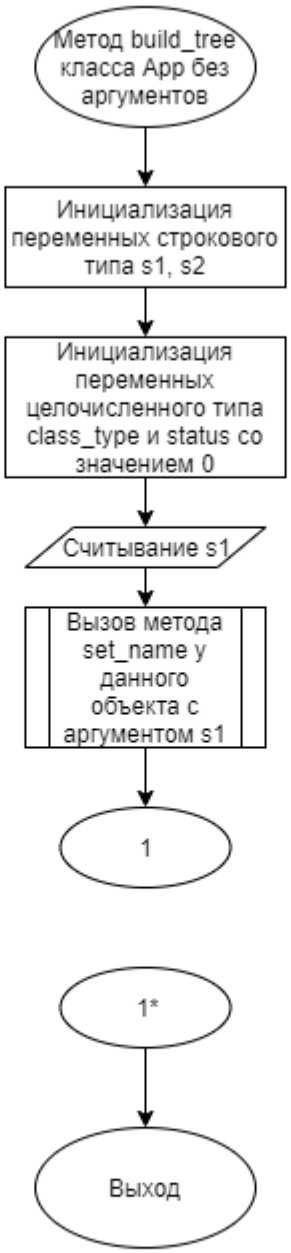
Параметры: int deep_lvl - уровень глубины текущей вершины относительно изначальной

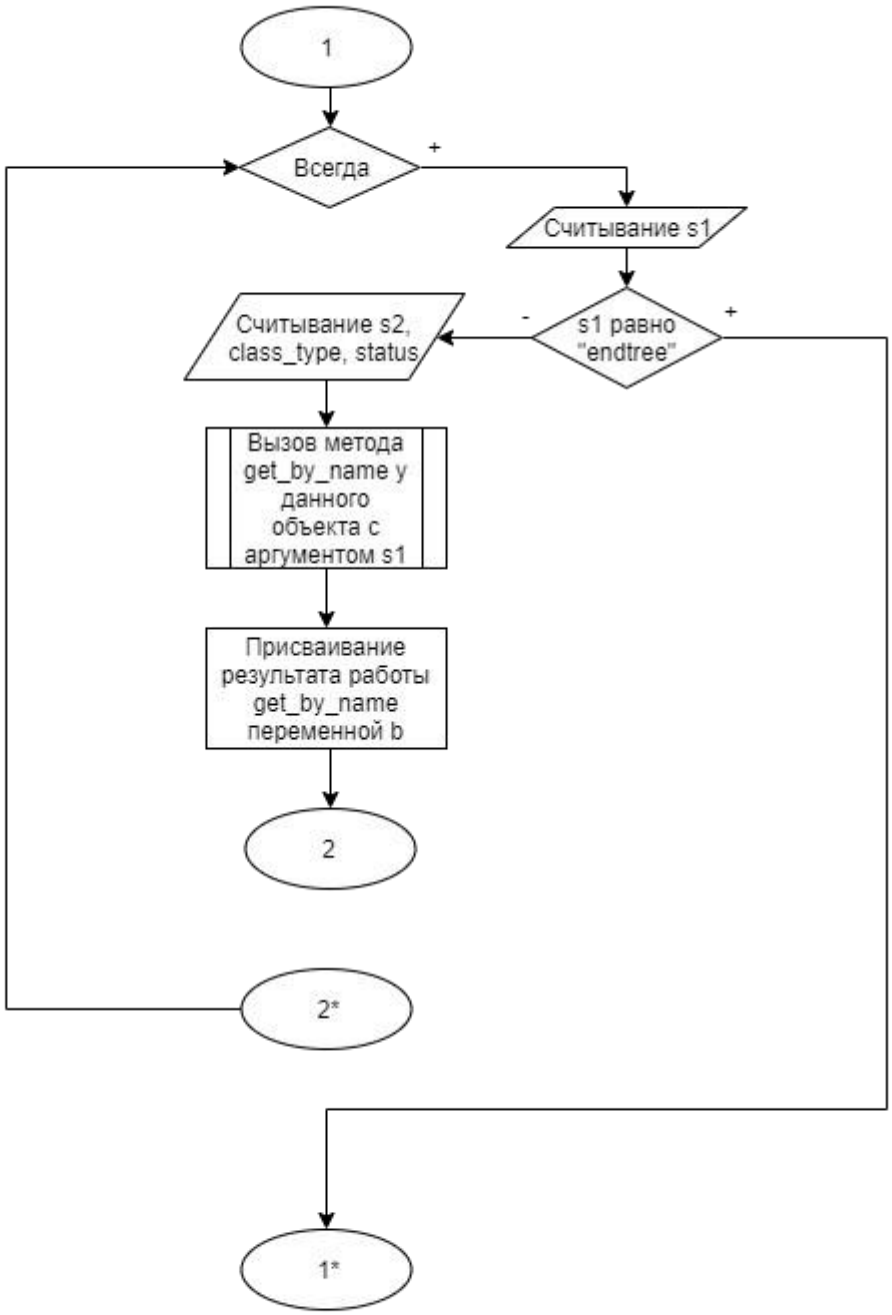
Возвращаемое значение: отсутствует

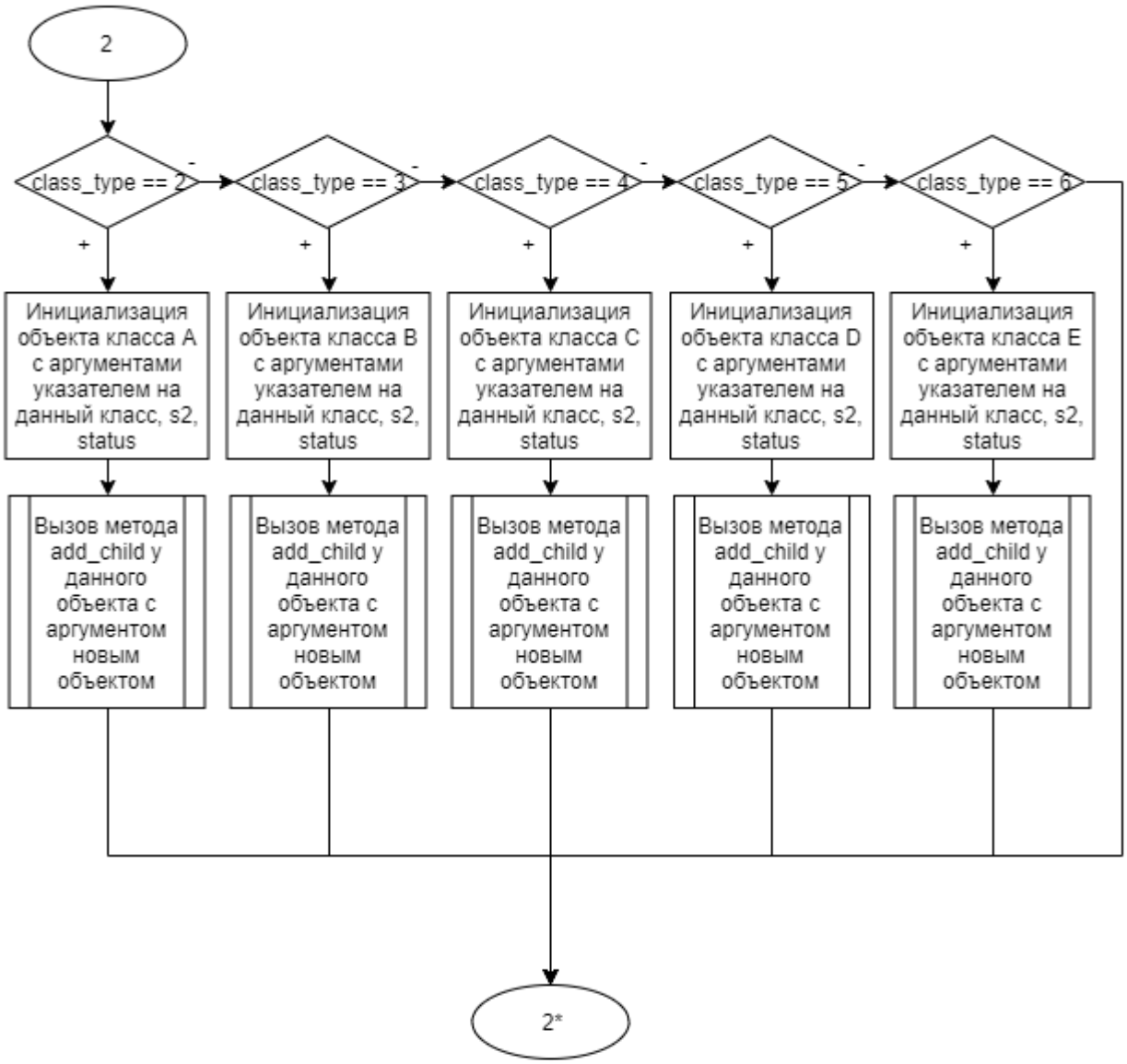
№	Предикат	Действия	№ перехода	Комментарий
1		Инициализация переменной indent целочисленного типа и присваивание ей значения 4 * deep_lvl	2	
2		Инициализация переменной i целочисленного типа и присвоение ей значения 0	3	
3	i меньше indent	Вывод пробела и увеличение i на единицу	3	
	i больше или равно indent		4	
4		Вывод имени вершины	5	

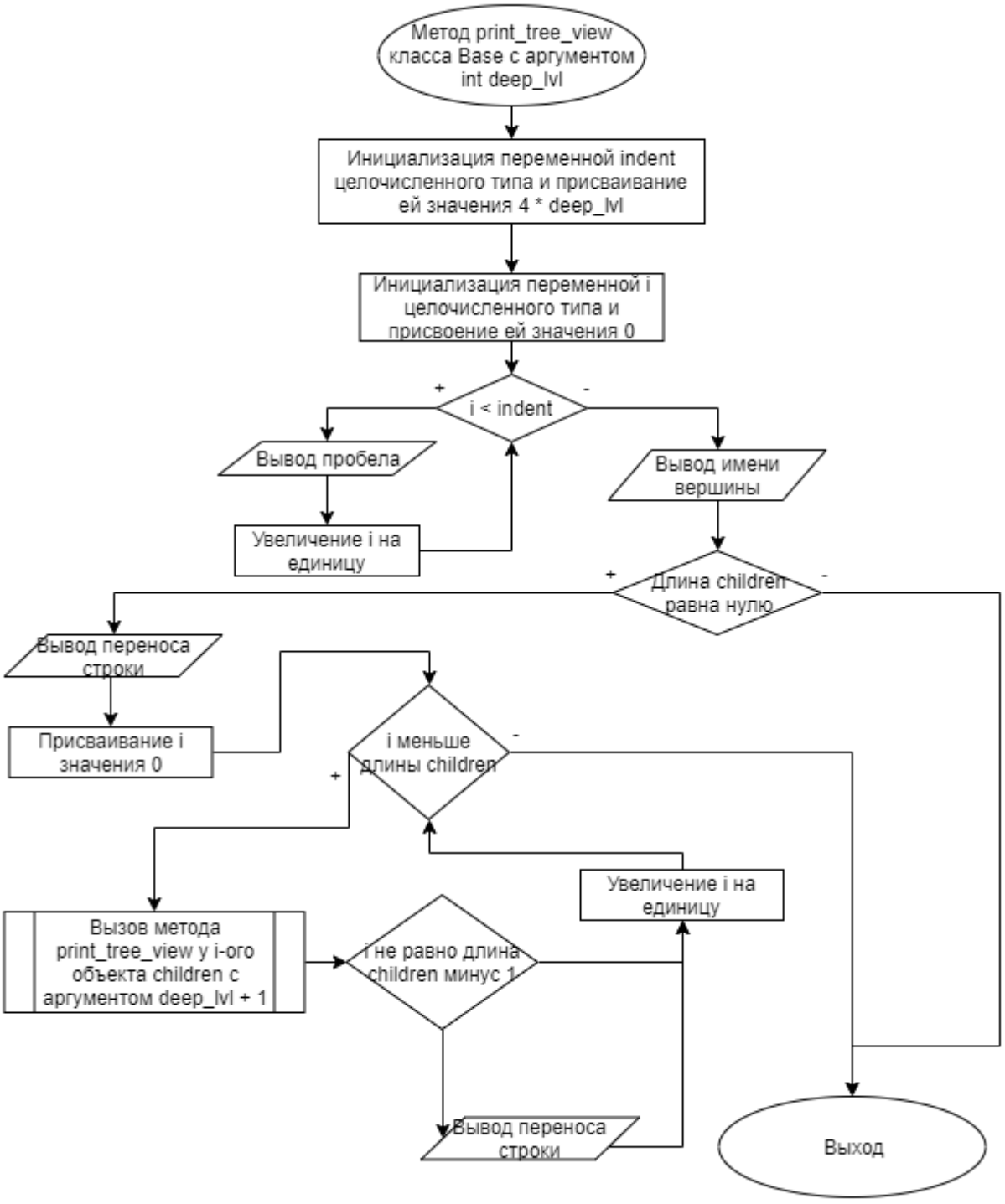
5	Длина закрытого свойства вектора children равна нулю		Ø	
	Длина закрытого свойства вектора children не равна нулю		6	
6		Вывод переноса строки	7	
7		Присваивание i значения 0	8	
8	i меньше длины children	Вызов метода print_tree_view у iого объекта children с аргументом deep_lvl + 1	9	
	i больше или равно длине children		Ø	
9	i не равно длине children минус 1	Вывод переноса строки	10	
	i равно длине children минус 1		10	
10		Увеличение i на единицу	8	

Блок-схема алгоритма


Метод build_tree класса App без аргументов
Инициализация переменных строкового типа s1, s2
Инициализация переменных целочисленного типа class_type и status со значением 0
Считывание s1
Вызов метода set_name у данного объекта с аргументом s1
1
1*
Выход


1
Всегда
+
Считывание s1
Считывание s2, class_type, status
-
s1 равно "endtree"
+
Вызов метода get_by_name у данного объекта с аргументом s1
Присваивание результата работы get_by_name переменной b
2
2*
1*


2
class_type == 2
class_type == 3
class_type == 4
class_type == 5
class_type == 6
-
+
Инициализация объекта класса A с аргументами указателем на данный класс, s2, status
Инициализация объекта класса B с аргументами указателем на данный класс, s2, status
Инициализация объекта класса C с аргументами указателем на данный класс, s2, status
Инициализация объекта класса D с аргументами указателем на данный класс, s2, status
Инициализация объекта класса E с аргументами указателем на данный класс, s2, status
Вызов метода add_child у данного объекта с аргументом новым объектом
2*


Метод print_tree_view класса Base с аргументом int deep_lvl
Инициализация переменной indent целочисленного типа и присваивание ей значения 4 * deep_lvl
Инициализация переменной i целочисленного типа и присвоение ей значения 0
i < indent
+
-
Вывод пробела
Увеличение i на единицу
Вывод имени вершины
Длина children равна нулю
+
-
Вывод переноса строки
Присваивание i значения 0
i меньше длины children
+
-
Увеличение i на единицу
Вызов метода print_tree_view у i-ого объекта children с аргументом deep_lvl + 1
i не равно длина children минус 1
Вывод переноса строки
Выход

Код программы

Файл a.cpp

```
#include <string>
#include "a.h"
#include "base.h"

A::A(Base * parent, std::string name, int status): Base(parent, name, status)
{};
```

Файл a.h

```
#ifndef AH
#define AH
#include <string>
#include "base.h"
        class A:
public Base{ public:
        A(Base * parent, std::string name, int status);
};

#endif
```

Файл app.cpp

```
#include <iostream>
#include <string>
#include "app.h"
#include "a.h"
#include "b.h"
#include "c.h"
#include "d.h" #include "e.h" using namespace

std; App::App(Base * parent): Base(parent, "",

1){};


void App::build_tree(){
string s1, s2;         int
class_type = 0, status;         //
Считываем имя корня
```

```
        cin >> s1;
        //  Задаем имя корня         this-
>set_name(s1);

        //  Начинаем считывание
while (true){
                //  Считывание двух строк, разделенных пробелом
cin >> s1;

                //  Проверяем, что не конец ввода
                if (s1 == "endtree"){
break;
                }
                cin >> s2 >> class_type >> status;

                //  Находим элемент с именем s1
                Base * b = this->get_by_name(s1);
                //  Добавляем к найденой вершине элемент с именем s2
                if (class_type == 2){
                        b->add_child(new A(b, s2, status));
                }else if (class_type == 3){
                        b->add_child(new B(b, s2, status));
                }else if (class_type == 4){
                        b->add_child(new C(b, s2, status));
                }else if (class_type == 5){
                        b->add_child(new D(b, s2, status));
                }else if (class_type == 6){
                        b->add_child(new E(b, s2, status));
                }
        }
}

void App::run(){         cout <<
"Object tree\n";         this-
>print_tree_view(0); }
```

Файл app.h

```
#ifndef APPH
#define APPH

#include "base.h"

class App : public Base{
public:
        App(Base * parent);
void build_tree();
void run();
};

#endif
```

Файл base.cpp

```
#include "base.h"
#include <iostream>
#include <vector>
#include <string>
using namespace std;



void Base::set_name(string new_name){
this->name = new_name;
return;
}
string Base::get_name(){
return this->name;
}

void Base::set_parent(Base * new_parent){
this->parent = new_parent;
return; }

Base::Base(Base * parent, string name, int status=0){
this->parent = parent;          this->name = name;
this->status = status;
}

Base * Base::get_parent(){
return this->parent;
}
        void Base::print_tree(){
if (!this->children.size())
return;

        //  Выводим имя данной вершины
cout << this->name;
        //  Выводим имена детей, раздлеяя их двумя пробелами
for (int i = 0; i < this->children.size(); ++i){
                cout << "  ";
                cout << this->children[i]->get_name();
        }

        //  Вызываем аналогичную функцию у детей
bool w = false;
        for (int i = 0; i < this->children.size(); ++i){
if (this->children[i]->children.size() && !w){
cout << "\n";                          w = true;
                }
                this->children[i]->print_tree();
        }
return; }
```

```
void Base::print_status_tree(){
        cout << "The object " << this->name;
        //  Проверяем, готов ли объект, и выводим соответствующую информацию
if (this->status > 0){                  cout << " is ready";
        }else{
                cout << " is not ready";
        }

        if (this->children.size()){
cout << "\n";
        }

        //  Выводим имена детей, раздлеяя их двумя пробелами
for (int i = 0; i < this->children.size(); ++i){
this->children[i]->print_status_tree();                 if
(i != this->children.size() - 1)
cout << "\n";
        }
return; }


void Base::print_tree_view(int deep_lvl=0){

        //  Количество пробелов для отступа
int indent = deep_lvl * 4;
        //  Выводим отступ        for(int
i = 0; i < indent; ++i){              cout
<< " ";
        }
        //  Вывод имени         cout
<< this->name;           if (!this-
>children.size())
                return;
cout << "\n";
        //  Вызов функции для детей данной вершины         for
(int i = 0; i < this->children.size(); ++i){
this->children[i]->print_tree_view(deep_lvl + 1);
if (i != this->children.size() - 1){
                        cout << "\n";
                }
        }
return; }


void Base::add_child(Base * h){
this->children.push_back(h);
return;
}


Base * Base::get_by_name(string name){
if (this->name == name){
return this;
        }
        for (int i = 0; i < this->children.size(); ++i){
                        Base * b = children[i]->get_by_name(name);
if (b != NULL)                                  return b;
```

```
        }
        return nullptr;
}
```

Файл base.h

```
#ifndef BASEH
#define BASEH
#include <vector>
#include <string>
class Base;

class Base{ private:
std::string name;
        Base * parent;
int status; protected:
        std::vector<Base *> children;
public:
```

```
        Base(Base * parent, std::string name, int status);
void set_name(std::string new_name);
        std::string get_name();

        void set_parent(Base * new_parent);
        Base * get_parent();

        void print_tree();         void
print_status_tree();         void
print_tree_view(int deep_lvl);

        void add_child(Base * h);

        Base * get_by_name(std::string name);
};

#endif
```

Файл b.cpp

```
#include <string>
#include "b.h"
#include "base.h"
B::B(Base * parent, std::string name, int status): Base(parent, name, status)
{};
```

Файл b.h

```
#ifndef BH
#define BH
#include <string>
#include "base.h"
        class B:
public Base{ public:
        B(Base * parent, std::string name, int status);
};

#endif
```

Файл c.cpp

```
#include <string>
#include "c.h"
#include "base.h"
C::C(Base * parent, std::string name, int status): Base(parent, name, status)
{};
```

Файл c.h

```
#ifndef CH
#define CH
#include <string>
#include "base.h"
        class C:
public Base{ public:
        C(Base * parent, std::string name, int status);
};

#endif
```

Файл d.cpp

```
#include <string>
#include "d.h"
#include "base.h"

D::D(Base * parent, std::string name, int status): Base(parent, name, status)
{};
```

Файл d.h

```
#ifndef DH
#define DH
#include <string>
#include "base.h"
        class D:
public Base{ public:
        D(Base * parent, std::string name, int status);
};
```

```
#endif
```

Файл e.cpp

```
#include <string>
#include "e.h"
#include "base.h"

E::E(Base * parent, std::string name, int status): Base(parent, name, status)
{};
```

Файл e.h

```
#ifndef EH
#define EH
#include <string>
#include "base.h"
        class E:
public Base{ public:
        E(Base * parent, std::string name, int status);
};

#endif
```

Файл main.cpp

```
#include "app.h"
        int
main(){
        App app(NULL);
app.build_tree();
app.run(); }
```

Тестирование

Входные данные	Ожидаемые выходные данные	Фактические выходные данные
root root obj1 2 3 obj1 obj2 6 4 obj1 obj3 5 -1 root obj2 2 2 obj3 obj7 4 4 endtree	Object tree root obj1 obj2 obj3 obj7 obj2	Object tree root obj1 obj2 obj3 obj7 obj2
ro ro a 4 3 a b 6 7 ro k 4 5 a t 3 -1 a c 3 32 endtree	Object tree ro a b t c k	Object tree ro a b t c k