Методические указания к лабораторной работе № 7 по курсу Программирование на основе классов и шаблонов

"Виртуальные функции и виртуальные классы"

СОДЕРЖАНИЕ

1. Цель работы

Целью лабораторной работы изучение механизмов динамического связывания. Студенты изучают понятия абстрактного класса, виртуальных функций и виртуальных классов. Они осваивают на практике по индивидуальным заданиям использование этих прием при программировании в среде C++.

2. Задачи, решаемые в лабораторной работе

В процессе выполнения ЛР студенты индивидуально должны выполнить следующие задачи. Перечислим основные требования к лабораторной работе:

Прочитать и усвоить раздел данных МУ – “ 3**. Основные понятия**” (Это желательно сделать до начала ЛР).

- Выполнить все задания из раздела “4. **Порядок выполнения работы**“:
- Создать в VS консольный проект.
- Обеспечить русификацию ввода и вывода с консоли.
- Построить структуру классов для виртуального вызова
- Проверка описания объекта абстрактного класса
- Проверка классов – описание объектов
- Проверка вызова функции через объекты
- Проверка вызова функции через указатель
- Проверка виртуального вызова функции
- Занесение объектов в список трех типов
- Распечатка списка объектов с помощью виртуального вызова
- Удаление одного объекта по варианту и распечатка списка
- Виртуальные классы
- Построить диаграммы классов для варианта
- Оформление отчета

Все действия по программированию выполняются в интерактивном режиме с использованием отладчика.

После выполнения перечисленных действий студенты: демонстрируют работу программы, оформляют на основе шаблона отчет по лабораторной работе и защищают ее, отвечая на контрольные вопросы, представленные в данных методических указаниях.

Примечание 1. Для удобства восприятия текста используется цвет и тип шрифта. Фрагменты исходного текста, включаемые в программы, я буду выделять синим цветом. Например, вставка заголовочного файла в программу будет выглядеть так:

```
#include <iostream>
```

2. Вывод результата в консольное окно, который формируется программой, будем помечать коричневым цветом и устанавливать непропорциональный шрифт **Courier New**. Например:

```
Введите iVal: 10
```

3. Формализованные описания языка и синтаксические правила будем записывать зеленым цветом:

<левая часть выражения присваивания> = <правая часть выражения присваивания>;

4. Если в тексте встречается переменная, которая подчеркнута, то это означает, что дается определение важного понятия и это понятие встречается в данном тексте первый раз. Например:

Программа – это упорядоченная совокупность операторов …

3. Основные понятия и примеры

3.1. Универсальное программирование

Как сделать программу более наглядной, универсальной и компактной одновременно? Этот вопрос очень важен при построении программных систем. При построении сложных программных систем применяются разные методы и механизмы для обеспечения такого результата. Наглядность и компактность, а также универсальность, способствуют сокращению сроков разработки и отладки программного обеспечения (ПО), в значительной мере влияют на его качество (надежность ПО, сопровождение ПО, мобильность ПО) и обеспечивают

увеличение сроков службы ПО. В принципе возможны два подхода достижения этих результатов: статическое и динамическое связывание.

Связывание в программировании означает внутреннюю настройку программы на функции и переменные. Другими словами, это определение того, на каком этапе формирования программы, эти связи устанавливаются. Принято рассматривать три возможных этапа: этап работы макропроцессора, этап компиляции и этап выполнения программ. Первые два этапа, обычно, называют статическим связыванием (связи можно изменять только до компиляции). Третий способ обеспечивает динамическое связывание, которое обеспечивается на этапе выполнения программы. Хотя последний способ является более сложным в реализации, он обеспечивает в большей степени универсальность и компактность программ. Простым примером динамического связывания является использование указателей и динамической памяти в программах.

Вопросы статического связывания были рассмотрены ранее в лабораторных работах, посвященных: перегрузке функций и операций, использования макрокоманд и т.д. В этой ЛР мы будем изучать механизмы динамического связывания.

3.2. Динамическое связывание

При динамическом связывании решение о том, какая функция будет вызываться в данный момент или какие данные будут использоваться в программе, переносится на поздний этап. Это этап выполнения программы, что, несомненно, является более гибким вариантом. Например, при простом вызове функции в программе мы должны дать перечень фактических параметров, которые будут использоваться. Настройки вызова выполняются при компиляции. Для изменения значений параметров необходимо сделать новую компиляцию программы. Если в качестве параметров использовать указатели, то их значения могут вычисляться в процессе работы программы и перекомпиляции не требуется. Это простейший пример динамического связывания. Использование указателей мы рассматривали ранее, поэтому здесь уделим больше внимания другим механизмам динамического связывания: виртуализации.

Когда в программе существует большое число разнообразных объектов, возникает проблема использования множества функций для каждого типа объектов. Она заключается в том, что имена однородных функций могут совпадать для разных типов объектов. Эта проблема решается частично с помощью механизма перегрузки функций, хотя эта технология относится к статическому связыванию. В тоже время, хотелось бы определять тип и алгоритм функции для каждого типа объекта автоматически. Например, перемещение (`Move`) по “графическому полотну” экрана объекта - линии (`Line`) и других графических объектов алгоритмически различается, но выбор метода для этих действий выполнять, в принципе, должен выполнить все-таки программист, придумав механизм распознавания типа конкретного объекта. Это характерно и для тех случаев, когда множества разнородных объектов размещаются в одном контейнере (например, в списке) и притом там трудноразличимы.

Для автоматического распознавания выполняемых функций для разных объектов используется механизм виртуальных функций, базирующийся на принципе иерархического наследовании классов. Перед тем, как рассмотреть виртуальные функции, рассмотрим то, как это может быть выполнено вручную.

3.3. Ручное динамическое связывание

Для вызова конкретной функции для объектов из контейнера (или других наборов объектов) необходимо иметь возможность определить тип или класс объекта. Для этого среди свойств класса таких объектов нужно зарезервировать такое переменную-член объекта, которое однозначно его идентифицирует, например, назовем её `ClassType`. Тогда для выполнения операцию распознавания функции для выполнения (например, метод `Move` – переместить объект) необходимо, с помощью переключателя конкретизировать вызов, опираясь на объект или указатель. Для уточнения объекта, в этом случае, используется эта вспомогательная переменная (`ClassType`). Для примера рассмотрим простую систему классов следующего вида:

```
// Базовый класс
```

```
class Bas {
public:
	int a;
	int ClassType;
	Bas(int x = 0) : a(x), ClassType(0) {} // Конструктор
};
// Производный класс 1
class Type1 : public Bas {
public:
	int Coord;
	Type1(int x=0) : Coord(x) { ClassType = 1; }    // Конструктор
	void Move(int Delta) { Coord += Delta; } // Метод для динамического вызова
};
// Производный класс 2
class Type2 : public Bas {
public:
	int Coord;
	Type2(int x = 0) : Coord(x) { ClassType = 2; }    // Конструктор
	void Move(int Delta) { Coord -= Delta; }   // Метод для динамического вызова
};
```

Тогда нужно выполнить разделение (переключение) по типам объектов. Пусть указатель `pObj` определяет текущий объект, тип которого в данном контексте возможно неизвестен (например, объект взят из контейнера). Тогда? используя переменную `ClassType` для управления переключателем, мы сможем корректно сделать типизацию и вызов универсальной функции (`Move`). Например:

```
// Ручная виртуализация
Bas * pObj;
Type1 T1(10);
Type2 T2(20);
// Задание значения указателя для T1
pObj = &T1;
cout << "T1 = " << T1.Coord << " T2 = " << T2.Coord <<endl;
//При  pObj->Move(); Ошибка, так как указатель па базовый класс, а так Move нет!
switch (pObj->ClassType)
{
case 1:
	((Type1 *)pObj)->Move(5); break; // Типизация указателя для Type1 (Type1 *)pObj
case 2:
	((Type2 *)pObj)->Move(5); break; // Типизация указателя для Type2(Type2*)pObj
}
cout << "T1 = " << T1.Coord << " T2 = " << T2.Coord <<endl;
// Задание значения указателя для T2
pObj = &T2;
switch (pObj->ClassType)
{
case 1:
	((Type1 *)pObj)->Move(5); break;
case 2:
	((Type2 *)pObj)->Move(5); break;
}
cout << "T1 = " << T1.Coord << " T2 = " << T2.Coord <<endl;
```

В результате выполнения этого фрагмента получим:

```
T1 = 10 T2 = 20
T1 = 15 T2 = 20
T1 = 15 T2 = 15
```

Типизация указателя (см. cast выражение для `Type1`, `Type2`) здесь необходима, так как компилятор не может однозначно по одному указателю определить тип объекта и вызываемой функции соответственно. Кроме того, в данном примере важно, чтобы поле `ClassType` должно быть объявлено в базовом классе и в открытой его части (`public`). Пример показывает очевидные недостатки ручного связывания: изменение набора объектов требует модернизации

программы (изменения исходного текста – позиций переключателя), число элементов переключателя может быть велико, в каждом случае с добавлением новых типов в проект придется организовывать новое переключение. Для упрощения доступа к таким функциям используется механизм виртуальных функций, который будет рассмотрен ниже.

3.4. Виртуальные функции

Виртуальная функция - это обычный функция член класса (метод класса), которая содержит перед описанием спецификатор `virtual`. Этот спецификатор должен стоять в описании класса перед прототипом функции или описанием функции:

```
virtual <прототип функции члена класса>;
virtual <описание функции члена класса> {< Тело функции>};
```

Примеры описания виртуальных функций:

```
class ...{
public:
...
        virtual void Print();  // Прототип виртуальной функции
        virtual int Summ(int a , int b) { ... } // Описание виртуальной функции
};
```

Отметим, что при описании виртуальных функций вне класса, повторно ключевое слово `virtual` указывать не надо, например:

```
class Name {
public:
        Name(){ a = 5;}
        virtual void Print()  // Прототип виртуальной функции
        int a;
};
void Name::Print()      // Задание здесь virtual будет ошибкой
{ cout << a<< endl; }
```

Описания виртуальных функций целесообразно производить в открытой части описания класса (`public`) для того, чтобы можно было обращаться к ним извне.

Использование виртуальных функций целесообразно только в цепочках наследования, в том случае, когда необходим динамический выбор из различных вариантов вызова методов.

3.5. Чистые виртуальные функции

Если функция определена так, что ее тело не задается (операторы для тела функции еще не известны или будут переопределяться) и ее описание не является фактически прототипом, то такая функция описывается специальным образом (“`=0`”) и она называется чистой виртуальной функцией. Формально это записывается так:

```
virtual <прототип функции члена класса> = 0;
```

Например:

```
virtual void Print() = 0;  // Чистая виртуальная функции
virtual int Summ(int a , int b) = 0; // Чистая виртуальная функция
```

Чистая виртуальная функция не имеет тела и не может быть поэтому быть вызвана для конкретных объектов. Чистые виртуальные функции используются в начале цепочки наследования классов, в первом базовом классе этой цепочки.

3.6. Абстрактный класс

Класс, в котором объявлена хотя бы одна чистая виртуальная функция, называется абстрактным классом. Пример:

```
// Абстрактный класс
class Sample {
public:
      int a;
      //…
      virtual int Summ(int a, int b) = 0; // Описание виртуальной функции
```

```
};
```

Объекты абстрактных классов создавать нельзя, так как не в полной мере определено поведение объекта (вызов методов). Поэтому описание, приведенное ниже, является ошибочным (компилятор в этом случае выдаст диагностическое сообщение):

```
Sample A; // Ошибка
pS1 = new Sample; // Ошибка
```

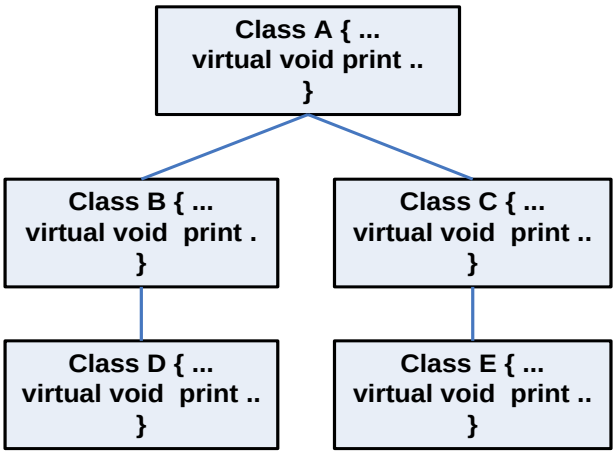
Однако в программе допускается объявлять и использовать указатели на объекты абстрактных классов (типов):

```
Sample * pS1; // Допускается указатель
```

При выполнении программы указатели на объекты наследники абстрактных классов преобразуются к указателям на базовые абстрактные классы, но в классах наследниках чистые виртуальные методы должны быть обязательно переопределены.

3.7. Структура наследования классов с виртуальными функциями

Для конструирования (создания механизма) виртуального вызова необходимо правильно построить структуру наследования классов. Такая структура должна минимально иметь два уровня наследования: минимум один базовый класс и несколько наследников. Число уровней наследования может быть произвольным. Наличие абстрактного класса в цепочке наследования тоже необязательно. Например, на диаграмме классов (показаны связи наследования):



Такой диаграмме классов соответствует следующее скелетное описание классов (без конкретизации тела класса):

```
/// Скелетная структура наследования
class A {
public:
     virtual void print()=0; // Описание чистой (необязательно) виртуальной функции
//…
};
class B: public A {
public:
//…
      void print() {} // Описание виртуальной функции
//…
      };
class C: public A {
//…
public:
      void print() {} // Описание виртуальной функции
//…
      };
```

```
class D: public B {
//…
public:
	void print() {} // Описание виртуальной функции
//…
	};
class E: public C {
//…
public:
	void print() {} // Описание виртуальной функции
//…
};
```

В данном примере пять классов и четыре разные функции `print` (функций с различной реализацией печати для объектов конкретного класса), одна описана как чистая виртуальная функция. Для объектов каждого типа будет вызываться функция со своей реализацией, только вызывать её следует через виртуальный вызов. Виртуальная функция в классе `A` объявлена как чистая виртуальная функция. Слово `virtual` достаточно указать перед прототипом виртуальной функции в базовом классе (см. пример выше).

3.8. Виртуальный вызов

Виртуальный вызов обеспечивает автоматическое определение нужной функции для вызова, в зависимости от типа объекта. Виртуальный вызов - это вызов виртуальной функции через указатель на базовый класс, где данная функция впервые объявлена как виртуальная. При этом цепочка описаний в классах наследниках данной функции с ключевым словом `virtual` не должна прерываться.

Рассмотрим уточненную структуру системы классов (`A`, `B`, `C`, `D`, `E`), приведенную выше. Для этого объявим конструкторы с параметром в каждом классе и переопределим виртуальный метод `print`.

```
// Описание классов для демонстрации виртуального вызова
class A {
public:
	int a;
	A(int k) : a(k) {} // Конструктор в абстрактом классе необходим!
	virtual void print() = 0; // Описание чистой (необязательно)
виртуальной функции
};
class B : public A {
public:
	B(int k) : A(k) {} // Конструктор B
	virtual void print()
	{
		cout << "B = " << a << endl;
	}
};
class C : public A {
public:
	C(int k) : A(k) {} // Конструктор C
	virtual void print()
	{
		cout << "C = " << a << endl;
	}
};
class D : public B {
public:
	D(int k) : B(k) {} // Конструктор D
	virtual void print()
	{
		cout << "D = " << a << endl;
```

```
	}
};
class E : public C {
public:
	E(int k) : C(k) {} // Конструктор E
	virtual void print()
	{
		cout << "E = " << a << endl;
	}
};
```

Пример виртуальных вызовов для нашей системы классов:

```
//Виртуальный вызов
A * pA; // указатель на базовый класс
B b(0);
C c(10);
D d(20);
E e(30);
pA = &b;
pA->print(); // виртуальный вызов 1
pA = &c;
pA->print(); // виртуальный вызов 2
pA = &d;
pA->print(); // виртуальный вызов 3
pA = &e;
pA->print(); // виртуальный вызов 4
```

При выполнении фрагмента программы мы получим следующий вывод:

```
B = 0
C = 10
D = 20
E = 30
```

Из вывода следует, что для каждого объекта вызываются виртуальные функции его класса.

Вызовы эти являются виртуальными, так как:

- имеется указатель на базовый класс (класс `A` – указатель `pA`);
- в нем функция `print` объявлена как виртуальная и впервые;
- в цепочках наследования (`A-B-D` и `A-C-E`) виртуальные описания функции `print` не прерываются (см. определение виртуального вызова).

В этом примере важно следующее: для первого и второго вызова используется одна и та же конструкция оператора `pA->print();` и не требуется специального переключателя с типизацией вызова методов. Такой обобщенный (динамический) вызов легко использовать в цикле, так как не надо изменять сам оператор вызова метода. Динамическое вычисление вызываемой функции определяется автоматически и неявно.

Виртуальные вызовы обеспечивают высокую универсальность программ, на основе механизма динамического связывания.

3.9. Виртуальные деструкторы

При создании объектов всегда в том или ином виде указывается тип объекта. Если объекты должны быть уничтожены, то тип также должен быть обязательно определен, так как должен быть выполнен деструктор объекта нужного класса (освободить память под объект). При ручном способе определения типа должен быть использован переключатель для определения типа. Так для системы классов, приведенной ниже, в объекте должен сохраняться тип класса (например, `ClassType`). Пусть у нас есть следующая система классов:

```
class A4 {
public:
	int a;
```

```
        int ClassType;
        A4() : a(0), ClassType(0){}
        ~A4() { cout << " Деструктор A4 " << endl; }
};
class B4 : public A4 {
public:
        int b;
        B4() : b(0) { ClassType = 1; }
        ~B4() { cout << " Деструктор B4 " << endl; }
};
class D4 : public B4 {
public:
        int d;
        D4() : d(0) { ClassType = 2; }
        virtual ~D4() { cout << " Деструктор D4 " << endl; }
};
```

Тогда попытка удалить объект без указания типа приведет к ошибке:

```
A4 *pA4 = new D4;
delete pA4; // Ошибка времени выполнения (тип не известен!)
```

Если тип задать явно, то удаление выполняется правильно:

```
cout << "Удаляем объект типа D4: " << endl;
delete (D4 *)pA4; // Правильно, но нужно типизировать
```

В общем случае для нашей системы классов можно использовать также и переключатель для удаления объектов, представленный ниже:

```
// Если тип задать явно, то удаление выполняется правильно:
A4 *pA4 = new B4;
cout << "Удаляем объект типа B4: " << endl;
// Переключатель по типу в классе
switch (pA4->ClassType)
{
case 0:
        delete (A4 *)pA4; break;
case 1:
        delete (B4 *)pA4; break;
case 2:
        delete (D4 *)pA4; break;
}
```

Получим результат после выполнения этого фрагмента программы:

```
Удаляем объект типа D4:
 Деструктор  D4
 Деструктор  B4
 Деструктор  A4
Удаляем объект типа B4:
 Деструктор  B4
 Деструктор  A4
```

Обратите внимание на тот факт, что при удалении объекта `D4` вызываются три деструктора в последовательности обратной порядку наследования. При удалении объекта `B4` – два деструктора.

Недостатки такого приема для обычных функций (с переключателем) были рассмотрены выше (нужна явная типизация указателя). Они аналогичны. Для удобного программирования и универсальности в C++ предусмотрен механизм виртуальных деструкторов. Для этого деструкторы должны быть описаны также с ключевым словом `virtual`, и вызываться через указатель на объект базового класса, где виртуальный деструктор в цепочке наследования объявлен впервые (сравните с определением виртуальной функции). Формальное описание виртуального деструктора имеет вид:

```
virtual  ~<Описание или прототип деструктора>;
```

В примере, расположенном ниже, объявлена система классов с цепочкой виртуальных деструкторов. Структура классов похожа на предыдущую систему классов (A5-B5-C5-D5):

```
/// ВИРТУАЛЬНЫЕ ДЕСТРУКТОРЫ
class A5 {
public:
	int a;
	A5(int k) : a(k) {}
	virtual ~A5() { cout << "a = " << a << " virtual Деструктор A5 " << endl; }
};
class B5 : public A5 {
public:
	B5(int k) : A5(k) {}
	virtual ~B5() { cout << "a = " << a << " virtual Деструктор B5 " << endl; }
};
class C5 : public B5 {
public:
	C5(int k) : B5(k) {}
	virtual ~C5() { cout << "a = " << a << " virtual Деструктор C5 " << endl; }
};
class D5 : public C5 {
public:
	D5(int k) : C5(k) {}
	virtual ~D5() { cout << "a = " << a << " virtual Деструктор D5 " << endl; }
};
```

После такого описания можно проверить, как работает механизм виртуальных деструкторов. Опишем указатели на динамические объекты и тут же их удалим:

```
	A5 *pA5; // Указатель на базовый класс
	cout << "A5:" << endl;
	pA5 = new A5(1);
	delete pA5; // Вызов виртуального деструктора Одинаково!
	cout << "B5:" << endl;
	pA5 = new B5(2);
	delete pA5; // Вызов виртуального деструктора Одинаково!
	cout << "C5:" << endl;
	pA5 = new C5(3);
	delete pA5; // Вызов виртуального деструктора Одинаково!
	cout << "D5:" << endl;
	pA5 = new D5(4);
	delete pA5; // Вызов виртуального деструктора Одинаково!
```

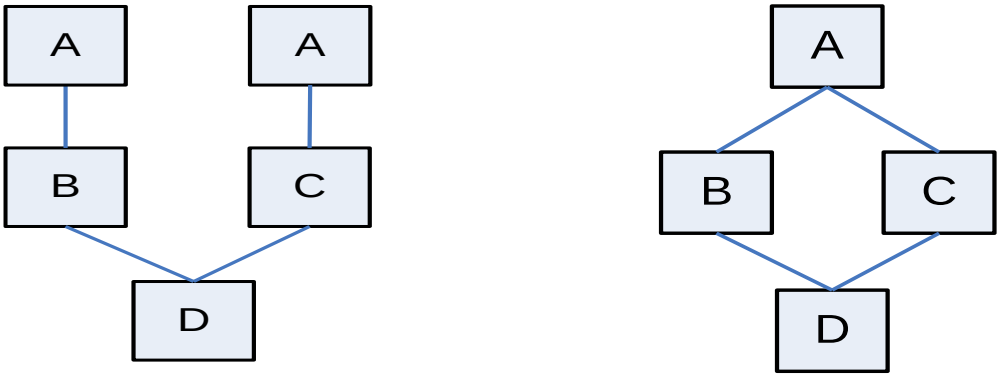
Фактически мы имеем виртуальный вызов деструктора. Результаты работы приведенного фрагмента показаны ниже:

```
A5:
a = 1 virtual Деструктор A5
B5:
a = 2 virtual Деструктор B5
a = 2 virtual Деструктор A5
C5:
a = 3 virtual Деструктор C5
a = 3 virtual Деструктор B5
a = 3 virtual Деструктор A5
D5:
a = 4 virtual Деструктор D5
a = 4 virtual Деструктор C5
a = 4 virtual Деструктор B5
a = 4 virtual Деструктор A5
```

Базовый указатель pA5 один, с помощью его удаляются объекты разных классов, причем записи для удаления одинаковые, а типизация удаляемых объектов является встроенной (неявной).

3.10. Проблемы множественного наследования. Виртуальные классы.

При множественном наследовании (наследовании из нескольких базовых классов) возможны ситуации, когда один и тот же базовый класс появляется несколько раз в цепочке наследования. В этом случае, если такая конструкция специально не предусмотрена, может возникнуть неоднозначность. Для исключения возможности дублирования используется механизм виртуальных классов. Несмотря на схожесть названий, этот прием не имеет ничего общего с механизмом виртуальных функций. Для иллюстрации дублирования рассмотрим следующие диаграммы классов:



В первой диаграмме (левый рисунок) часть в виде объекта A включается дважды в итоговый объект D. Если такая конструкция специально не предусмотрена (конструкция объектов такова, что дублирование необходимо), то может возникнуть неоднозначность. Для исключения неоднозначности подойдет такая конструкция объекта, которая представлена на правом рисунке. Такой вариант возможен при использовании виртуальных классов, которое будет рассмотрено в следующем разделе. Здесь рассмотрим ручной способ исключения неоднозначности. Пусть классы для левой диаграммы имеют следующее описание (для удобности восприятия обозначим их A1, B1, C1 и D1 соответственно):

```
///////// Классы без виртуальности
class A1 {
public:
	int a;
	A1(int x) : a(x) {}
};
class B1 : public A1 {
public:
	int b;
	B1(int x, int y) : A1(x), b(y) {}
};
class C1 : public A1 {
public:
	int c;
	C1(int x, int y) : A1(x), c(y) {}
};
class D1 : public B1, public C1 {
public:
	int d;
	D1(int x, int y, int z, int q) : B1(x, y), C1(x + 50, z) { d = q; }
};
```

Отметим, что для конструктора D1 с параметрами нет необходимости явно вызывать конструктор базового класса A1, так как он вызывается из конструкторов B1 и C1

соответственно. Теперь, если мы опишем один объект класса D1, то можем проверить какие значения переменной класса A1 – “a” можем получить при разной адресации с помощью операций разрешения области видимости (“::”).

```
D1 d101(1,2,3,4);
//Доступ к параметрам
cout << "После создания: " << endl;
//cout << "d101.a = " << d101.a << endl; // ошибка компиляции a в двух классах наследниках!
cout << "d101.b = " << d101.b << endl;
cout << "d101.c = " << d101.c << endl;
cout << "d101.d = " << d101.d << endl;
// Проверка члена данных "a"
 cout << "d101.B1::a = " << d101.B1::a << endl;
 cout << "d101.C1::a = " << d101.C1::a << endl;  // Разные при ссылках на a
d101.B1::a = 5; // Изменяем
d101.C1::a = 15; // Изменяем
cout << "После изменения a в B1(d101.B1::a = 5) и C1(d101.C1::a = 15): " << endl;
 cout << "d101.B1::a = " << d101.B1::a << endl;
 cout << "d101.C1::a = " << d101.C1::a << endl;  // Разные при ссылках на a
 cout << "d101.A1::a = " << d101.A1::a << endl; // Берем из класса, который описан первым(B1)!
 cout << "d101.C1::A1::a = " << d101.C1::A1::a << endl;
```

Во-первых, прямая адресация через объект типа D1 к полю d011.a недопустима, так как неизвестно какой из вложенных объектов здесь используется! При явном указании класса B1 или C1, их значения будут разными (B1 - 1 и C1- 51 соответственно). Для прямого доступа к полю “a” используется операция разрешения области видимости (“::”).

```
После создания:
d101.b = 2
d101.c = 3
d101.d = 4
d101.B1::a = 1
d101.C1::a = 51
После изменения a в B1(d101.B1::a = 5) и C1(d101.C1::a = 15):
d101.B1::a = 5
d101.C1::a = 15
d101.A1::a = 5
d101.C1::A1::a = 15
```

Во-вторых, если указана явная адресация с классом A1, то рассматривается класс B1, который в цепочке наследования для D1 представлен первым. Если указана двойная адресация. Например, если зададим - C1::A1, то доступ будет получен к тому подобъекту, который явно указан. Основная проблема примера заключается в том, что подобъекты класса A1 дублируются. В большинстве программных реализаций такое дублирование должно быть исключено. Для этого используется механизм виртуальных базовых классов, описанный ниже.

3.11. Виртуальные базовые классы

Диаграмме второго вида соответствует описание, в котором перед именем базового класса ставиться специальное ключевое слово virtual. Такой базовый класс называется виртуальным базовым классом. Признак виртуальности должен быть установлен перед каждым виртуальным базовым классом. Формально это можно записать так:

```
class <имя класса> : virtual <базовый класс> …{ <Описание класса> };
```

Описание системы классов с виртуальным классом A3 приведено ниже.

```
/// ВИРТУАЛЬНЫЕ КЛАССЫ
class A3 {
public:
      int a;
      A3(int x) : a(x) {}
};
class B3 : virtual public A3 {
```

```
public:
      int b;
      B3(int x, int y) : A3(x), b(y) {}
};
class C3 : virtual public A3 {
public:
      int c;
      C3(int x, int y) : A3(x), c(y) {}
};
class D3 : public B3, public C3 {
public:
      int d;
      D3(int x,int y,int z,int q) : A3(x), B3(x+10,y), C3(x+100,z) {d = q;}
};
```

Отметим, что по умолчанию первым будет вызван конструктор виртуального класса без параметров, если он не задан явно. Использование и вызов конструктора виртуального класса выполняется обязательно. В нашем примере он вызывается явно `A3(x)`. Причем явный или неявный вызов конструктора виртуального класса выполняется всегда в начале цепочки конструкторов. Опишем объект класса `D3` и `V101` и проверим доступ к параметру `a` подобъекта класса `A3`.

```
// Виртуальные базовые классы
// (virtual A3)-B3 , (virtual A3)-C3, B3|C3-D3  с виртуальностью
D3 V101(1,2,3,4); // Объект с виртуальными классами
cout <<"V101.a =" << V101.a<<endl;//Обращение к полю a класса (A3)как обычно в классе D3
cout << "V101.A3::a = " << V101.A3::a << endl;  // Через класс  A3 прямо
cout << "V101.B3::a = " << V101.B3::a << endl;  // Через класс  B3 прямо
cout << "V101.C3::a = " << V101.C3::a << endl;  // Через класс  C3 прямо
cout << "V101.b = " << V101.b << endl;
cout << "V101.c = " << V101.c << endl;
cout << "V101.d = " << V101.d << endl;
```

Из распечатки результатов, расположенной ниже, видно, что только один параметр “a” доступен и изменяется в операторах присваивания. Допустимо также обращение к параметру через объект класса `D3` (`V101.a`). Результат будет таким:

```
V101.a = 1
V101.A3::a = 1
V101.B3::a = 1
V101.C3::a = 1
V101.b = 2
V101.c = 3
V101.d = 4
```

Таким образом, при сложной структуре наследования, исключается неоднозначность доступа и отсутствует необходимость указания прямой адресации параметров посредством операции разрешения области видимости. Допустимы и другие конструкции наследования с виртуальными классами. Если перед каждым наследованием этого повторяющегося класса стоит ключевое слово `virtual`, то доступ всегда будет к одному и тому же параметру.

3.12. Смешанное наследование (виртуальные и не виртуальные базовые классы)

Возможны и другие случаи наследования, когда один и тот же класс выступает как виртуальный класс, так и как невиртуальный в сложной иерархии наследования. Такой пример рассмотрен ниже (класс `A6`). В этом случае для корректного доступа необходима прямая квалификация доступа с помощью операции разрешения области видимости, например:

```
class A6 {
public:
      int a;
      A6(int x=0) { a = x; }
};
class B6 : virtual public A6 {
```

```
public:
	int b;
	B6(int x=0, int y=0) : A6(y), b(x) {}
};
class C6 : virtual public A6 {
public:
	int c;
	C6(int x=0, int y=0) : A6(y), c(x) {}
};
class E6 : public A6 {
public:
	int e;
	E6(int x = 0, int y = 0) : A6(y), e(x) {}
};
class D6 : public B6, public C6, public E6 {
public:
	int d;
	D6() : d(0) {}
	D6(int x, int y, int z, int v, int w) : B6::A6(w), B6(y, w), C6(z, w), E6(v, 2 * w)
	{
		d = x;
	}
};
```

В объекте типа D6 включаются виртуальные и не виртуальные классы одного типа (A6). Тогда для исключения ошибки указывается имя класса, в котором определено поле.

Проблема возникает для этой системы классов при вызове конструктора базового класса, который с одной стороны является виртуальным и не является виртуальным (A5). Для вызова этого конструктора необходимо указать явно имя порожденного класса, для которого он впервые объявлен как виртуальный. При этом конструктор базового класса будет вызываться дважды: как виртуальный – в начале, и как базовый для конструктора E5.

```
// Смешанные виртуальные классы
D6 d51(1,2,3,4,5);
 // d51.a = 100; // Ошибка компиляции
 cout << " d51.B6::a = " << d51.B6::a << endl;
 cout << " d51.E6::a = " << d51.E6::a << endl;
d51.C6::a = 500;
 cout << " d51.B6::a = " << d51.B6::a << endl;
 cout << " d51.E6::a = " << d51.E6::a << endl;
 cout << " d51.C6::a = " << d51.C6::a << endl;
 cout << " d51.A6::a = " << d51.A6::a << endl; // Берем из класса, который описан первым
 cout << " d51.B6::A5::a = " << d51.B6::A6::a << endl;
```

Прямое обращение через объект D5 является ошибочным, а значения доступные через E6 – 10, отличаются от значений виртуального базового класса A6, полученные через подобъекты C6 и B5.

```
d51.B5::a = 5
d51.E5::a = 10
d51.B5::a = 500
d51.E5::a = 10
d51.C5::a = 500
d51.A5::a = 500
d51.B5::A5::a = 500
```

Эти примеры можно проверить, вставив их в свой проект и получив в консольном окне свой вариант результатов.

3.13. Контейнеры

Контейнером называется специальная разновидность объектов, которые обеспечивают хранение других объектов, а также их совместную обработку (например, печать, изменение и т.д.). Контейнеры-объекты создаются на основе контейнерных классов, которые имеют специальную структуру, специальные свойства и специальные операции.

По размерности контейнеры подразделяются на следующие типы:

– С предопределенной размерностью (фиксированной) – например, стандартные массивы;
– С динамической размерностью – например, списки;
– С изменяемой размерностью – например, массивы с наращиваемой размерностью.

По типу доступа контейнеры подразделяются на следующие типы:

– С прямым доступом по индексу (номеру) – например, массивы;
– С прямым доступом по параметру (строка) – например, ассоциативные массивы;
– С последовательным доступом к элементам– например, списки;
– Со случайным доступом к элементам – например, множества.

По упорядоченности контейнеры подразделяются на следующие типы:

– Упорядоченные контейнеры – например, массивы и списки;
– Неупорядоченные контейнеры – например, множества и мультимножества.

3.14. Контейнерные классы в VS

В системе программирования VS предусмотрено значительное разнообразие контейнерных классов для разных применений. Они содержатся в разных библиотеках:

– Библиотеке STL (`vector`, `list`, `map`, `stack`, `set`, `multiset`, `map`, `multimap` и `queue`);
– Библиотеке MFC (`CArray`, `CList`, `CMap`, `CObArray`, `CObList`, `CMapPtrToPtr` и др);
– Библиотеке ATL (`CAtlArray`, `CAtlList`, `CAtlMap` и др.);
– Библиотеке платформы .NET (`Array`, `List` и др);

В данной ЛР мы познакомимся с контейнерным классом типа список.

Контейнеры типа список обеспечивают работу с множеством объектов, число которых заранее неизвестно и может изменяться во время работы программы. Кроме того, порядок этих объектов может также изменяться, например, с добавлением новых объектов в произвольное место упорядоченного множества. Работа с массивами, особенно если их размерность велика, приводит к значительным временным затратам процессорного времени. Хотя прямого доступа к элементам списка не определяется (операция индексирования недоступна - просмотр списка возможет только последовательно), динамические возможности перемещения по списку, его изменения, поиска и сортировки выполняются значительно эффективнее, чем в массивах. Контейнерные объекты типа список применяются практически во всех программах средней и большой сложности. На основе списков строятся более сложные структуры данных: деревья, сетевые структуры данных и многие другие.

3.15. Работа с объектами класса `list` (STL).

В библиотеке **STL** (напомним, что это – библиотека шаблонных классов - `Standard Template Library`) описывается шаблонный класс `list`, который позволяет описывать списки переменных любого типа. Формальное определение шаблона дано ниже:

```
template <class Type, class Allocator=allocator<Type>> class list{ … };
```

Для работы объектами этого класса необходимо подключить заголовочный файл:

```
#include <list>
```

Для описания списков предусмотрены разные конструкторы. Примеры описания списков даны ниже:

```
// Списки list описания и инициализация
list <int> l0; // Пустой список l0
list <int> l1(3); // Список с тремя элементами равными 0
list <int> l2(5, 2); // Список из пяти элементов равными 2
list <int> l3(l2); // Список l3 на основе списка l2
list <int>::iterator l3_Iter; // Описание итератора l3_Iter
l3_Iter = l3.begin();  // Итераторные вычисления на начало
l3_Iter++; l3_Iter++; // Итераторные вычисления продвинуть на два объекта
list <int> l4(l3.begin(), l3_Iter); // Список l4 на основе первых двух элементов L3
```

Функцию печати опишем в заголовочном файле так, чтобы в ней в данной функции

использован прямой итератор для класса list, а для индексации при печати элементов списка использовалась вспомогательная целая переменная:

```
void lPrint(list<int>& l)
{
	list<int>::iterator iter;
	int i = 0;
	if (!l.empty())
	{
		for (iter = l.begin(); iter != l.end(); iter++, i++)
		{
			cout << "l[" << i << "] = " << *iter << endl;
		}
	}
	else
		cout << "Список пуст!" << endl;
}
```

Для описания пустого списка l (типа list) также нужно указать тип int (инстанцировать шаблонный объект):

```
list <int> l; // Пустой список l
cout<< "Добавление:" << endl;
l.push_back( 1 ); // Добавление в конец списка
l.push_back( 2 ); // Добавление в конец списка
l.push_front( 5 ); // Добавление в начало списка
lPrint(l); // Собственная функция печати целого списка
```

В функции печати используется итератор iter, который позволяет продвигаться по списку (iter++). Для установки итератора на первый элемент используется метод begin. Остановка просмотра проверяется методом end. Для проверки пустого списка используется метод empty. Для дальнейшей демонстрации возможностей списков будем использовать эту функцию. В первом фрагменте получим результат:

```
Добавление:
l[0] = 5
l[1] = 1
l[2] = 2
```

Перечень методов класса список (list) приведен в разделе Справочные материалы, подробное описание методов вы можете найти в документации, литературе и справочной системе MSDN[5].

3.16. Отладка программ.

В данной лабораторной работе отладчик имеет особое значение. С его помощью мы можем проверить вызов типов перегруженных методов и задание параметров по умолчанию.

При разработке программ важную роль играет отладчик, который встроен в систему программирования. В режиме отладки можно проверить работоспособность программы и выполнить поиск ошибок самого разного характера. Отладчик позволяет проследить ход (по шагам) выполнения программы и одновременно получить текущие значения всех переменных и объектов программы, что позволяет установить моменты времени (и операторы), в которые происходит ошибка и предпринять меры ее устранения. В целом, отладчик позволяет выполнить следующие действия:

– Запустить программу в режиме отладки без трассировки по шагам (**F5**);
– Выполнить программу по шагам (**F10**);
– Выполнить программу по шагам с обращениями к вложенным функциям(**F11**);
– Установить точку останова (**BreakPoint** – **F9**);
– Выполнить программу до первой точки останова - breakpoint (**F5**);
– Просмотреть любые данные в режиме отладки в специальном окне (**locals**);
– Просмотреть любые данные в режиме отладки при помощи наведения на них мышки;
– Изменить любые данные в режиме отладки в специальном окне (**locals** и **Watch**);
– Установить просмотр переменных в специальном окне (**Watch**);
– Остановить и выключить отладку при необходимости;
– Просмотреть последовательность и вложенность вызова функций.

При выполнении всех лабораторных курса студенты должны продемонстрировать использование отладчика VS, знать все его возможности, уметь воспользоваться технологиями отладки и отвечать на все контрольные вопросы, связанные с отладкой и тестированием программ (см. раздел контрольных вопросов).

Примечание. Подробное описание материала и понятий вы можете найти в литературе [1 - 6] или справочной системе MS VS [2].

4. Порядок работы и методические указания (основные требования)

4.1.Создать и русифицировать консольный проект в VS.

ПРОЕКТ VS. Последовательность действий для создания консольного проекта в VS описана в методических указаниях к ЛР № 1 для данного курса (ПКШ), смотрите на сайте дисциплины. Русификация ввода и вывода информации в проекте для консольного режима выполняется на основе приемов, также показанных в методических указаниях к ЛР №1.

4.2. Варианты для выполнения заданий ЛР

ВАРИАНТЫ. До начала выполнения ЛР необходимо познакомиться с таблицей вариантов (см. ниже) и выполнять задание следующих пунктов в соответствии со своим вариантом.

4.3. Проверить работу вызова функций при ручном динамическом связывании

РУЧНОЕ СВЯЗЫВАНИЕ. Описать два класса (`Type1` и `Type2`) на основе материала теоретической части данных МУ и базовый класс `Bas`. Описать два объекта классов `T1` и `T2` и указатель для `Bas - pObj`. Попеременно присвоить указателю значения адреса объектов и помощью переключателей для выполнения детализации типа вызываемой функции (см. в теоретической части) продемонстрировать вызов нужной функции **в пошаговом режиме отладчика** VS.

4.4. Построить структуру классов для изучения виртуального вызова методов

СОЗДАНИЕ СИСТЕМЫ КЛАССОВ. Нужно в своем заголовочном файле проекта (LR6.H) описать четыре класса: `Abstr - абстрактный`, `Deriv1`, `Deriv2` и `Deriv3` названия классов изменять нельзя. Связи наследования организованы следующим образом: `Abstr - абстрактный` класс, `Deriv1 - наследуется от Abstr`, `Deriv2 наследуется от Deriv1 и Deriv3 наследуется от Deriv1`. Во всех классах предусмотреть виртуальный метод – `print`, который будет использоваться для демонстрации виртуального вызова. В классе `Abstr` является чистой виртуальной функцией, а в классах `Deriv1`, `Deriv2` и `Deriv3` в нем записывается текст для идентификации вызываемого класса.

ОПИСАНИЕ КЛАССОВ. Классы `Deriv1`, `Deriv2` и `Deriv3` не являются абстрактными и имеют, по крайней мере, одну виртуальную функцию (`print`), определенную в классе `Abstr` в качестве чистой виртуальной функции (тело такой функции обозначается так - "`=0`"). В таблице, расположенной ниже, приведены основные сведения о создаваемых классах. Для каждой группы задаются отдельные групповые варианты, связанные с типом используемых данных. Применительно к данному пункту задания – это тип переменной класса (`long`, `float`, `int`, `double`).

Класс - имя	Тип класса	Тип члена	Тип метода	Название члена класса
`Abstr`	Абстрактный (наследник класса `CObject`)	Данное	`int (см.вар)`	`iPar(dPar, fPar, lPar)`
		Метод	Конструктор	`Abstr(int p)`
		Метод	Конструктор	`Abstr()`
		Метод – чистый, виртуальный	`void`	`print()`
`Deriv1`	Объектный (наследник `Abstr`)	Данное	`int (см.вар)`	`iDPar(dDPar, fDPar, lDPar)`
		Метод	Конструктор	`Deriv1 (int p)`
		Метод	Конструктор	`Deriv1 ()`

Класс - имя	Тип класса	Тип члена	Тип метода	Название члена класса
		Метод- виртуальный	void	print()
Deriv2	Объектный (наследник Deriv1)	Данное	int (см.вар)	iDPar(dDPar, fDPar, lDPar)
		Метод	Конструктор	Deriv2(int p)
		Метод	Конструктор	Deriv2()
		Метод- виртуальный	void	print()
Deriv3	Объектный (наследник Deriv1)	Данное	int (см.вар)	iDPar(dDPar, fDPar, lDPar)
		Метод	Конструктор	Deriv3(int p)
		Метод	Конструктор	Deriv3()
		Метод- виртуальный	void	print()

Виртуальный метод print может иметь вид:

```
virtual void print()  {  cout <<"Derv1 = "<< iDPar<< " Abstr = " << iPar << endl;}
```

4.5. Построить диаграмму классов для предыдущего пункта

ДИАГРАММА КЛАССОВ. Необходимо поострить диаграмму классов для системы классов, описанных выше, и отобразить ее в отчете в виде диаграммы классов. В диаграмме отобразить связи наследования.

4.6. Проверка описания объекта абстрактного класса

ПРОВЕРКА ОБЪЕКТОВ АБСТРАКТНОГО КЛАССА. Проверить в главной программе описание объектов для абстрактного класса Abstr. Результаты проверки занести в отчет, указав причину полученных результатов.

4.7. Проверка созданных классов – описание объектов

ОПИСАНИЕ ОБЪЕКТОВ. В главной программе описать указатель и три объекта для созданной системы классов. Проверить компиляцию и сборку программы. Пример:

```
Abstr *pAbs;
Derv1 d1(33);
Derv2 d2(44);
Derv3 d3(66);
```

4.8. Проверка вызова функции через объекты

ВЫЗОВЫ МЕТОДОВ ЧЕРЕЗ ОБЪЕКТ. Проверить в главной программе вызов функции **print** для созданных объектов через объект (для всех объектов), например:

```
d2.print();
```

Результаты вызова скопировать в отчет.

4.9. Проверка вызова функции через указатель на объект

ВЫЗОВЫ МЕТОДОВ ЧЕРЕЗ УКАЗАТЕЛЬ НА ОБЪЕКТ. Проверить в главной программе вызов функции print для созданных объектов через указатель (для всех объектов), например:

```
Derv1 *pD2 = &d2;
pD2 ->print();
```

Результаты вызова скопировать в отчет.

4.10. Проверка виртуального вызова функции

ВИРТУАЛЬНЫЙ ВЫЗОВ. Проверить в главной программе вызов функции **print** для созданных объектов через указатель на базовый класс (для всех трех объектов) – виртуальный вызов, например:

```
pAbs = &d2;
pAbs->print();
```

В пошаговом режиме отладчика убедиться, что при виртуальном вызове производиться вызов функции, соответствующей нужному типу объекта. Результаты вызова скопировать в отчет.

4.11. Описание объекта списка типа `list` библиотеки `STL`

ОПИСАНИЕ ОБЪЕКТА СПИСОК. Описать объект класса `list` с названием `ListDer`. В справочной системе VS познакомиться с основными методами этого класса: `push_back`, `push_front`, `pop_back`, `pop_front`, `clear`, `empty`, `insert`, `erase`, `reverse`, `sort`, `size`, `remove` и др. (см.Приложение)

4.12. Занесение объектов в список трех типов

ФОРМИРОВАНИЕ СПИСКА. Занести с помощью трех циклов в список `ListDer` объекты 3-типов для наших классов (`Deriv1`, `Deriv2` и `Deriv3`). Число заносимых объектов каждого типа минимум три объекта. Тип метода определяется вариантом. Объекты создать динамически (`new`), определить их разное содержание для основного класса и базового абстрактного класса.

4.13. Распечатка списка объектов с помощью виртуального вызова

РАСПЕЧАТКА СПИСКА. В главной программе с помощью одного цикла распечатать список с использованием виртуального вызова метода `print()`. Для организации цикла создать итератор. Создать шаблон функции печати списка и осуществить его вызов для списка `ListDer`.

4.14. Удаление одного элемента списка по варианту

Удалить элемент списка по варианту. Вновь распечатать список, используя функцию.

4.15. Виртуальные деструкторы

Описать во всех классах (`Abstr`, `Deriv1`, `Deriv2` и `Deriv3`) виртуальные деструкторы (см. выше) и в одном цикле обеспечить удаление объектов всего списка `ListDer`. В деструкторах сделать печать идентифицирующую тип класса. Результаты удаления занести в отчет.

4.16. Виртуальные классы

ВИРТУАЛЬНЫЕ КЛАССЫ. Построить новую структуру классов с виртуальными классами, на примере, рассмотренном выше. Структура должна быть сходна с описанием, из раздела “Основные понятия”. Буквенные имена классов заданы в таблице групповых вариантов. Типы переменных в классах должны соответствовать таблице вариантов группы. Продемонстрировать использование виртуальных классов с помощью распечатки любой переменной виртуального базового класса.

4.17. Дополнительные требования для сильных студентов (д.т.)

Для сильных студентов предлагаются дополнительные требования при выполнении ЛР № 6. Эти требования могут быть выполнены в любой последовательности. На титульном листе отчета по ЛР необходимо указать, что все дополнительные требования выполнены. Нужно:

- Удалить из списка элемент по номеру, указанному в таблице вариантов. Список после удаления повторно распечатать.
- Проверить механизмы виртуальных классов для смешанного варианта виртуальных базовых классов и не виртуальных базовых классов. Нарисовать диаграмму классов. Результаты и выводы поместить в отчет.
- Проверить сохранение или отсутствие механизма виртуального вызова функций в том случае, если цепочка описаний одной функции с ключевым словом `virtual` временно прерывается в середине дерева наследования классов.

5. Варианты по группам и студентам

Ниже представлена таблица вариантов. В данной ЛР не назначаются индивидуальные варианты для каждого конкретного студента.

Вар №	Группа	Типы переменной для класса	Номер удаляемого элемента (д.т.)	Занесение в список	Виртуальные классы - имена
1.	ИУ5 – 21	`int`	3	Хвост	O,P,Q,R
2.	ИУ5 – 22	`float`	2	Хвост	I , J, K ,L
3.	ИУ5 – 23	`double`	5	Голова	Y,U,Z,X
4.	ИУ5 – 24	`int`	6	Хвост	A,F,C,D
5.	ИУ5 – 25	`double`	3	Голова	O,P,Q,R
6.	СУЦ	`long`	1	Хвост	E, F, G, H

6. Ошибки и их запоминание

Для накопления профессионального опыта в программировании рекомендуется запоминать и фиксировать ошибки, возникающие на различных стадиях разработки программ: разработки алгоритмов, подготовки текста, компиляции и отладки программ. С этой целью вводится требование размещения в отчете по ЛР фиксации ошибок в специальной таблице (см. шаблон отчета по ЛР в конце документа). При этом запоминается: тип и суть ошибки, этап возникновения и способ устранения. Такая работа является очень полезной и позволяет избавиться от ошибок и легче находить способы их устранения.

7. Контрольные вопросы

1. Что такое проект в VS? Для чего нужны проекты и в чем их преимущество?
2. Поясните структуру объектов программы по диаграмме объектов?
3. Что такое переменная этапа компиляции? Как она описывается?
4. Что такое **абстрактный** класс? Дайте определение. Пример.
5. Почему нельзя создавать объекты абстрактных классов?
6. Что такое **виртуальная** функция (ВФ)? Чистая ВФ? Пример.
7. Что такое **виртуальный** вызов?
8. В чем преимущества виртуального вызова для программистов?
9. Что такое чистая виртуальная функция? Покажите ее в вашей программе.
10. Зачем для виртуального вызова нужен указатель на базовый класс?
11. Как влияет использование виртуального вызова на размер программ?
12. Как влияет использование виртуального вызова на изменяемость программ?
13. Что такое **виртуальный** класс?
14. Что такое **виртуальный** деструктор?
15. Какая проблема решается при использовании виртуальных классов?
16. Чем отличается статическое и динамическое связывание?
17. На каком этапе разработки и выполнении программ происходит статическое связывание?
18. На каком этапе разработки и выполнении программ происходит динамическое связывание?
19. В чем состоят преимущества динамического связывания, и как они могут использоваться в программах?
20. Какие действия можно предпринять во время отладки программы с классами?

8. Литература

1. Г. Шилдт “C++ Базовый курс”: Пер. с англ.- М., Издательский дом “Вильямс”, 2011 г. – 672с
2. Г. Шилдт “C++ Руководство для начинающих”: Пер. с англ. - М., Издательский дом “Вильямс”, 2005 г. – 672с
3. Г. Шилдт “Полный справочник по C++”: Пер. с англ.- М., Издательский дом “Вильямс”, 2006 г. – 800с
4. Бьерн Страуструп "Язык программирования C++"- М., Бином, 2010 г.
5. MSDN Library for Visual Studio 2005 (Microsoft Document Explorer – входит в состав дистрибутива VS. Нужно обязательно развернуть при установке!)

6. Общее методическое пособие по курсу для выполнения ЛР и КЛР/ДЭ (см. на сайте 1-й курс www.sergebolshakov.ru) – см. кнопку в конце каждого раздела сайта!!!
7. Г.С.Иванова, Т.Н. Ничушкина, Е.К.Пугачев "Объектно-ориентированное программирование". – М., МГТУ, 2001 г.
8. Другие методические материалы по дисциплине с сайта www.sergebolshakov.ru.
9. Конспекты лекций по дисциплине “Программирование на основе классов и шаблонов”.
10. Эккель Б. Философия C++. Введение в стандартный C++. 2-е изд.- СПб.: Питер, 2004.- 572с.: ил.
11. Эккель Б. Эллисон Ч. Философия C++. Практическое программирование. - СПб.: Питер, 2004.- 608с.: ил.
12. Страуструп, Бьярн. Программирование: принципы и практика с использованием C++, 2-е изд. : Пер. с англ. - М . : ООО "И . Д. Вильяме", 2016. - 1328с. : ил . - Парал. тит. англ. ISBN 978-5-8459- 1 949-6 (рус .)
13. Страуструп Б. "Дизайн и эволюция C++. Классика CS" – СПб,: Питер , 2007. – 445с.
14.

9. Справочные материалы

Конструкторы класса list:

list	Constructs a list of a specific size or with elements of a specific value or with a specific **allocator** or as a copy of some other list.

Определения typedefs list:

allocator_type	A type that represents the **allocator** class for a list object.
const_iterator	A type that provides a bidirectional iterator that can read a **const** element in a list.
const_pointer	A type that provides a pointer to a **const** element in a list.
const_reference	A type that provides a reference to a **const** element stored in a list for reading and performing **const** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **const** element in a list.
difference_type	A type that provides the difference between two iterators that refer to elements within the same list.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a list.
pointer	A type that provides a pointer to an element in a list.
reference	A type that provides a reference to a **const** element stored in a list for reading and performing **const** operations.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed list.

size_type	A type that counts the number of elements in a list.
value_type	A type that represents the data type stored in a list.

Методы класса list:

assign	Erases elements from a list and copies a new set of elements to the target list.
back	Returns a reference to the last element of a list.
begin	Returns an iterator addressing the first element in a list.
clear	Erases all the elements of a list.
empty	Tests if a list is empty.
end	Returns an iterator that addresses the location succeeding the last element in a list.
erase	Removes an element or a range of elements in a list from specified positions.
front	Returns a reference to the first element in a list.
get_allocator	Returns a copy of the **allocator** object used to construct a list.
insert	Inserts an element or a number of elements or a range of elements into a list at a specified position.
max_size	Returns the maximum length of a list.
merge	Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.
pop_back	Deletes the element at the end of a list.
pop_front	Deletes the element at the beginning of a list.
push_back	Adds an element to the end of a list.
push_front	Adds an element to the beginning of a list.
rbegin	Returns an iterator addressing the first element in a reversed list.
remove	Erases elements in a list that match a specified value.
remove_if	Erases elements from the list for which a specified predicate is satisfied.
rend	Returns an iterator that addresses the location succeeding the

	last element in a reversed list.
resize	Specifies a new size for a list.
reverse	Reverses the order in which the elements occur in a list.
size	Returns the number of elements in a list.
sort	Arranges the elements of a list in ascending order or with respect to some other order relation.
splice	Removes elements from the argument list and inserts them into the target list.
swap	Exchanges the elements of two lists.
unique	Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from the list.