МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский технологический университет»
МИРЭА

Институт Информационных технологий
наименование института (полностью)

Кафедра Вычислительной техники
наименование кафедры (полностью)

КОНСПЕКТ ЛЕКЦИЙ ПО ДИСЦИПЛИНЕ

Объектно-ориентированное программирование

(наименование дисциплины)

Направление подготовки
(код и наименование)

Профиль
(код и наименование)

Форма обучения **очное**
(очная, очно-заочная, заочная)

Программа подготовки
(академический, прикладной бакалавриат)

Квалификация выпускника **Бакалавр**

Москва 2019

Содержание

1. Объектная организация окружающего мира 5
1.1. Этапы разработки программы 11
2. Язык C++ 14
2.1. Ключевые слова и имена 14
2.2. Структура предложения C++ 15
3. Класс 16
3.1. Виртуальный объект 16
3.2. Описание класса 16
3.2.1. Управление доступом к элементам класса 17
3.2.2. Инкапсуляция 18
3.2.3. Конструктор объекта 18
3.2.4. Деструктор объекта 19
3.2.5. Объявление объекта и доступ к его элементам 19
3.2.6. Указатели и ссылки на объекты 20
3.2.7. Указатель this 22
3.2.8. Присвоение объектов 23
3.2.9. Параметризованные конструкторы 24
3.2.10. Передача объектов функциям 26
3.2.11. Объявление элементов класса спецификацией const 28
3.2.12. Конструктор копии 28
3.2.13. Объекты в качестве возвращаемого значения функции 29
3.2.14. Встраиваемые функции 30
3.2.15. Дружественная функция 31
3.2.16. Дружественный класс 32
3.2.17. Операторы new и delete 33
3.3. Объявление элементов класса спецификацией static 35
4. Наследование 37

4.1. Защищенные члены класса 38
4.2. Множественное наследование 39
4.3. Виртуальные базовые классы 41
4.4. Указатель на объект производного класса 43
5. Полиморфизм 46
5.1. Пояснения к полиморфизму 46
5.2. Виртуальный метод 46
5.3. Наследование виртуального метода 48
5.4. Чисто виртуальные методы и абстрактные классы 48
5.5. Перегрузка функций 50
5.5.1. Аргументы, передаваемые функции по умолчанию 50
5.5.2. Определение адреса перегруженной функции 51
5.6. Перегрузка методов 53
5.6.1. Определение адреса перегруженного метода 53
5.7. Перегрузка операторов 53
6. Шаблоноы функций и классов 54
6.1. Шаблонны функций 54
6.2. Перегрузка шаблонных функций 56
6.3. Шаблоны классов 58
6.4. Использование параметров, не являющихся типами 61
6.5. Объявление элементов класса спецификацией static 66
7. Структуры и объединения 68
8. Обработка исключительных ситуаций 71
9. Стандартная библиотека STL 75
9.1. Знакомство с библиотекой стандартных шаблонов 75
9.2. Классы-контейнеры 76
9.3. Алгоритмы 78
10. Динамическая идентификация и приведение типов 83

10.1. Понятие о динамической идентификации типа 83
10.2. Оператор dynamic jcast 84
10.3. Операторы const_cast, reinterpret_cast и static_cast 85
11. Распределение частей описания класса по файлам 88
11.1. Конструктивное построение программы 88
12. Список литературы 90

1. ОБЪЕКТНАЯ ОРГАНИЗАЦИЯ ОКРУЖАЮЩЕГО МИРА

Окружающий нас мир состоит из объектов. Объекты созданы человеком или природой. Человек научился из объектов конструировать системы для решения определенных задач. Система, в свою очередь тоже является объектом. Так, что все происходящее вокруг – это, взаимодействие объектов.

Объект – то, что может быть индивидуально описано и рассмотрено.

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

По мере расширения предметных областей применения вычислительной техники появляются новые парадигмы программирования, соответствующие им алгоритмические языки и инструменты разработки. Их появление обусловлено необходимостью наиболее адекватного, взаимно однозначного отображения новой предметной области в языки программирования и в функциональные возможности инструментов разработки.

Объектная, системная организация окружающего мира и широкое проникновение вычислительной техники во все сферы человеческой деятельности, естественным образом породила парадигму объектно-ориентированного программирования.

Отметим некоторые общие особенности объектов и систем.

Рукотворные, создание человеком объекты имеют определенный жизненный цикл, который можно выразить следующей схемой

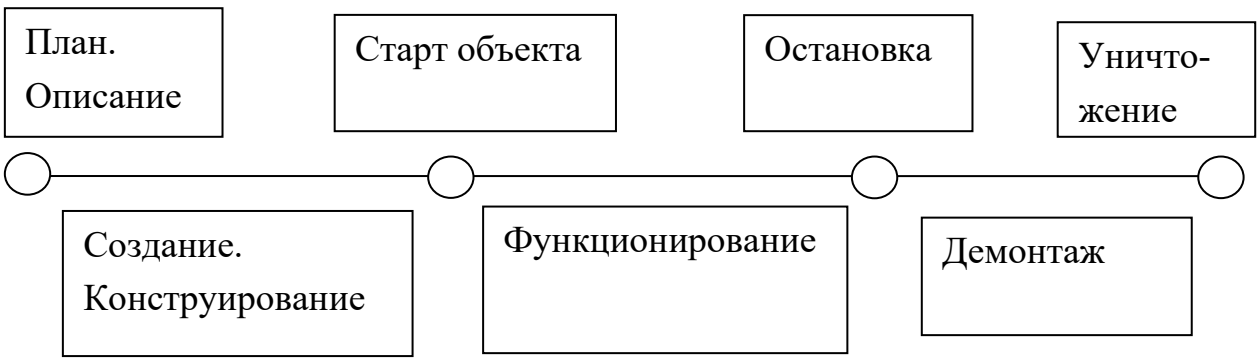


Рисунок 1

Примеры: дом, ПК, автомобиль и т.д.

Совершенствование объекта отображается в реализации новых, более совершенных, качественных, функционально развитых экземпляров.

Разработка любой системы, включает, в том числе выполнение следующих работ:

1. Определение цели, множества задач, для решения которых предназначена система.
2. Описание, создание необходимых объектов, составных частей (элементов) системы.
3. Конструктивная сборка системы.
4. Определения правил взаимодействия составных объектов системы. Построение соответствующих интерфейсов.
5. Запуск системы для функционирования, решения задач (согласно назначению).

Конструктивно построенная из составных объектов (элементов) система имеет иерархическую структуру.

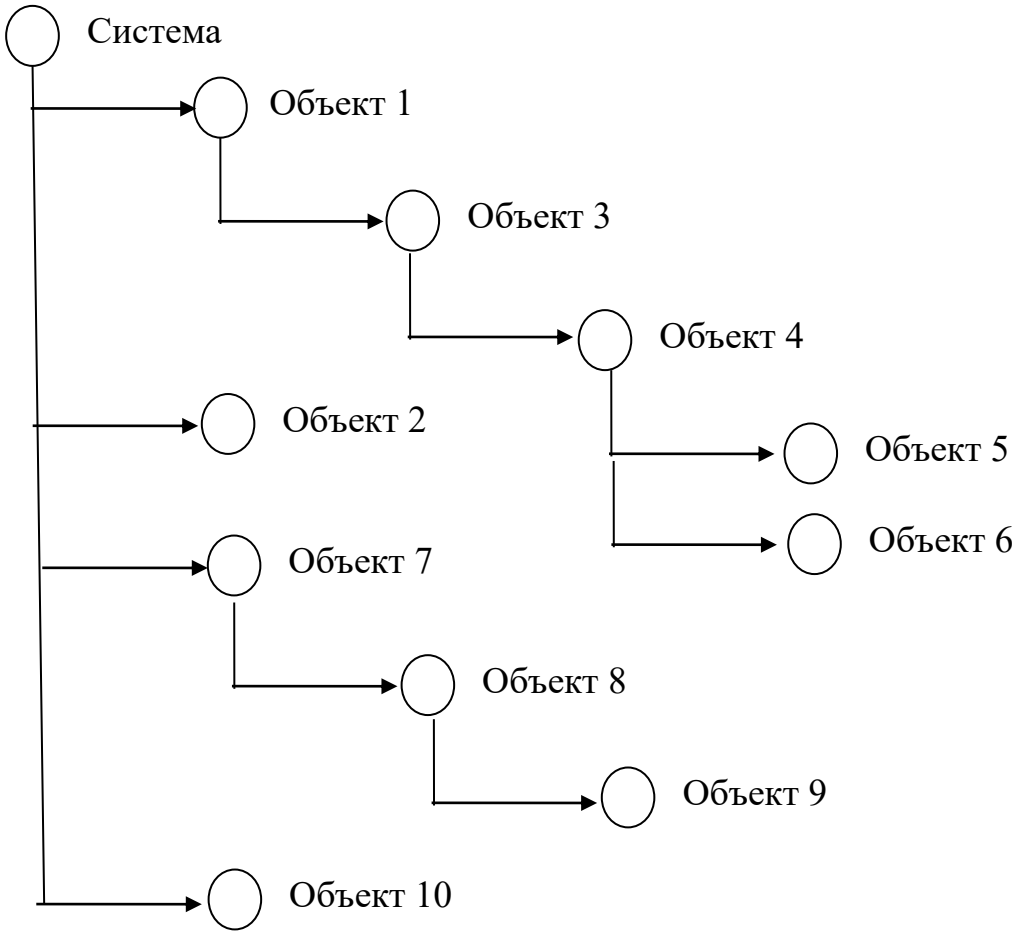


Рисунок 2

Разбиение системы на объекты является объектной декомпозицией. Такая декомпозиция существенно упрощает конструирование сложных систем.

Разработка интерфейсов взаимодействия происходит между объектами системы и внешней средой. Способ организации интерфейса зависит от особенности объекта и его назначения. Схематически это представлено на следующем рисунке.

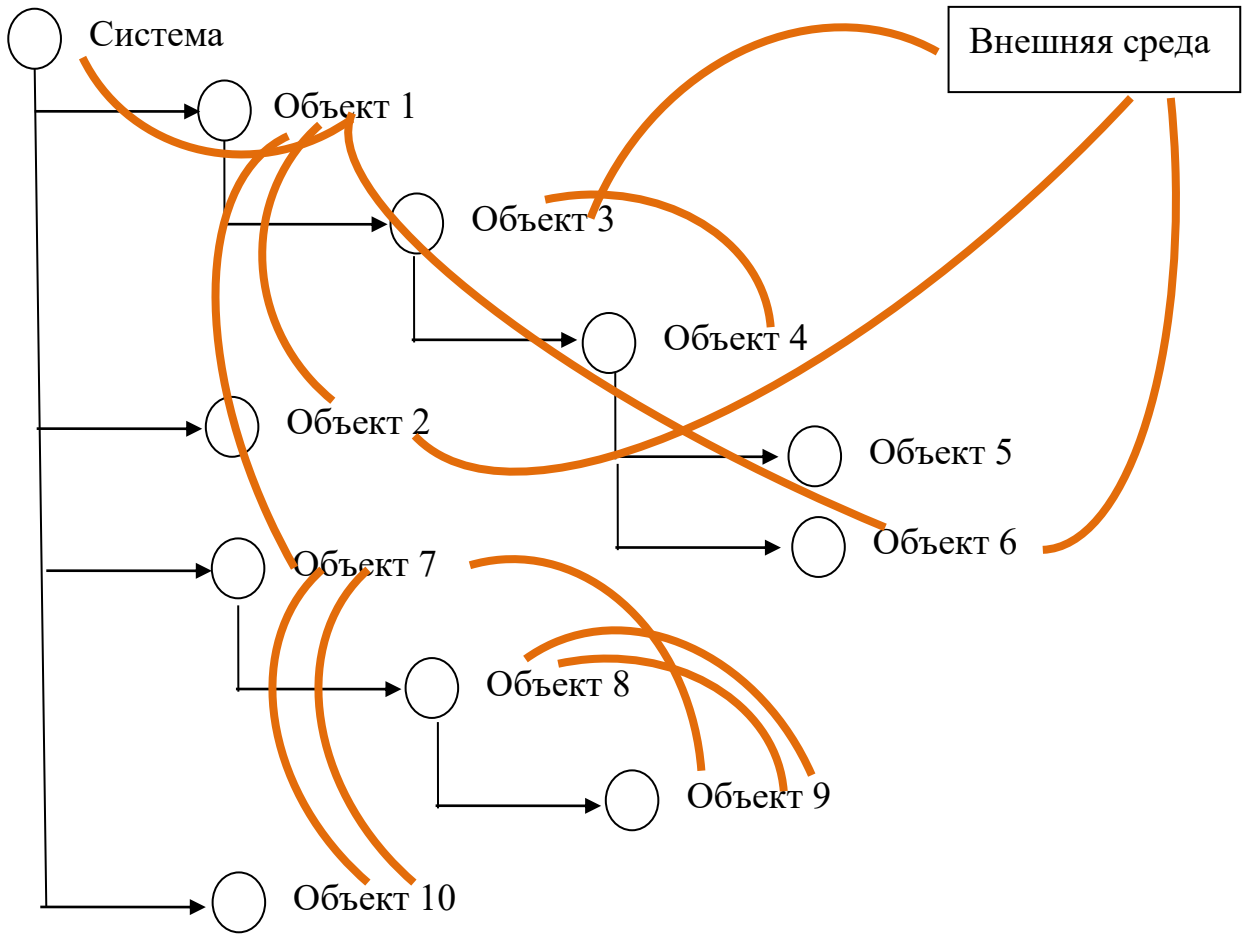


Рисунок 3

Объект при взаимодействии с другим объектом, отвечает обратной реакцией, выполняет определенные действия, согласно своего функционального назначения. Схематически это выглядит следующим образом.

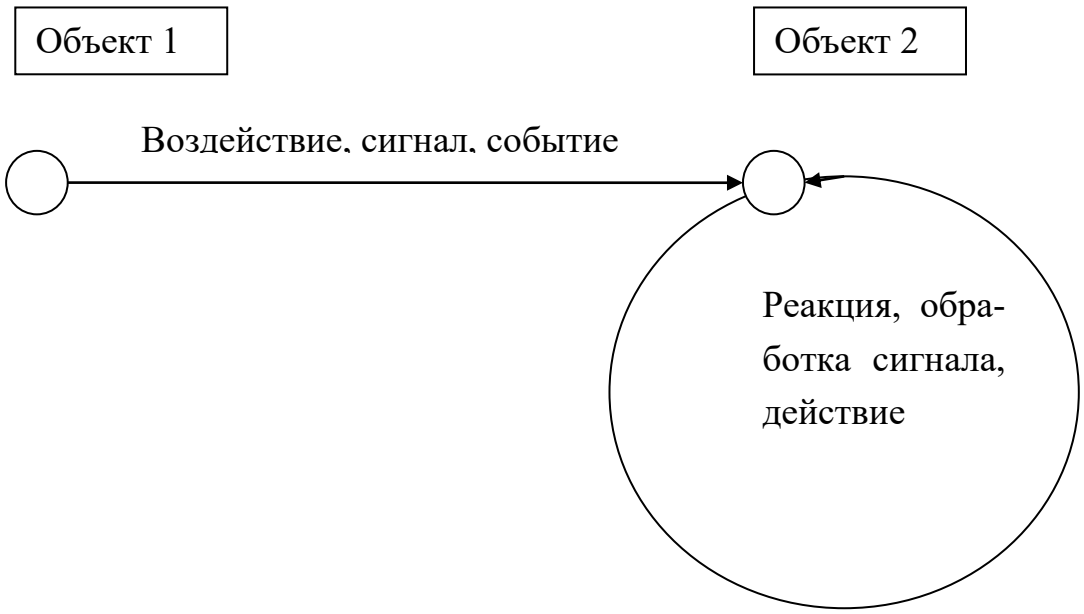


Рисунок 4

В рамках системы устанавливается множество взаимодействия пар объектов. Их количество и взаимонаправленность определяется функциональным назначением системы и функциональными возможностями объектов.

Алгоритм решения задачи представляет собой последовательность выполнения воздействий (сигналов, событий) от определенных объектов и соответствующих реакций от взаимодействующих объектов.

Пример, позвонить по телефону.

Получается, для решения задачи, часть объектов должны обладать свойством запустить воздействие (принять решения, выдать сигнал, генерировать определенной событие). Объекты с таким функционалом еще называют субъектами.

В рамках парадигмы объектно-ориентированного программирования была сформулирована требование, наиболее адекватно отобразить объектную, системную организацию предметной области в виртуальную. Для достижения этой цели были разработаны объектно-ориентированные языки программирования. Эти алгоритмические языки наиболее взаимно однозначно позволяют отобразить системное, объектное строение предметной области и упрощают построение алгоритмов решения задач. Таким языкам программирования предъявляются определенные требования. Язык должен позволять:

1. Описать объект.
2. Конструировать наследственность свойств и функционала объектов.

3. Конструировать иерархию объекта.
4. Конструировать связи взаимодействующих объектов.
5. Определить интерфейсы взаимодействия объектов (механизмы, кодирование, правила).

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

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

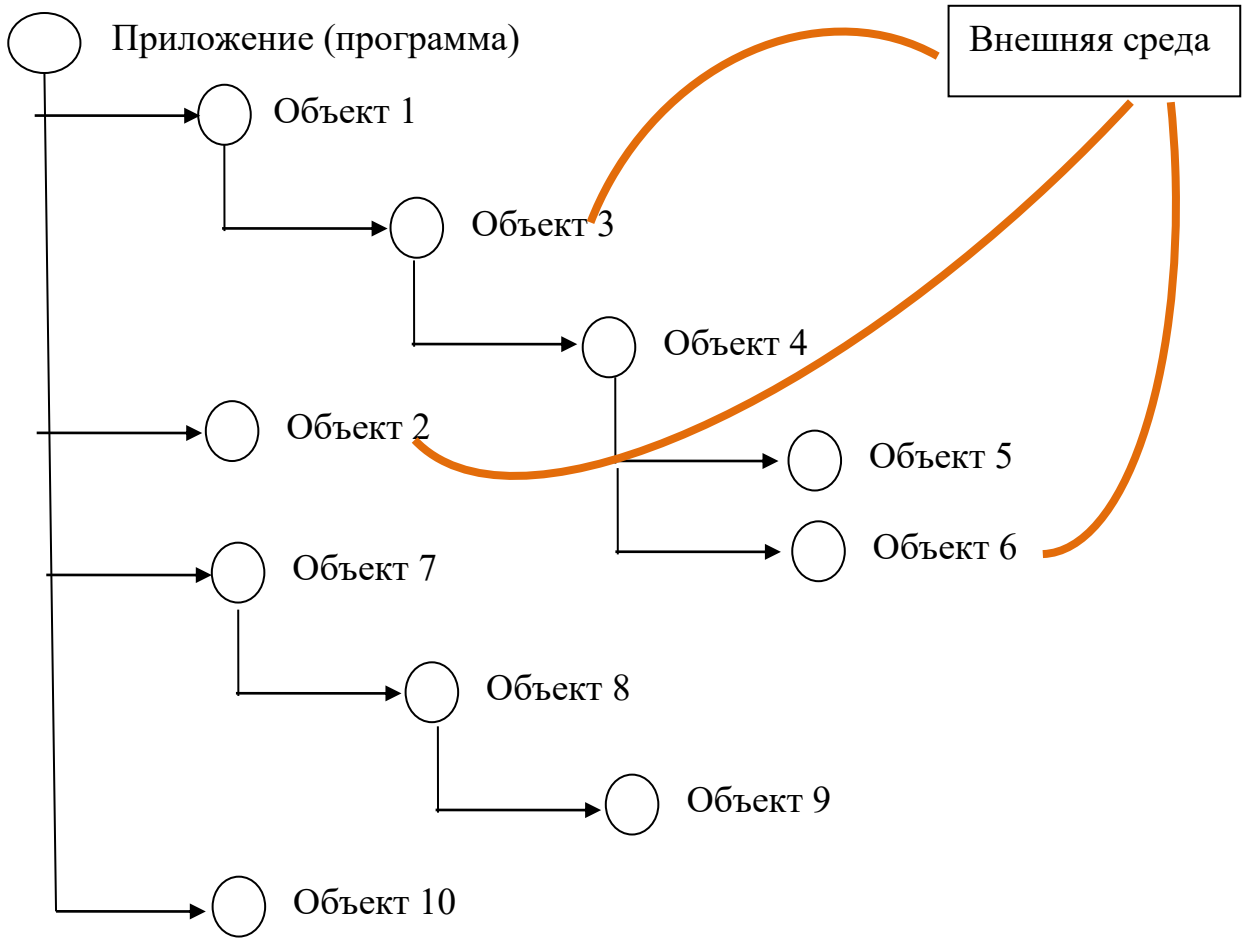


Рисунок 5

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

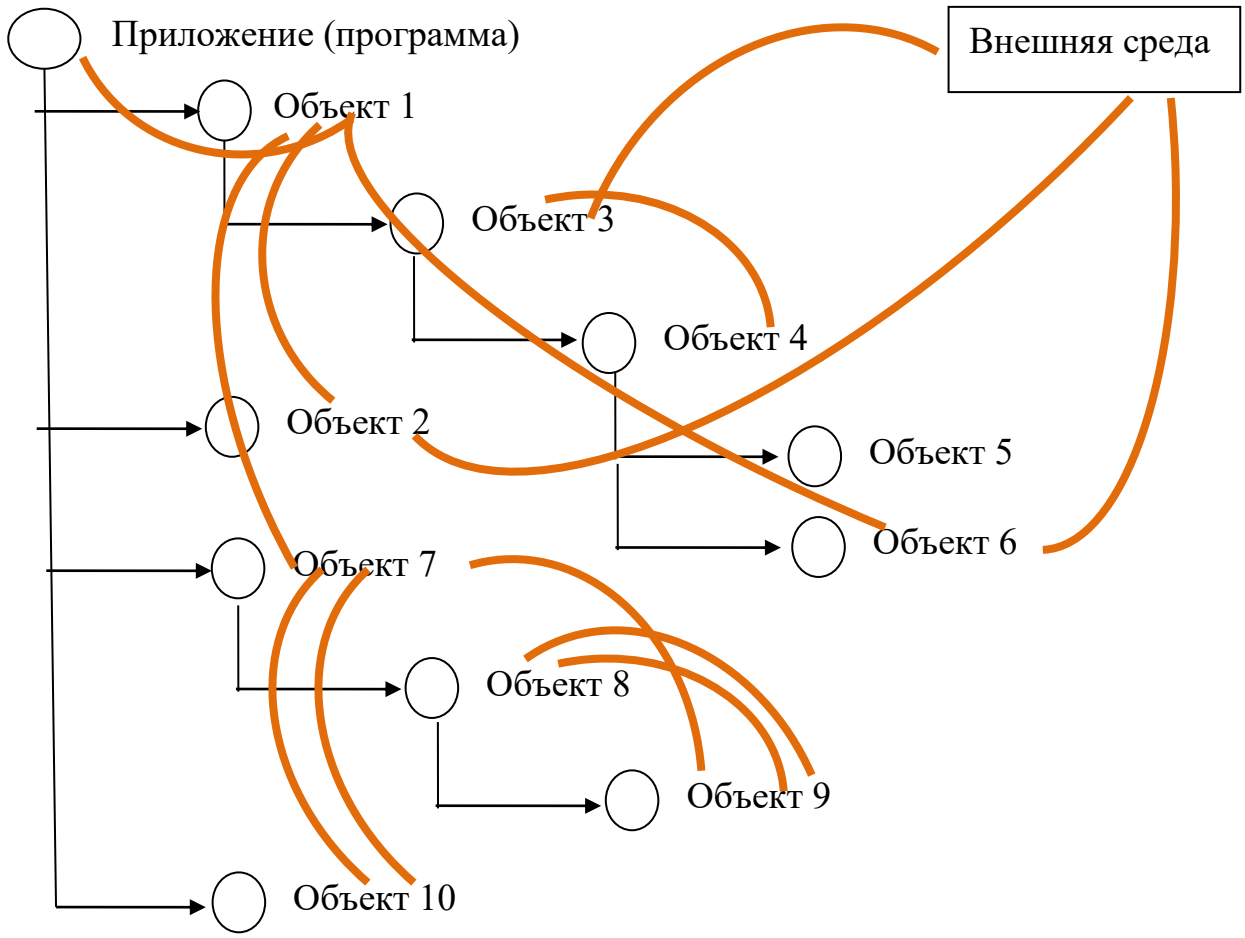


Рисунок 6

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

1.1. Этапы разработки программы

Содержание этапов разработки программы (системы) при объектно-ориентированном программировании отличается от сформулированных при процедурном программировании (Таблица 1)

Процедурное программирование	**Объектно-ориентированное программирование**

Постановка задачи (что)	Постановка задачи (что)
Метод решения (чем)	Набор объектов. Используемые методы решения.
Алгоритм решения задачи (как)	Архитектура программы-системы. Взаимодействие объектов. Алгоритм функционирования, решения задачи (как).
Блок-схема алгоритма	Схема архитектуры. Схема взаимодействия объектов. Схема алгоритма решения задачи.
Код программы	Код описания объектов. Код конструирования системы (архитектуры программы). Код взаимодействия объектов. Код алгоритма решения задачи.
Тестирование и отладка	Тестирование и отладка
Доработка документации	Доработка документации
Сдача программы и сопроводительной документации	Сдача программы и сопроводительной документации

Последовательность тем лекционного курса соответствует последовательности построения системы. Будут обсуждены следующие основные темы:

1. Описание объекта.
2. Описание, разработка архитектуры объекта.
3. Описание, разработка архитектуры системы, взаимосвязи объектов.
4. Описание, разработка взаимодействия объектов (создание интерфейсов).
5. Реализация алгоритма (алгоритмов) решения поставленной задачи (поставленных задач).

В рамках каждой темы будут изложены фрагменты языка C++ для реализации соответствующих конструкций и алгоритмов.

Примеры кода в основном взяты из книг Герберта Шилдта «C++ базовый курс» и «Самоучитель C++». Дополнительные пояснения к коду из этой книги, несомненно, способствуют освоению материала.

Для более глубокого освоения лекционного материала параллельно будут проводиться практические и лабораторные занятия. Каждому студенту будет предложен курсовой проект. В рамках этих занятий будут разобраны и представлены для самостоятельного выполнения множество заданий. Основная составная часть каждого задания это - задача. По назначению и сложности они классифицируются следующим образом:

Практические занятия

Будут рассмотрены задачи пояснительного характера, отвечающие на вопрос, как определенная конструкция реализована на языке C++, как она работает, какие особенности, требования, возможности.

Выполняется одно подготовительное задание для последующего выполнения лабораторных и курсовой работы. Дата выдачи данного задания является датой выдачи курсовой, начала его выполнения.

Курсовая работа

Для самостоятельного выполнения будут предложены задания содержащие задачи второго уровня сложности. Курсовое задание функционально наполняет последую лабораторную работу.

Разработку программного кода необходимо выполнить с использованием автоматизированной системы обучения «Аврора».

2. ЯЗЫК С++

Язык С++, к изучению которого посвящен данный лекционный курс относится к множеству объектно-ориентированных алгоритмических языков. Язык является наследником, развитием языка С. По этому, синтаксис языка удовлетворяет всем правилам оговоренным в языке С и сдержит все его языковые конструкции. Подразумеваем, что слушатели знакомы с алгоритмическим языком С.

Детально будут изложены синтаксические и семантические особенности алгоритмического языка С++. Рассмотрены реализованные в языке механизмы. Лекционному курсу соответствуют примеры, решаемые на практических и лабораторных занятиях. На практических и лабораторных занятиях будут предлагаться примеры следующего содержания:

- элементы языка, как программируются и как работают;
- задачи на конструирование объектов и работа с объектами;
- задачи на конструирование приложения;
- взаимодействие между объектами с использованием открытых методов;
- взаимодействие между объектами по сигналам и обработчикам;
- организация ввода и вывода данных с использованием потоков.

Описание некоторых элементов языка будет изложено фрагментарно, исходя из пройденного объема материала лекционного курса и обсуждаемой темы. Другими словами, некоторые элементы языка будут обсуждаться многократно, разной степени детализацией.

2.1. Ключевые слова и имена

Часть идентификаторов С++ входит в фиксированный словарь ключевых слов. Прочие идентификаторы после специального объявления становятся именами. Имена служат для обозначения классов, объектов, переменных, типов данных, методов, функций и меток.

Список ключевых слов:

asm auto bool break case

catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	operator	private	protected
public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while			

2.2. Структура предложения C++

Предложения в C++ называются операторами. Оператор завершается символом «;». Оператор C++ состоит из выражений и может содержать вложенные операторы. Выражение является частью оператора и строится на основе множества символов операций, ключевых слов и операндов. Операндами являются литералы и имена. Одной из характеристик выражения является его значение, которое вычисляется на основе значений операндов по правилам, задаваемым операциями.

3. КЛАСС

3.1. Виртуальный объект

В языке C++ реальному объекту из предметной области ставиться в соответствии виртуальный объект. Язык обеспечивает жизненный цикл виртуального объекта по аналогии жизненного цикла объекта из предметной области. Схемы этих жизненных циклов совпадают.

Схема жизненного цикла виртуального объекта.

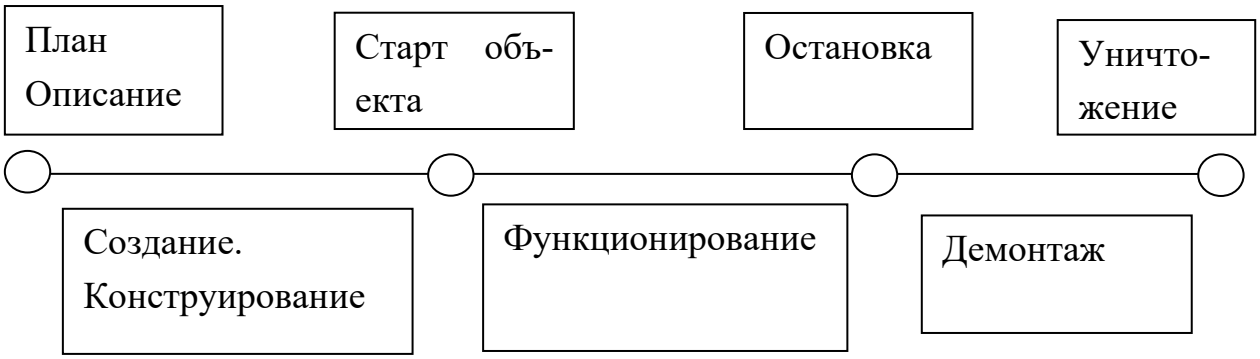


Рисунок 7

Реализация жизненного цикла виртуального объекта на языке C++ (Таблица 2)

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

3.2. Описание класса

Описание класса состоит из двух частей: заголовочного и реализации.

Синтаксис описания заголовочной части класса:

```
class  «имя класса» {
[private:]
        «список скрытых элементов класса»
public:
        «список доступных элементов класса»
protected:
        «список защищенных элементов класса»
};
```

имя класса ::= идентификатор
элемент класса ::= описание свойства (поля, переменной)
::= описание заголовка метода

Синтаксис описания заголовка метода:
«тип возвращаемого значения» «имя метода» ([список параметров]);

Описание переменных аналогично описанию в языке С.

Часть реализации содержит описание методов класса. Синтаксис описания метода:

«тип возвращаемого значения» «имя класса» :: «имя метода» ([список параметров])

```
{
   // тело метода (код алгоритма метода)
}
```

Здесь «имя класса» — это имя того класса, которому принадлежит определяемый метод.

3.2.1. Управление доступом к элементам класса

Ключевые слова private, public и protected обозначают разделы описания класса. Каждый раздел определяет права доступа к элементу класса.

private – скрытые элементы класса. К свойствам и методам данного раздела имеют доступ другие элементы класса. Любое объявление, появляющееся до ключевого слова управления доступом, считается «скрытым» по умолчанию.

public –доступные элементы класса. К свойствам и методам данного раздела доступ разрешен другим элементам класса и из вне непосредственным доступом.

protected – защищенные элементы класса.

3.2.2.Инкапсуляция

Все программы состоят из двух основных элементов: инструкций (кода) и данных. Код – это часть программы, которая выполняет действия, а данные представляют собой информацию, на которую направлены эти действия.

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

На базе одного класса могут быть созданы множество объектов. Объект созданный на согласно описанию класса поддерживает инкапсуляцию.

У каждого объекта есть свои границы («рамки»), Эта граница определяется согласно описанию пространства класса.

Внутри объекта код, данные или обе эти составляющие могут быть закрытыми в «рамках» этого объекта или открытыми.

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

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

3.2.3. Конструктор объекта

Для создания объекта согласно описанию класса в списке доступных методов определяется специальный – конструктор. Имя конструктора совпадает с

именем класса. В описании заголовка конструктора отсутствует «тип возвращаемого значения». Конструктор возвращает адрес возданного объекта.

Конструктор без параметров является конструктором по умолчанию.

Синтаксис описания:

«имя класса» ([список параметров]);

В момент отработки конструктора, первоначально, для создаваемого объекта согласно описанию класса, выделяется необходимая оперативная память. После этого отрабатывает алгоритм согласно коду конструктора представленной в части реализации.

В описании класса метод конструктора может отсутствовать. В таком случае, в момент создания объекта отрабатывает конструктор по умолчанию, которого генерирует транслятор и происходит только выделение памяти для объекта.

3.2.4. Деструктор объекта

Деструктор – это метод класса, который вызывается при демонтаже (удалении) объекта. Имя деструктора совпадает с именем конструктора, т.е. класса, но предваряется символом «~». Деструктор не возвращает значение, следовательно, в его объявлении отсутствует тип возвращаемого значения. У деструктора нет параметров. В описании класса может присутствовать только один деструктор. Синтаксис описания:

~«имя класса» ( );

3.2.5. Объявление объекта и доступ к его элементам

Согласно описанию класса, можно создать произвольное количество объектов данного класса. Объект создается посредством объявления. Синтаксическая запись объявления объекта совпадает с объявлением простой переменной.

«имя класса» «имя объекта» [,«имя объекта 1» …];

имя объекта ::= идентификатор

При этом отрабатывает конструктор объекта по умолчанию. Доступ к открытым элементам объекта осуществляется по операции «точка».

«имя объекта».«имя элемента объекта» [([список аргументов])]

Пример

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

// ----- Заголовочная часть.
class myclass {
     int a;
public:
          myclass ( ); // конструктор
     void show    ( );
};

// ----- Часть реализации.
myclass :: rmyclass ( )
{
     cout << "В конструкторе \n";
     a = 10;
}
void  myclass :: show ( )
{
     cout << a ;
}

// ----- Основная программа
int main ( )
{
     myclass ob;       // объявление объекта, отработка конструктора
     ob.show ( );      // вызов открытого метода.
     return  0;
}
```

При завершении работы приложения уничтожаются созданные объекты. Для каждого отрабатывает деструктор.

3.2.6.Указатели и ссылки на объекты

Указатель на объект объявляется также, как на переменную простого типа. Значение указателя на объект должно быть инициировано. Для получения адреса объекта используется оператор «&». Для доступа к элементам объекта используется оператор стрелка «–>».

Пример

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

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

// ----- Часть реализации.
cl_1 :: cl_1 ()
{
     cout << "В конструкторе \n";
     i = 10;
}
void  cl_1 :: show_i ( )
{
     cout << “i = “ << i << “\n”;
}

// ----- Основная программа
int main ( )
{
     cl_1   ob;           // объявление объекта
     cl_1 * p_ob;         // объявление указателя объекта

     p_ob = & ob;         // инициализация указателя объекта

     p_ob –> show_i ( );  // вывод значения свойства объекта
     return  0;
}
```

```
Программа выведет на консоль следующее:
В конструкторе
i = 10
```

В C++ ссылка – это простой ссылочный тип.

При объявлении переменной, для хранения значений выделяется память. Именем переменной является идентификатор. Для доступа к значениям переменной, используется имя переменной. Объявление ссылки означает задание альтернативного идентификатора. По сути, ссылка это - указатель, который привязан к определенной области памяти.

Пример

```
int  a;      // переменная типа int, по адресу  0xdd000075
int &ra = a; // альтернативное имя переменной по адресу 0xdd000075
cout << & a << “\n” << & ra << “\n”;
```

```
Программа выведет на консоль следующее:
```

```
0xdd000075
0xdd000075
```

Ссылку нельзя объявить без привязки к переменной. Ссылка инициализируется в момент объявления. После объявления ссылки её невозможно привязать к другой переменной.

Важно отличать ссылки от оператора взятия адреса & (address of). Оператор взятия адреса используется для уже созданного объекта с целью получить его адрес, а ссылка это только задание альтернативного имени объекта.

Отличие указателя от ссылки в том, что получить само значение переменной, на которую указывает указатель, можно только выполнив операцию разыменования * (символ «*» в объявлении является объявлением указателя, а при применении к уже созданной переменной является оператором разыменования). Значение указателя можно менять.

Существует два оператора: * и &. Первый по данному адресу, который хранится в переменной типа int *, возвращает собственно данные, расположенные по этому адресу. Второй по данной переменной узнаёт её адрес в памяти.

Пример

```
int   a  =   3;   // переменная типа int
int   b  =   5;   // переменная типа int
int & ra =   a;   // альтернативное имя переменной a
int * p  = & a;

p = & b;
cout << * p << "\n" << ra << "\n";

Программа выведет на консоль следующее:
5
3
```

3.2.7.Указатель this

Каждый объект содержит свой экземпляр свойств (полей, переменных) согласно описанию класса. Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами данного класса совместно. Поэтому возникает необходимость обеспечить работу методов с свойствами именно того объекта, для которого они были вызваны. Это обеспечивается передачей в метод скрытого параметра this, в котором хранится указатель на вызвавший ме-

тод объект. Указатель this неявно используется внутри метода для ссылок на элементы объекта. В явном виде этот указатель применяется в основном для возвращения из метода указателя (return this;) или ссылки (return * this;) на вызвавший объект.

Пример

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

// ----- Заголовочная часть.
class cl_1 {
     int i;
public:
     void load_i ( int val );
     void get_i  ( );
};
// ----- Часть реализации.
cl_1 :: load_i ( int val )
{
     this -> i = val;    // тоже самое, что и   i = val;
}
cl_1 :: get_i ( )
{
     return this -> i;   // тоже самое, что и   return i;
}

// ----- Основная программа
int main ( )
{
     cl_1    ob;              // отработка конструктора по умолчанию

     ob.load_i ( 50 );                // инициализация свойства i.
     cout << "i = " << ob.get_i ( );  // вывод значения i.

     return  0;
}

Программа выведет на консоль следующее:
i = 50
```

3.2.8.Присвоение объектов

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

«имя объекта 1» = «имя объекта 2»;

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

Пример

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

// ----- Заголовочная часть.
class cl_1 {
private:
     int i;
public:
     void set_i  ( int k );
     void show_i ( );
};
// ----- Часть реализации.
void  cl_1 :: set_i ( int k )
{
     i = k;
}
void  cl_1 :: show_i ( )
{
     cout << "i = " << i << "\n";
}

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

Программа выведет на консоль следующее:
i = 11
i = 15
```

3.2.9.Параметризованные конструкторы

В описании конструктора можно задать параметры. При вызове конструкта объекта ему передаются соответствующие аргументы. С их помощью при создании объекта переменным объекта можно присвоить некоторые начальные значения, выполнить определенные методы объекта.

Синтаксис вызова параметризированного конструктора при объявлении объекта имеет два способа реализации:

«имя класса» «имя объекта» (список аргументов);

«имя класса» «имя объекта» = «имя класса» (список аргументов);

Если у конструктора только один параметр, то можно использовать альтернативный способ вызова конструктора:

«имя класса» «имя объекта» = аргумент;

Пример

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

// ----- Заголовочная часть.
class cl_3 {
     int i;
public:
          cl_3   ( );       // конструктор по умолчанию
          cl_3   ( int j ); // конструктор с одним параметром
     void show_i ( );
};
// ----- Часть реализации.
cl_3 :: cl_3 ( )
{
     cout << "Конструктор по умолчанию \n";
     i = 10;
}
cl_3 :: cl_3 ( int j )
{
     cout << "Конструктор с параметром \n"; ;
     i = j;
}
void  cl_3 :: show_i ( )
{
     cout << “i = ” << i << “\n”;
}

// ----- Основная программа
int main ( )
{
     cl_3 ob;          // отработка конструктора по умолчанию
     cl_3 ob1 ( 5 );   // отработка конструктора с параметром
     cl_3 ob2 = 7;     // отработка конструктора с параметром
     ob.show_i  ( );   // вызов открытого метода.
     Ob1.show_i ( );   // вызов открытого метода.
     Ob2.show_i ( );   // вызов открытого метода.
     return  0;
}
```

```
Программа выведет на консоль следующее:
Конструктор по умолчанию
Конструктор с параметром
Конструктор с параметром
```

```
i = 10
i = 5
i = 7
```

3.2.10. Передача объектов функциям

Объекты можно передавать функциям в качестве аргументов точно так же, как передаются переменные других типов. Параметр функции объявляется как имеющий тип класса. При вызове функции объект этого класса используется в качестве аргумента. Как и для переменных других типов, по умолчанию объекты передаются в функции по значению. Внутри функции создается копия аргумента и эта копия, а не сам объект, используется функцией. Поэтому изменение копии объекта внутри функции не влияет на сам объект. Это иллюстрируется следующим примером:

Пример

```
#include <iostream>
using namespace std;
class samp {
    int i;
public:
         samp  ( int n ) { i = n; }
    void set_i ( int n ) { i = n; }
    int  get_i ( )       { return i; }
/* Заменяет переменную o.i ее квадратом. Однако это не влияет на
объект, используемый для вызова функции sqr_it () */
void sqr_it ( samp o ) {
    o.set_i ( o.get_i ( ) * o.get_i ( ) );
    cout << "Для копии объекта a значение i равно: " << o.get_i();
    cout << "\n";
}

int main ( )
{
    samp a ( 10 );
    sqr_it (  a ); // передача объекта a по значению
    cout << " переменная a.i в функции main ( ) не изменилась: ";
    cout << a.get_i(); // выводится 10
    return 0;
}
Программа выведет на консоль следующее:
Для копии объекта a значение i равно: 100 но переменная a.i в
функции main() не изменилась: 10
```

Если при передаче объекта в функцию делается его копия, это означает, что появляется новый объект. Когда работа функции, которой был передан объект, завершается, то копия аргумента удаляется.

Когда при вызове функции создается копия объекта, конструктор копии не вызывается. Поскольку конструктор обычно используется для инициализации некоторых составляющих объекта, он не должен вызываться при создании копии уже существующего объекта. Если бы это было сделано, то изменилось бы содержимое объекта, поскольку при передаче объекта функции необходимо его текущее, а не начальное состояние.

Однако если работа функции завершается и копия удаляется, то деструктор копии вызывается. Это происходит потому, что иначе оказались бы невыполненными некоторые необходимые операции. Например, для копии может быть выделена память, которую, после завершения работы функции, необходимо освободить.

Пример

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

class samp {
     int i;
public:
     samp  ( int n );
    ~samp  ( );
 int get_i ( );
};

samp :: samp  ( int n )
{
     i = n;
     cout << "Работа конструктора \n";
}
samp :: ~samp ( )
{
     cout << "Работа деструктора \n";
}
int samp :: get_i ( )
{
     return i;
}

// Возвращает квадрат переменной o.i
int sqr_it ( samp o ) {
     return o.get_i ( ) * o.get_i ( );
}
```

```
int main ( )
{
     samp a ( 10 );
     cout << sqr_it( a ) << "\n";
     return 0;
}
Программа выведет на консоль следующее:
Работа конструктора
100
Работа деструктора
Работа деструктор
```

3.2.11. Объявление элементов класса спецификацией const

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

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

3.2.12. Конструктор копии

Для исключения ошибки при передаче объекта в метод или в функцию по значению, определен специальный перегружаемый конструктор – конструктор копии. Синтаксис определения заголовка конструктора копии:

«имя класса» (const «имя класса» & «имя параметра»);

В части реализации определяется алгоритм для инициализации свойств создаваемого временного объекта. Для инициализации доступны все элементы временного объекта согласно описанию класса. В конструкторе «имя параметра» соответствует ссылке на объект, передаваемой в функции или методу. Надо отметить, по ссылке «имя параметра» по операции «точка» доступны все элементы передаваемого объекта (из закрытого раздела тоже), согласно описанию класса.

При передаче в функцию или в метод объекта данного класса по значению, при создании временного объекта поразрядное копирование не выполняется, отрабатывает конструктор копии. В алгоритме реализации надо предусмотреть инициирование всех необходимых свойств временного объекта. При выходе из функции или метода отработка деструктора будет выполнена относительно временного объекта.

Пример

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

class cl_1 {
     int i;
public:
     cl_1  ( const cl_1 & ob );
};

cl_1 :: cl_1  (const cl_1 & ob )
{
     i = ob.i;
}

void func ( cl_1 ob_local ) {
      . . . .
}

int main ( )
{
     cl_1  ob;        // Конструктор по умолчанию
     func ( ob );     // Конструктор копии
     return 0;
}
```

В конструкторе копии локальному объекту принадлежит свойство «i» в левой части оператора присваивания. По оператору «точка» получаем значение переданного в качество аргумента объекта. При этом, так как, оператор принадлежит методу класса, доступ к закрытому элементу переданного объекта этого же класса, получаем непосредственно.

3.2.13. Объекты в качестве возвращаемого значения функции

Так же как объект может быть передан функции в качестве аргумента, он может быть и возвращаемым значением функции. Для этого, во-первых, в объ-

явлении функции указать тип класса в качестве типа возвращаемого значения. Во-вторых, объект типа класса возвращается с помощью обычной инструкции return.

Имеется одно важное замечание по поводу объекта в качестве возвращаемого значения функции: если функция возвращает объект, то для хранения возвращаемого значения автоматически создается временный объект. После того как значение возвращено, этот объект удаляется. Во многих учебниках указано, что в момент возврата объекта происходит вызов конструктора копии и деструктора. Это не естественно, объект обрабатывается согласно алгоритму функции или метода. Перед возвратом если применить конструктор копии, он может исказить данные.

3.2.14. Встраиваемые функции

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

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

При описании класса встраиваемую функцию можно определить двумя способами.

Первый способ определение кода реализации в заголовочной части класса.

Второй способ относится части реализации. Синтаксис представления

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

Пример

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

class cl_1 {
     int i;
     int j;
public:
```

```
     int get_i ( ) { return i; }
     int get_j ( );
};

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

3.2.15. Дружественная функция

Дружественной является функция, которая не является членом класса, но получает доступ к закрытым элементам объекта класса. Для определения дружественной функции используется ключевое слово friend.

Дружественная функция определяется в разделе открытых элементов описания класса. Синтаксис представления

friend «тип функции» «имя функции» («параметры»);

В составе параметров должен быть параметр с типом данного класса.

Пример

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

class cl {
  // ...
public:
  friend void frnd ( cl ob );
 // ...
 };
```

Если дружественная функция определена раньше определения класса, то компилятор выдаст ошибку относительно идентификатора наименования класса. Для исключения такой ошибки, в языке C++ имеется оператор предварительного объявления наименования класса. Синтаксис оператора:

class «имя класса»;

Пример

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

class myclass;

int sum ( myclass x )
{
```

```
  return x.a + x.b;
}

class myclass {
  int a, b;
public:
  myclass ( int i, int j ) { a=i; b=j; }
  friend int sum ( myclass x ); // дружественная функция
};

int main()
{
  myclass n ( 3, 4 );
  cout << sum ( n );

  return 0;
}
```

3.2.16. Дружественный класс

По аналогии с дружественной функцией, в алгоритмах реализации методов дружественного класса можно получить доступ к закрытым элементам объекта исходного класса. Дружественный класс определяется в разделе открытых элементов описания класса. Синтаксис представления

friend «имя дружественного класса»;

Пример

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

class cl_f;

class cl_1 {
     int a, b;
public:
     cl_1 ( int i, int j ) { a=i; b=j; }
     friend  cl_f; // дружественный класс
};
class cl_f {
public:
     void change ( cl_1 & c ) { c.a = c.b; }
};
```

3.2.17. Операторы new и delete

При выделении динамической памяти использовали функцию malloc(), а при освобождении памяти — функцию free(). Вместо этих стандартных функций в C++ стал применяться более безопасный и удобный способ выделения и освобождения памяти. Выделить память можно с помощью оператора new, а освободить ее с помощью оператора delete. Ниже представлена основная форма этих операторов:

«указатель на объект класса» = new «имя класса» [(«аргументы»)];

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

delete «указатель на объект класса»;

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

Хотя операторы new и delete выполняют сходные с функциями malloc() и free() задачи, они имеют несколько преимуществ перед ними. Во-первых, оператор new автоматически выделяет требуемое для хранения объекта заданного типа количество памяти. Вам теперь не нужно использовать sizeof, например, для подсчета требуемого числа байтов. Это уменьшает вероятность ошибки. Во-вторых, оператор new автоматически возвращает указатель на заданный тип данных. Вам не нужно выполнять приведение типов, операцию, которую вы делали, когда выделяли память, с помощью функции malloc() (см. следующее замечание). В-третьих, как оператор new, так и оператор delete можно перегружать, что дает возможность простой реализации вашей собственной, привычной модели распределения памяти. В-четвертых, допускается инициализация объекта, для которого динамически выделена память. Наконец, больше нет

необходимости включать в ваши программы заголовок <cstdlib>.

Пример

```
#tinclude <iostream>
using namespace std;
int main(}
{
     int * p;
     p = new int; // выделение памяти для целого
     if ( ! p ) {
          cout << "Ошибка выделения памяти\п";
          return 1;
     }
     * p = 1000;
     cout << "Это целое, на которое указывает p: " << * p << "\п";
     delete p; // освобождение памяти
     return 0;
}
```

Пример

Создание объекта

```
#include <iostream>
using namespace std;
class samp {
     int i, j;
public:
     void set_ij ( int a, int b ) { i = a; j = b; }
     int  get_product() { return i * j; }
};
int main() {
     samp * p;
     p = new samp; // выделение памяти объекту
     if( ! p ) {
          cout << "Ошибка выделения памяти\п";
          return 1;
     }
     p -> set_ij ( 4, 5 );
     cout << "Итог равен:" << p->get_product(} << "\n";
     return 0;
}
```

Для динамически размещаемого одномерного массива используйте такую форму оператора new:

«указатель на массив» = new «имя типа» [«размер»];

После выполнения этого оператора указатель «указатель на массив» будет указывать на начальный элемент массива. Для удаления динамически раз-

мещенного одномерного массива следует использовать следующую форму оператора delete:

delete [] «указатель на массив»;

3.3. Объявление элементов класса спецификацией static

Статические свойства (поля, переменные).

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

Статические поля

Статические свойства применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

Свойства статических полей:

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

```
#include <iostream.h>
class Example
{
public:
     static int value; //объявление в классе
};

int Example::value; //определение статического поля в глобальной
                    //области, по умолчанию инициализируется нулем.
// int Example::value=10; // пример инициализации произвольным
                          // значением
```

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

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

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

мощью статических методов;

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

Статические методы

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

4. НАСЛЕДОВАНИЕ

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

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

Для наследования одного класса другим, используется следующая основная форма записи:
class «имя производного класса» : «спецификатор доступа» «имя базового класса»

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

Спецификатор доступа определяет, как элементы базового класса (base class) наследуются производным классом (derived class). Если спецификатором доступа наследуемого базового класса является ключевое слово public, то все открытые члены базового класса остаются открытыми и в производном. Если спецификатором доступа наследуемого базового класса является ключевое слово private, то все открытые члены базового класса в производном классе становятся закрытыми. В обоих случаях все закрытые члены базового класса в производном классе остаются закрытыми и недоступными. Важно понимать, что если спецификатором доступа является ключевое слово private, то хотя открытые члены базового класса становятся закрытыми в производном, они остаются доступными для функций — членов производного класса.

Пример

```
#include <iostream>
using namespace std;
class base {
     int x;
public:
     void setx  ( int n ) { x = n; }
     void showx ( )       { cout << x << "\n"; }
// Класс наследуется как открытый
class derived: public base
```

```
{
     int y;
public:
     void sety  ( int n ) { y = n; }
     void showy ( )       { cout << y << "\n"; }
};
int main ( ) {
     derived ob;
     ob.setx  ( 10 ); // доступ к члену базового класса
     ob.sety  ( 20 ); // доступ к члену производного класса
     ob.showx ( );    // доступ к члену базового класса
     ob.showy ( );    // доступ к члену производного класса
     return 0;
}
```

4.1. Защищенные члены класса

Как вы узнали из предыдущего раздела, у производного класса нет доступа к закрытым членам базового. Это означает, что если производному классу необходим доступ к некоторым членам базового, то эти члены должны быть открытыми. Однако возможна ситуация, когда необходимо, чтобы члены базового класса, оставаясь закрытыми, были доступны для производного класса. Для реализации этой идеи в C++ включен спецификатор доступа protected (защищенный). Спецификатор доступа protected эквивалентен спецификатору private с единственным исключением: защищенные члены базового класса доступны для членов всех производных классов этого базового класса. Вне базового или производных классов защищенные члены недоступны.

Когда базовый класс наследуется производным классом как открытый (public), защищенный член базового класса становится защищенным членом производного класса. Когда базовый класс наследуется как закрытый (private), то защищенный член базового класса становится закрытым членом производного класса. Базовый класс может также наследоваться производным классом как защищенный (protected). В этом случае открытые и защищенные члены базового класса становятся защищенными членами производного класса. (Естественно, что закрытые члены базового класса остаются закрытыми, и они не доступны для производного класса.)

4.2. Множественное наследование

Имеются два способа, посредством которых производный класс может наследовать более одного базового класса. Во-первых, производный класс может использоваться в качестве базового для другого производного класса, создавая многоуровневую иерархию классов. В этом случае говорят, что исходный базовый класс является косвенным (indirect) базовым классом для второго производного класса. (Отметьте, что любой класс — независимо от того, как он создан — может использоваться в качестве базового класса.) Во-вторых, производный класс может прямо наследовать более одного базового класса. В такой ситуации созданию производного класса помогает комбинация двух или более базовых классов. Ниже исследуются результаты, к которым приводит наследование нескольких базовых классов. Когда класс используется как базовый для производного, который, в свою очередь, является базовым для другого производного класса, конструкторы всех трех классов вызываются в порядке наследования. (Это положение является обобщением ранее исследованного принципа.) Деструкторы вызываются в обратном порядке. Таким образом, если класс B1 наследуется классом Dl, а D1 — классом D2, то конструктор класса B1 вызывается первым, за ним конструктор класса D1, за которым, в свою очередь, конструктор класса D2. Деструкторы вызываются в обратном порядке. Если производный класс напрямую наследует несколько базовых классов, используется такое расширенное объявление:

class имя_проязводяого_класса: сп_доступа имя_базового_класса1, сп_доступа. имя_базового_класса2, ..., сп_доступа имя_базового-класса N

{

... тело класса

};

Здесь имя_бозового_класса1 ... имя_базового__класса N — имена базовых классов, сп_доступа — спецификатор доступа, который может быть разным у разных базовых классов. Когда наследуется несколько базовых классов, конструкторы выполняются слева направо в том порядке, который задан в объявлении производного класса. Деструкторы выполняются в обратном порядке. Когда класс наследует несколько базовых классов, конструкторам которых необходимы аргументы, производный класс передает эти аргументы, используя расширенную форму объявления конструктора производного класса:

констр_прокзв_класса (список_арг) : имя_базового_класса1 (список_арг) , имя_базового_класса2 (список_арг) , . . . , имя_базового_классаN (список_арг)

{

... тело конструктора производного класса

}

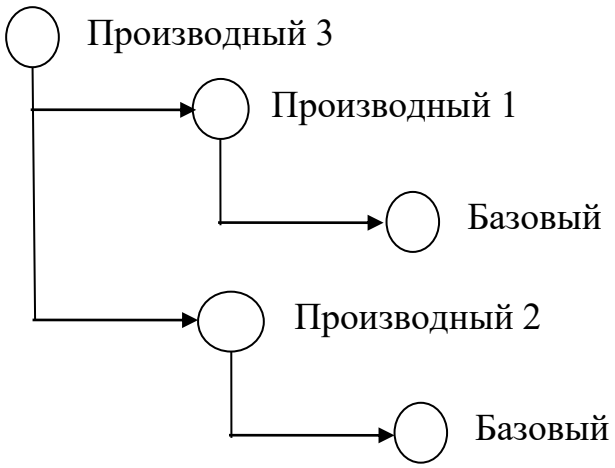
Здесь имя_бозового_класса1 ... имя_базового_классаN — имена базовых классов. Если производный класс наследует иерархию классов, каждый производный класс должен передавать предшествующему в цепочке базовому классу все необходимые аргументы.

Пример

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

class base {
public:
  int i;
};

class derived1 :  public base {
public:
  int j;
};

class derived2 : public base {
public:
  int k;
};

class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10;                          // Неопределенность!!!
  ob.j = 20;
  ob.k = 30;

  ob.sum = ob.i + ob.j + ob.k;      // Неопределенность!!!

  cout << ob.i << " ";             // Неопределенность!!!

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

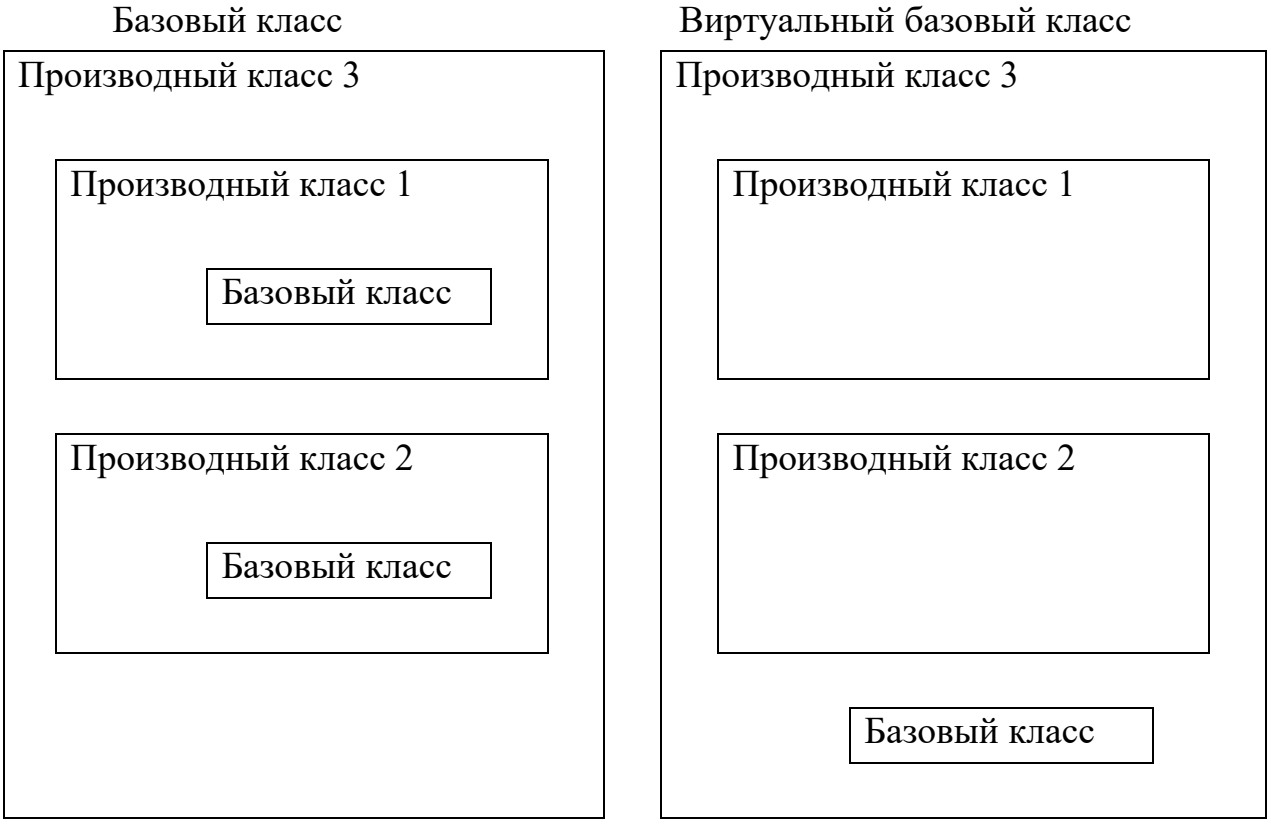
4.3. Виртуальные базовые классы

При многократном прямом наследовании производным классом одного и того же базового класса может возникнуть проблема. Чтобы понять, что это за проблема, рассмотрим следующую иерархию классов:



Здесь базовый класс Базовый наследуется производными классами Производный! и Производный2. Производный класс Производный3 прямо наследует производные классы Производный! и Производный2. Однако это подразумевает, что класс Базовый фактически наследуется классом Производный3 дважды — первый раз через класс Производный!, а второй через класс Производный!. Однако, если член класса Базовый будет использоваться в классе Производный3, это вызовет неоднозначность. Поскольку в классе Производный3 имеется две копии класса Базовый, то будет ли ссылка на элемент класса Базовый относиться к классу Базовый, наследуемому через класс Производный!, или к классу Базовый, наследуемому через класс Производный21 Для преодоления этой неоднозначности в C++ включен механизм, благодаря которому в классе Производный3 будет включена только одна копия класса Базовый. Класс, поддерживающий этот механизм, называется виртуальным базовым классом (virtual base class). В таких ситуациях, когда производный класс более одного раза косвенно наследует один и тот же базовый класс, появление двух

копий базового класса в объекте производного класса можно предотвратить, если базовый класс наследуется как виртуальный для всех производных классов. Такое наследование не дает появиться двум (или более) копиям базового класса в любом следующем производном классе, косвенно наследующем базовый класс. В этом случае перед спецификатором доступа базового класса необходимо поставить ключевое слово virtual.



Пример

В этом примере для предотвращения появления в классе derived3 двух копий класса base используется виртуальный базовый класс.

```
// В этой программе используется виртуальный базовый класс
#include <iostream>
using namespace std;
class base {
public:
   int i;
};
// Наследование класса base как виртуального
class derivedl: virtual public base {
public:
     int j;
};
```

```
// Здесь класс base тоже наследуется как виртуальный
class derived2: virtual public base {
public:
      int k;
};
/* Здесь класс derived3 наследует как класс derivedl, так и класс de-
rived2. Однако в классе derived3 создается только одна копия класса base
*/
class derived3: public derivedl, public derived2 {
public:
      int product () { return i * j * k; }
};
int main ( ) {
      derived3 ob; // Здесь нет неоднозначности, поскольку
                   // представлена только одна копия класса base
                   // ob.i = 10; ob.j = 3; ob.k = 5;
      cout << "Результат равен " << ob. product () << '\n';
      return 0;
}
```

Если бы классы derivedl и derived2 наследовали класс base не как виртуальный, тогда инструкция

```
ob.i = 10;
```

вызывала бы неоднозначность и при компиляции возникла бы ошибка.

4.4. Указатель на объект производного класса

Указатель, объявленный в качестве указателя на базовый класс, также может использоваться, как указатель на любой класс, производный от этого базового. В такой ситуации представленные ниже инструкции являются правильными:

```
base * p;              // указатель базового класса
base base_ob;          // объект базового класса
derived derived ob; // объект производного класса
// Естественно, что указатель p может указывать на объект базового класса
p = & base_ob;         // указатель p для объекта базового класса
// Кроме базового класса указатель p может указывать
// на объект производного класса
p = & derived_ob;      // указатель p для объекта производного класса
```

Указатель базового класса может указывать на объект любого класса, производного от этого базового и при этом ошибка несоответствия типов генерироваться не будет. Для указания на объект производного класса можно вос-

пользоваться указателем базового класса, при этом доступ может быть обеспечен только к тем объектам производного класса, которые были унаследованы от базового. Объясняется это тем, что базовый указатель "знает" только о базовом классе и ничего не знает о новых членах, добавленных в производном классе. Указатель базового класса можно использовать для указания на объект производного класса, но обратный порядок недействителен. Указатель производного класса нельзя использовать для доступа к объектам базового класса. Арифметика указателей связана с типом данных (т. е. классом), который задан при объявлении указателя. Таким образом, если указатель базового класса указывает на объект производного класса, а затем инкрементируется, то он уже не будет указывать на следующий объект производного класса. Этот указатель будет указывать на следующий объект базового класса.

Пример

```
#include <iostream>
#include <cstring> // for older compilers, use <string.h>
using namespace std;

class B_class {
  char author[80];
public:
  void put_author(char *s) { strcpy(author, s); }
  void show_author() { cout << author << "\n"; }
} ;

class D_class : public B_class {
  char title[80];
public:
  void put_title(char *num) {
    strcpy(title, num);
  }
  void show_title() {
    cout << "Title: ";
    cout <<  title << "\n";
  }
};

int main()
{
  B_class *p;
  B_class B_ob;

  D_class *dp;
  D_class D_ob;

  p = &B_ob;  // address of base

  // Access B_class via pointer.
```

```
  p->put_author("Tom Clancy");

  // Access D_class via base pointer.
  p = &D_ob;
  p->put_author("William Shakespeare");

  // Show that each author went into proper object.
  B_ob.show_author();
  D_ob.show_author();
  cout << "\n";

  /* Since put_title() and show_title() are not part
     of the base class, they are not accessible via
     the base pointer p and must be accessed either
     directly, or, as shown here, through a pointer to the
     derived type.
  */
  dp = &D_ob;
  dp->put_title("The Tempest");
  p->show_author(); // either p or dp can be used here.
  dp->show_title( );

  return 0;
}
```

Указатель на базовый можно использовать для доступа к элементам производного класса после приведения типа.

```
( ( D_class * ) p ) -> show_title ();
```

5. ПОЛИМОРФИЗМ

5.1. Пояснения к полиморфизму

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

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

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

При проектировании иерархии классов, некоторые методы сохраняя названия, изменяются по сути (меняется алгоритм реализации).

Полиморфизм – механизм обеспечивающий возможность определения различных реализации метода одним названием для классов различных уровней иерархии.

5.2. Виртуальный метод

Виртуальный метод (virtual method) является членом класса. Он объявляется внутри базового класса и переопределяется в производном классе. Для того, чтобы метод стал виртуальным, перед объявлением метода ставится ключевое слово virtual. Если класс, содержащий виртуальный метод, наследуется, то в производном классе виртуальный метод переопределяется. Виртуальный метод внутри базового класса определяет вид интерфейса этого метода. Каждое переопределение виртуального метода в производном классе определяет ее реализацию, связанную со спецификой производного класса. Таким образом, переопределение создает конкретный метод. При переопределении виртуального метода в производном классе, ключевое слово virtual не требуется. Виртуальный метод может вызываться так же, как и любой другой метод.

Однако наиболее интересен вызов виртуального метода через указатель, благодаря чему поддерживается динамический полиморфизм. Если указатель

базового класса ссылается на объект производного класса, который содержит виртуальный метод и для которого виртуальный метод вызывается через этот указатель, то компилятор определяет, какую версию виртуального метода вызвать, основываясь при этом на типе объекта, на который ссылается указатель. При этом определение конкретной версии виртуального метода имеет место не в процессе компиляции, а в процессе выполнения программы. Другими словами, тип объекта, на который ссылается указатель, и определяет ту версию виртуального метода, который будет выполняться. Поэтому, если два или более различных класса являются производными от базового, содержащего виртуальный метод, то, если указатель базового класса ссылается на разные объекты этих производных классов, выполняются различные версии виртуального метода. Этот процесс является реализацией принципа динамического полиморфизма. Фактически, о классе, содержащем виртуальный метод, говорят как о полиморфном классе (polymorphic class).

Пример

```
class base {
public:
  virtual void who () { cout << "Base\n"; }
};
class first_d : public base {
public:
  void who () { cout << "First derivation\n"; }
};
class second_d : public base {
public:
  void who () { out << "Second derivation\n"; }
};

int main ()
{
  base      base_obj;
  base    * p;
  first_d   first_obj;
  second_d  second_obj;

  p = & base_obj;
  p -> who ();  // access base's who

  p = & first_obj;
  p -> who (); // access first_d's who

  p = & second_obj;
  p -> who ();  // access second_d's who

  return 0;
}
```

5.3. Наследование виртуального метода

Если метод объявляется как виртуальный, он остается таковым независимо от того, через сколько уровней производных классов он может пройти.

Если виртуальный метод в классе не переопределен, то отрабатывает переопределенный ближайший по иерархии классов.

Пример

```
class base {
public:
  virtual void who () { cout << "Base\n"; }
};
class first_d : public base {
public:
  void who () { cout << "First derivation\n"; }
};
class second_d : public first_d {
public:
};

int main ()
{
  base      base_obj;
  base    * p;
  first_d   first_obj;
  second_d  second_obj;

  p = & base_obj;
  p -> who ();  // access base's who

  p = & first_obj;
  p -> who (); // access first_d's who

  p = & second_obj;
  p -> who ();  // access first_d's who

  return 0;
}
```

5.4. Чисто виртуальные методы и абстрактные классы

Иногда, когда виртуальный метод объявляется в базовом классе, он не выполняет никаких значимых действий. Это вполне обычная ситуация, поскольку часто в базовом классе законченный, полнофункциональный объект не

описывается. Вместо этого в нем просто содержится базовый набор методов и переменных, для которых в производном классе определяется все недостающее. Когда в виртуальном методе базового класса отсутствует значимое действие, в любом классе, производном, от этого базового, такой метод обязательно должен быть переопределен. Для реализации этого положения в C++ поддерживаются так называемые чистые виртуальные методы (pure virtual function). Чистые виртуальные методы не имеют кода реализации в базовом классе. Туда включаются только прототипы этих методов. Для чистого виртуального метода используется такая основная форма:

virtual «тип» «имя метода» (список параметров) = 0;

Ключевой частью этого объявления является приравнивание метода нулю. Это сообщает компилятору, что в базовом классе не существует тела метода.

Чисто виртуальный метод – это виртуальный метод, который не имеет реализации в базовом классе.

Если метод задан как чисто виртуальный, это предполагает, что он обязательно должен подменяться в каждом производном классе. Если этого нет, то при компиляции возникнет ошибка. Таким образом, создание чистых виртуальных методов — это путь, гарантирующий, что производные классы обеспечат их переопределение. Если класс содержит хотя бы один чисто виртуальный метод, то о нем говорят как об абстрактном классе (abstract class). Поскольку в абстрактном классе содержится, по крайней мере, один метод, у которого отсутствует тело метода, технически такой класс неполон, и ни одного объекта этого класса создать нельзя. Таким образом, абстрактные классы могут быть только наследуемыми. Они никогда не бывают изолированными. Важно понимать, однако, что по-прежнему можно создавать указатели абстрактного класса, благодаря которым достигается динамический полиморфизм. (Также допускаются и ссылки на абстрактный класс.) Если виртуальный метод наследуется, то это соответствует его виртуальной природе. Это означает, что если производный класс используется в качестве базового для другого производного класса, то виртуальный метод может подменяться в последнем производном классе (так же, как и в первом производном классе). Например, если базовый класс B со-

держит виртуальный метод f( ), и класс D1 наследует класс B, а класс D2 наследует класс D1, тогда метод f( ) может подменяться как в классе D1, так и в классе D2.

Абстрактный класс – класс, который содержит хотя бы одну чисто виртуальный метод.

5.5. Перегрузка функций

Перегрузка функции – это механизм, который позволяет двум родственным функциям иметь одинаковые имена.

Несколько функций могут иметь одинаковые имена, но при условии, что их параметры будут различными. Это один из способов реализации полиморфизма.

```
void f ( int i);         // один целочисленный параметр
void f ( int i, int j); // два целочисленных параметра
void f ( double k);      // один действительный парамтер

int main()
{
  f ( 10 );      // вызов f ( int )

  f ( 10, 20 ); // вызов f ( int, int )

  f ( 12.23 );  // вызов f ( double )

  return 0;
}
```

5.5.1.Аргументы, передаваемые функции по умолчанию

Аргумент по умолчанию позволяет, если при вызове функции соответствующий аргумент не задан, присвоить параметру значение по умолчанию. Применение аргумента по умолчанию является скрытой формой перегрузки функций. Чтобы передать параметру аргумент по умолчанию, нужно в инструкции определения функции приравнять параметр тому значению, которое вы хотите передать, когда при вызове функции соответствующий аргумент не будет указан. Например, в представленной ниже функции двум параметрам по умолчанию присваивается значение 0:

void f (int a = 0, int b = 0) ;

Данный синтаксис напоминает инициализацию переменных. Теперь эту функцию можно вызвать тремя различными способами. Во-первых, она может вызываться с двумя заданными аргументами. Во-вторых, она может вызываться только с первым заданным аргументом. В этом случае параметр b по умолчанию станет равным нулю. Наконец, функция f() может вызываться вообще без аргументов, при этом параметры a и b по умолчанию станут равными нулю. Таким образом, все следующие вызовы функции f() правильны:

```
F ( );          // a и b по умолчанию равны 0
f { 10 );       // a равно 10, b по умолчанию равно 0
f ( 10, 99 ); // a равно 10, b равно 99
```

Невозможно передать по умолчанию значение a и при этом задать b. Когда вы создаете функцию, имеющую один или более передаваемых по умолчанию аргументов, эти аргументы должны задаваться только один раз: либо в прототипе функции, либо в ее определении, если определение предшествует первому использованию функции. Аргументы по умолчанию нельзя задавать одновременно в определении и в прототипе функции. Это правило остается в силе, даже если вы просто дублируете одни и те же аргументы по умолчанию. Все параметры, задаваемые по умолчанию, должны указываться правее параметров, передаваемых обычным путем. Еще несколько слов об аргументах по умолчанию: они должны быть константами или глобальными переменными. Они не могут быть локальными переменными или другими параметрами.

5.5.2.Определение адреса перегруженной функции

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

p = zap;

В языке С любой тип указателя может использоваться как указатель на функцию, поскольку имеется только одна функция, на которую он может ссылаться. Однако в С++ ситуация несколько более сложная, поскольку функция может быть перегружена. Таким образом, должен быть некий механизм, который позволял бы определять адреса перегруженных версий функции. Решение оказывается не только элегантным, но и эффектным. Способ объявления указателя и определяет то, адрес какой из перегруженных версий функции будет получен. Уточним, объявления указателей соответствуют объявлениям перегруженных функций. Функция, объявлению которой соответствует объявление указателя, и является искомой функцией.

Здесь представлена программа, которая содержит две версии функции spaceQ. Первая версия выводит на экран некоторое число пробелов, заданное в переменной count. Вторая версия выводит на экран некоторое число каких-то иных символов, вид которых задан в переменной ch. В функции main() объявляются оба указателя на эти функции. Первый задается как указатель на функцию, имеющую только один целый параметр. Второй объявляется как указатель на функцию, имеющую два параметра. /* Иллюстрация присваивания и получения указателей на перегруженные функции */

```
#include <iostream>
using namespace std;
// вывод заданного в переменной count числа пробелов
void space (int count) {
    for ( ; count; count — ) cout << ' ';
}
// вывод заданного в переменной count числа символов,
// вид которых задан в переменной ch
void space (int count, char ch) {
    for (; count; count — ) cout << ch;
}

int main () {
    // Создание указателя на функцию с одним целым
    // параметром.
```

```
    void (*fpl) (int) ;
    // Создание указателя на функцию с одним целым и
    // одним символьным параметром.
    void (*fp2) (int, char);
    fpl = space; // получение адреса функции space (int)
    fp2 = space; // получение адреса функции space (int,
char)
    fpl (22); // выводит 22 пробела cout << " | \n";
    fp2 (30, 'x'); // выводит 30 символов x cout << "
\n";
    return 0;
    }
```

Как показано в комментариях, на основе того, каким образом объявляются указатели flpl и fp2, компилятор способен определить, какой из них на какую из перегруженных функций будет ссылаться. Повторим, если вы присваиваете адрес перегруженной функции указателю на функцию, то объявление указателя определяет, адрес какой именно функции ему присваивается. Более того, объявление указателя на функцию должно точно соответствовать одной и только одной перегруженной функции. Если это не так, будет внесена неоднозначность, что приведет к ошибке при компиляции программы.

5.6. Перегрузка методов

5.6.1.Определение адреса перегруженного метода

5.7. Перегрузка операторов

6. ШАБЛОНОЫ ФУНКЦИЙ И КЛАССОВ

Используя механизм шаблонов, можно создавать обобщенные функции и классы. В обобщенной функции (или классе) обрабатываемый ею (им) тип данных задается как параметр. Таким образом, одну функцию или класс можно использовать для разных типов данных, не предоставляя явным образом конкретные версии реализации для каждого типа данных.

Шаблоны дают возможность определять при помощи одного фрагмента кода целый набор взаимосвязанных функций (перегруженных), называемых шаблонными функциями, или набор связанных классов, называемых шаблонными классами.

Другими словами, обобщенная функция – это функция перегружающая сама себя.

Например, можно написать один шаблон функции сортировки массива, на основе которого C++ будет автоматически генерировать отдельные шаблонные функции, сортирующие массивы типов int, float, массив строк и т.д.

6.1. Шаблонны функций

Перегруженные функции обычно используются для выполнения похожих операций над различными типами данных. Если для каждого типа данных должны выполняться идентичные операции, то оптимальным решением является использование шаблонов функций. При этом программист должен написать всего одно описание шаблона функции. Основываясь на типах аргументов, использованных при вызове этой функции, компилятор будет автоматически генерировать объектные коды функций, обрабатывающих каждый тип данных. Шаблоны функций, являясь компактным решением, позволяют компилятору полностью контролировать соответствие типов.

Описание шаблона функции

```
template <формальные параметры > «тип» «имя функции» ( список параметров )
{
    // тело функции
}
```

Каждому формальному параметру предшествует ключевое слово class

или typename. Формальные параметры в описании шаблона используются для определения типов параметров функции, типа возвращаемого функцией значения и типов переменных, объявляемых внутри функции. Далее, за этим заголовком, следует обычное описание функции. Ключевое слово class или typename, используемое в шаблоне функции при задании типов параметров, означает «любой встроенный тип или тип, определяемый пользователем».

```
template <class T>
T& inc_value ( T & val ) {
    ++val; return val;
}

int main ( )
{
    int x = 0;
    x = ( int )inc_value < int > ( x );
    cout << x << endl;
    char c = 0;
    c = ( char ) inc_value < char > ( c );
    cout << c << endl;
    return 0;
}

template < class T1 >
void PrintArray ( const T1 * array, const int count )
{
    for ( int i = 0; i < count; i++ )
        cout << array [ i ] << " ";
    cout << endl;
}
```

В шаблоне функции PrintArray() объявляется формальный параметр T1 для массива, который будет выводиться функцией PrintArray(). T1* называется параметром типа. Когда компилятор обнаруживает в тексте программы вызов функции PrintArray(), он заменяет T1 во всей области определения шаблона тип первого параметра функции PrintArray() и C++ создает шаблонную функцию вывода массива указанного типа данных. После этого вновь созданная функция компилируется.

```
int main ( )
{
    const int aCount = 5;
    const int bCount = 7;
    const int cCount = 6;
    int    a [ aCount ] = { 1,2,3,4,5 };
    double b [ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
    char  c [ cCount ] = "HELLO"; //6-я позиция для null
    cout << "Array a:" << endl;
    PrintArray ( a, aCount ); // шаблон для integer
    cout << "Array b:" << endl;
    PrintArray ( b, bCount );  // шаблон для double
    cout << "Array c:" << endl;
    PrintArray ( c, cCount );  // шаблон для character
    return 0;
}
```

Шаблоны функций расширяют возможности многократного использования программного кода, но программа может создавать слишком много копий шаблонных функций и шаблонных классов. Для этих копий могут потребоваться значительные ресурсы памяти. В примере шаблонный механизм позволяет программисту избежать необходимости написания трех отдельных функций с прототипами:

```
void PrintArray ( const int*,    const int );
void PrintArray ( const double*, const int );
void PrintArray ( const char*,   const int );
```

которые используют один и тот же код, кроме кода для типа T1.

6.2. Перегрузка шаблонных функций

Шаблонные функции и перегрузка функций тесно связаны друг с другом. Все родственные функции, полученные из шаблона, имеют одно и то же имя; поэтому компилятор использует механизм перегрузки для того, чтобы обеспечить вызов соответствующей функции. Сам шаблон функции может быть перегружен несколькими способами. Во–первых, можно определить другие шаблоны, имеющие то же самое имя функции, но различные наборы параметров. Например,

```
template < class T1 >
void PrintArray ( const T1 * array, const int count)
{
    for ( int i = 0; i < count; i++ )
        cout << array[i]<<" ";
    cout << endl;
}

template < class T1 >
void PrintArray ( const T1 * array,
                  const int  lowSubscript,
                  const int  highSubscript )
{
    for ( int i = lowSubscript; i <= highSubscript; i++ )
        cout << array [ i ] << " ";
    cout << endl;
}

int main ( )
{
    const int aCount = 5;
    const int bCount = 7;
    const int cCount = 6;
    int    a [ aCount ] = { 1,2,3,4,5 };
    double b [ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
    char   c [ cCount ] = "HELLO"; //6-я позиция для null
    …
    cout << "Array a from 1 to 3:" << endl;
    PrintArray ( a, 2 );       // шаблон для integer
    cout << "Array b from 4 to 7:" << endl;
    PrintArray ( b, 3, 6 );    // шаблон для double
    cout << "Array c from 3 to 5:" << endl;
    PrintArray ( c, 2, 4 );    // шаблон для character
    return 0;
}
```

Шаблон функции может быть также перегружен, если ввести другую нешаблонную функцию с тем же самым именем, но другим набором параметров. Например, шаблон функции PrintArray можно перегрузить версией нешаблонной функции, которая выводит массив символов в столбик.

```
template < class T1 >
void PrintArray ( const T1 * array, const int count )
{…}

template < class T1 >
void PrintArray ( const T1 * array,
                  const int lowSubscript,
                  const int highSubscript )
{…}

void PrintArray ( char * array, const int count )
{
    for ( int I = 0; I < count; i++ )
        cout << array [ I ] << endl;
}

int main ( )
{
    const int cCount=6;
    char c [ cCount ] = "HELLO"; //6-я позиция для null
    cout << "Array c:" << endl;
    PrintArray ( c, cCount );
    …
    return 0;
}
```

Компилятор действует по следующему алгоритму. Сначала он пытается найти и использовать функцию, которая точно соответствует по своему имени и типам параметров вызываемой функции. Если такая функция не находится, то компилятор ищет шаблон функции, с помощью которого он может сгенерировать шаблонную функцию с точным соответствием типов параметров и имени функции. Если такой шаблон обнаруживается, то компилятор генерирует и использует соответствующую шаблонную функцию.

6.3. Шаблоны классов

По аналогии обобщенной функцией определяются обобщенные классы. Для этого создается класс, в котором определяются все используемые им алго-

ритмы, при этом реальный тип обрабатываемых в нем данных будет задан как параметр при создании объектов этого класса.

Обобщенные классы особенно полезны в случае, когда в них используется логика, которую можно обобщить. Например, алгоритмы, которые поддерживают функционирование очереди целочисленных значений, также подходят и для очереди символов.

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

Общий формат объявления обобщенного класса имеет следующий вид:

```
template < class Ttype > class «имя класса»
{
    // описание класса
}
```

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

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

Здесь «тип» определяет имя типа данных, который будет обрабатываться объектом класса. Функции члены класса, автоматически являются обобщенными.

```
// Демонстрация класса очереди queue.
#include <iostream>
using namespace std;

const int SIZE = 100;

// Описание класса queue.
template <class QType> class queue {
  QType q [ SIZE ];
  int   sloc, rloc;
public:
  queue ( )  { sloc = rloc = 0; }
  void  qput ( QType i );
  QType qget ( );
};

// Занесение объекта в очередь.
template < class QType > void queue < QType > :: qput ( QType i )
```

```
{
    if ( sloc == SIZE ) {
        cout << "Очередь заполнен.\n";
        return;
    }
    Sloc ++;
    q [ sloc ] = i;
}

// Извлечение объекта из очереди.
template < class QType > QType queue < QType > :: qget ( )
{
    if ( rloc == sloc ) {
        cout << "Очередь пуст.\n";
        return 0;
    }
    Rloc ++;
    return q [ rloc ];
}

int main ( )
{
    Queue < int > a, b;  // две очереди для целых чисел

    a.qput ( 10 );
    b.qput ( 19 );

    a.qput ( 20 );
    b.qput (  1 );

    cout << a.qget ( ) << " ";
    cout << a.qget ( ) << " ";
    cout << b.qget ( ) << " ";
    cout << b.qget ( ) << "\n";

    queue < double > c, d; // две очереди для действительных чисел

    c.qput ( 10.12 );
    d.qput ( 19.99 );

    c.qput ( -20.0   );
    d.qput (   0.986 );

    cout << c.qget ( ) << " ";
    cout << c.qget ( ) << " ";
    cout << d.qget ( ) << " ";
    cout << d.qget ( ) << "\n";

    return 0;
}
```

При выполнении этой программы получаем следующий результат:

```
10 20 19 1
10.12 -20 19.99 0.986
```

Тип данных, хранимых в очереди, обобщен в объявлении класса. Он неизвестен до тех пор, пока не будет объявлен объект класса, который и определит реальный тип данных.

Обобщенные функции и классы представляют собой мощные средства, которые помогают увеличить эффективность работы программиста. Обобщенные функции и классы избавляют от утомительного труда по созданию отдельных реализаций для каждого типа данных, подлежащих обработке единым алгоритмом.

6.4. Использование параметров, не являющихся типами

В описании шаблона можно также задать параметры не являющиеся типами (стандартный параметр). Синтаксис включает определение типа и имени параметра. Пример.

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

// Здесь, int size это параметр не являющийся типом.
template < class AType, int size > class atype
{
    AType a [ size ]; // параметр size передает длину массива
public:
    atype ( ) {
        register int i;
        for ( i = 0; i < size; i++ ) a [ i ] = i;
    }
    AType & operator[] ( int i );
};

// Обеспечение контроля границ.
template < class AType, int size >
AType & atype < AType, size > :: operator[] ( int i )
{
    if ( i < 0 || I > size - 1 ) {
        cout << "\nЗначение индекса ";
        cout << i << " за пределами границ массива.\n";
        exit ( 1 );
    }
    return a [ i ];
```

```
}

int main ( )
{
    Atype < int,    10 > intob;    // 10-элементный массив целых чисел
    Atype < double, 15 > doubleob; // 15-элементный массив действительных
                                   //    чисел
    int i;

    cout << "Целочисленный массив: ";
    for ( i = 0; i < 10; i++ ) intob [ i ] = i;
    for ( i = 0; i < 10; i++ ) cout << intob [ i ] << "  ";
    cout << '\n';

    cout << "Действительный массив: ";
    for ( i = 0; i < 15; i++ ) doubleob [ i ] = ( double ) i/3;
    for ( i = 0; i < 15; i++ ) cout << doubleob [ i ] << "  ";
    cout << '\n';

    intob [ 12 ] = 100; // Ошибка времени выполнения

    return 0;
}
```

С помощью шаблона классов можно достаточно просто определить и реализовать без потерь в эффективности выполнения программы и, не отказываясь от статического контроля типов, такие контейнерные классы, как списки и ассоциативные массивы. Кроме того, шаблоны класса позволяют определить сразу для целого семейства типов обобщенные (генерические) функции, например, такие, как sort (сортировка). Рассмотрим простой шаблон класса stack (стек). Понятие «стек» не зависит от типа данных, которые помещаются в стек. Тип данных должен быть задан только тогда, когда приходит время создать стек «фактически». Поэтому достаточно создать некое общее описание понятия стека и на основе этого родового класса создавать классы, являющиеся специфическими версиями для конкретного типа данных. Шаблоны классов называются еще параметризованными типами, так как они имеют один или большее количество параметров типа, определяющих настройку подового шаблона класса на специфический тип данных при создании объекта класса.

```
 template<class T> class stack
{
    T* v;
    T* p;
```

```
    int sz;
public:
    stack(int s)
{
    v = p = new T[sz=s];
}
~stack()
{
    delete[] v;
}
void push(T a)
{
    * p++ = a;
}

T pop()
{
    return *--p;
}
int size() const { return p-v; }
};
```

Префикс template<class T> указывает, что описывается шаблон класса с параметром T, обозначающим тип классов Stack которые будут создаваться на основе этого шаблона. Идентификатор T определяет тип данных– элементов, хранящихся в стеке. Имя шаблонного класса, за которым следует тип, заключенный в угловые скобки <>, является именем класса (определяемым шаблоном класса), и его можно использовать как все имена класса.

Например, ниже определяется объект sc класса stack<char>:

```
stack<char> sc(100);   // стек символов
```

Если не считать особую форму записи имени, класс stack<char> полностью эквивалентен классу определенному так:

```
class stack_char
{
    char* v;
    char* p;
    int sz;
public:
    stack_char(int s)
    {
        v = p = new char[sz=s];
    }
    ~stack_char()
    {
        delete[] v;
```

```
    }
    void push(char a)
    {
        *p++ = a;
    }
    char pop()
    {
        return *--p;
    }
    int size() const { return p-v; }
};
```

Имея определение шаблонного класса stack, можно следующим образом определять и использовать различные стеки: typedef struct _Point {int x,y;}Point;

```
class shape {};
void f(stack<Complex>& sc)  // параметр типа 'ссылка на complex'
{
    stack<shape*> ssp(200);  // стек указателей на фигуры
    stack<Point> sp(400);    // стек структур Point

    sc.push(Complex(1,2));
    Complex z=sc.pop();
    z*2.5;
    cout<<z;
    stack<int>*p = 0;          // указатель на стек целых
    p = new stack<int>(800);  // стек целых размещается в свободной памя-
ти
    for ( int i = 0; i<400; i++)
    {
        p->push(i);
        Point p1;
        p1.x=i;
        p1.y=i+400;
        sp.push(p1);
    }
    cout<<sp.size()<<endl; //…
}
```

В случае внешней реализацию методов шаблона классов stack сам шаблон определяется следующим образом:

```
template<class T> class stack
{
    T* v;
    T* p;
    int sz;
public:
    stack(int);
    ~stack();
    void push(T);
    T pop();
    int size() const;
```

```
};
```

В этом случае определение методов шаблона классов stack выполняется, как и для функций-членов обычных, нешаблонных классов. Подобные функции так же параметризируются типом, служащим параметром для их шаблонного класса, поэтому определяются они с помощью шаблона класса для функции. Если это происходит вне шаблонного класса, это надо делать явно:

```
template<class T>
void stack<T>::push(T a)
{
    *p++ = a;
}
template<class T>
T stack<T>::pop()
{
    return *--p;
}
template<class T>
int stack<T>::size() const {return p-v;}

template<class T>
stack<T>::stack(int s)
{
    v = p = new T[sz=s];
}

template<class T>
stack<T>::~stack()
{
    delete[] v;
}
```

Для приведенного примера (функция f) компилятор должен создать определения конструкторов для классов stack<shape*>, stack<Point>, stack<Complex> и stack<int>, деструкторов для stack<Complex>, stack<shape*> и stack<Point>, версии функций push() для stack<complex>, stack<int> и stack<Point> и версию функции pop() для stack<complex>. Такие создаваемые функции будут совершенно обычными методами, например: void stack<complex>::push(complex a) { *p++ = a; } Здесь отличие от обычного метода только в форме имени класса. Точно так же, как в программе может быть только одно определение метода класса, возможно только одно для метода шаблонного класса. Важно составлять определение шаблона классов таким образом, чтобы его зависимость от глобальных данных была минимальной. Дело

в том, шаблон классов будет использоваться для порождения функций и классов на основе заранее неизвестного типа и в неизвестных контекстах. Практически любая, даже слабая зависимость от контекста может проявиться как проблема при отладке программы пользователем, который, вероятнее всего, не был создателем шаблона классов.

6.5. Объявление элементов класса спецификацией static

Статические свойства (поля, переменные).

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

Статические поля

Статические свойства применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

Свойства статических полей:

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

```
#include <iostream.h>
class Example
{
public:
     static int value; //объявление в классе
};

int Example::value; //определение статического поля в глобальной
                    //области, по умолчанию инициализируется нулем.
// int Example::value=10; // пример инициализации произвольным
                          // значением
```

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

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

3) на статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как private, нельзя изменить с по-

мощью операции доступа к области действия; это можно сделать только с помощью статических методов;

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

Статические методы

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

7. СТРУКТУРЫ И ОБЪЕДИНЕНИЯ

Синтаксически класс похож на структуру. Класс и структура имеют фактически одинаковые свойства. В C++ определение структуры расширили таким образом, что туда, как и в определение класса, удалось включить функции-члены, в том числе конструкторы и деструкторы. Таким образом, единственным отличием между структурой и классом является то, что члены класса, по умолчанию, являются закрытыми, а члены структуры — открытыми. Расширенный синтаксис описания структуры:

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

В соответствии с формальным синтаксисом C++ как структура, так и класс создают новые типы данных. В том, что структуры и классы обладают фактически одинаковыми свойствами, имеется кажущаяся избыточность. Те, кто только знакомится с C++, часто удивляются этому дублированию. При таком взгляде на проблему уже не кажутся необычными рассуждения о том, что ключевое слово class совершенно лишнее. Объяснение этому может быть дано в "строгой" и "мягкой" формах. "Строгий" довод состоит в том, что необходимо поддерживать линию на совместимость с C. Стиль задания структур C совершенно допустим и для программ C++. Поскольку в C все члены структур по умолчанию открыты, это положение также поддерживается и в C++. Кроме этого, поскольку класс синтаксически отличается от структуры, определение класса открыто для развития в направлении, которое в конечном итоге может привести к несовместимости со взятым из C определением структуры. Если эти два пути разойдутся, то направление, связанное с C++, уже не будет избыточным. "Мягким" доводом в пользу наличия двух сходных конструкций стало отсутствие какого-либо ущерба от расширения определения структуры в C++ таким образом, что в нее стало возможным включение функций-членов. Хотя структуры имеют схожие с классами возможности, большинство программистов ограничивают использование структур взятыми из C формами и не применяют их для задания функций-членов. Для задания объекта, содержащего данные и

код, эти программисты обычно указывают ключевое слово class. Однако все это относится к стилистике и является предметом собственного выбора.

В C++ объединение также представляет собой тип класса, в котором функции и данные могут содержаться в качестве его членов. Объединение похоже на структуру тем, что в нем по умолчанию все члены открыты до тех пор, пока не указан спецификатор private. Главное же в том, что в C++ все данные, которые являются членами объединения, находятся в одной и той же области памяти (точно так же, как и в C). Объединения могут содержать конструкторы и деструкторы. Объединения C++ совместимы с объединениями C. Если в отношениях между структурами и классами существует, на первый взгляд, некоторая избыточность, то об объединениях этого сказать нельзя. В объектно-ориентированном языке важна поддержка инкапсуляции. Поэтому способность объединений связывать воедино программу и данные позволяет создавать такие типы классов, в которых все данные находятся в общей области памяти. Это именно то, чего нельзя сделать с помощью классов. Применительно к C++ имеется несколько ограничений, накладываемых на использование объединений. Во-первых, они не могут наследовать какой бы то ни было класс и не могут быть базовым классом для любого другого класса. Объединения не могут иметь статических членов. Они также не должны содержать объектов с конструктором или деструктором, хотя сами по себе объединения могут иметь конструкторы и деструкторы.

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

Например, рассмотрим следующий фрагмент:

```
union { // анонимное объединение
    int i;
    char ch [4];
};
    // непосредственный доступ к переменным i и ch
    i = 10;
```

```
ch[0] = 'X';
```

Поскольку переменные i и ch не являются частью какого бы то ни было объекта, доступ к ним осуществляется непосредственно. Тем не менее они находятся в одной и той же области памяти.

8. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Обработка исключительных ситуаций C++ обеспечивает встроенный механизм обработки ошибок, называемый обработкой исключительных ситуаций (exception handling). Используя их, можно упростить управление и реакцию на ошибки во время выполнения программ. Обработка исключительных ситуаций организуется с помощью трех ключевых слов: try, catch и throw. Инструкции программы, во время выполнения которых вы хотите обеспечить обработку исключительных ситуаций, располагаются в блоке try. Если исключительная ситуация (т. е. ошибка) имеет место внутри блока try, она возбуждается (ключевое слово throw), перехватывается (ключевое слово catch) и обрабатывается.

Любая инструкция, которая возбуждает исключительную ситуацию, должна выполняться внутри блока try. Функции, которые вызываются из блока try также могут возбуждать исключительную ситуацию. Любая исключительная ситуация должна перехватываться инструкцией catch, которая располагается непосредственно за блоком try, возбуждающем исключительную ситуацию.

Основная форма инструкций try и catch:

```
try {
   // блох возбуждения исключительной ситуации
}
catch (type1 arg) {
   // блок перехвата исключительной ситуации
}
catch (type2 arg) {
   // блок перехвата исключительной ситуации
}
catch (type3 arg) {
   // блок перехвата исключительной ситуации
. . . . . .
catch (typeN arg) {
   // блок перехвата исключительной ситуации
}
```

Блок try должен содержать ту часть программы, в который отслеживаются ошибки. После того как исключительная ситуация возбуждена, она перехва-

тывается соответствующей этой конкретной исключительной ситуации инструкцией catch, которая ее обрабатывает. С блоком try может быть связано более одной инструкции catch. То, какая именно инструкция catch используется, зависит от типа исключительной ситуации. То есть, если тип данных, указанный в инструкции catch, соответствует типу исключительной ситуации, выполняется данная инструкция catch. При этом все оставшиеся инструкции блока try игнорируются. Если исключительная ситуация перехвачена, аргумент arg получает ее значение. Если не нужен доступ к самой исключительной ситуации, можно в инструкции catch указать только ее тип type, аргумент arg указывать не обязательно. Можно перехватывать любые типы данных, включая и типы создаваемых классов. Фактически в качестве исключительных ситуаций часто используются именно типы классов.

Основная форма инструкции throw:

throw «исключительная_ситуация»;

Инструкция throw должна выполняться либо внутри блока try, либо в любой функции, которую этот блок вызывает (прямо или косвенно).

«исключительная_ситуация» — это возбуждаемая инструкцией throw исключительная ситуация. Если возбуждается исключительная ситуация, для которой нет соответствующей инструкции catch, может произойти ненормальное завершение программы. Возбуждение необрабатываемой исключительной ситуации приводит к вызову стандартной библиотечной функции terminate(). По умолчанию для завершения программы функция terminate() вызывает функцию abort(), однако при желании можно задать собственную процедуру завершения программы. За подробностями обращайтесь к справочной документации вашего компилятора.

1. В следующем очень простом примере показано, как в C++ функционирует система обработки исключительных ситуаций:

```
// Простой пример обработки исключительной ситуации
#include <iostream>
using namespace std;

int main(}
```

```
{
    cout << "начало\n";
    try { // начало блока try
        cout << "Внутри блока try\n";
        throw 10; // возбуждение ошибки
        cout << "Эта инструкция выполнена не будет";
    }
    catch (int i) { // перехват ошибки ,
        cout << "перехвачена ошибка номер: ";
        cout << i << "\n";
    }
    cout << "конец";
    return 0;
}
```

После выполнения программы на экран будет выведено следующее:

```
начало
Внутри блока try
перехвачена ошибка номер: 10
конец
```

Блок try, содержащий три инструкции, и инструкция catch (int i), обрабатывающая исключительную ситуацию целого типа. Внутри блока try будут выполнены только две из трех инструкций — инструкции cout и throw. После того как исключительная ситуация возбуждена, управление передается выражению catch и выполнение блока try завершается. Таким образом, инструкция catch вызывается не явно, управление выполнением программы просто передается этой инструкции. Следовательно, следующая за инструкцией throw инструкция cout не будет выполнена никогда. После выполнения инструкции catch, управление программы передается следующей за ней инструкции.

Как уже упоминалось, тип исключительной ситуации должен соответствовать типу, заданному в инструкции catch. Например, в предыдущем примере, если изменить тип данных в инструкции catch на double, то исключительная ситуация не будет перехвачена и будет иметь место ненормальное завершение программы. Это продемонстрировано в следующем фрагменте:

```
// Этот пример работать не будет
```

```
#include <iostream>
using namespace std;
int main()
{
    cout << "начало\n";
    try { // начало блока try
        cout << "Внутри блока try\n";
        throw 10; // возбуждение ошибки
        cout << "Эта инструкция выполнена не будет";
    }
    catch (double i) {  // Эта инструкция не будет работать
                        // с исключительной ситуацией целого типа
        cout << "перехвачена ошибка номер: ";
        cout << i << "\n";
    }
    cout << "конец";
    return 0;
}
```

Поскольку исключительная ситуация целого типа не будет перехвачена инструкцией catch типа double, на экран программа выведет следующее:

```
начало
Внутри блока try
Abnormal program termination
```

9. СТАНДАРТНАЯ БИБЛИОТЕКА STL

9.1. Знакомство с библиотекой стандартных шаблонов

Ядро библиотеки стандартных шаблонов образуют три основополагающих элемента: контейнеры, алгоритмы и итераторы. Эти элементы функционируют в тесной взаимосвязи друг с другом, обеспечивая искомые решения проблем программирования.

Контейнеры (containers) — это объекты, предназначенные для хранения других объектов. Контейнеры бывают различных типов. В каждом классе-контейнере определяется набор функций для работы с этим контейнером. Например, список содержит функции для вставки, удаления и слияния (merge) элементов.

Алгоритмы (algorithms) выполняют операции над содержимым контейнеров. Существуют алгоритмы для инициализации, сортировки, поиска или замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) — это объекты, которые по отношению к контейнерам играют роль указателей. Они позволяют получать доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива. Имеется пять типов итераторов, которые описаны ниже:

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

По аналогии с потоковым вводом/выводом под вводом понимается ввод информации из контейнера, т. е. считывание, а под выводом — вывод информации в контейнер, т. е. запись.

С итераторами можно работать точно так же, как с указателями. Над ними можно выполнять операции инкремента и декремента. К ним можно применить оператор *. Типом итераторов объявляется тип iterator, который определен в различных контейнерах.

В библиотеке стандартных шаблонов также поддерживаются обратные итераторы (reverse iterators).

Различные типы итераторов

Термин	Тип итератора
Randlter	Произвольного доступа (random access)
Bilter	Двунаправленный (bidirectional)
Forlter	Однонаправленный (forward)
Inlter	Ввода (input)
Outlter	Вывода (output)

Вдобавок к контейнерам, алгоритмам и итераторам, в библиотеке стандартных шаблонов поддерживается еще несколько стандартных компонентов. Главными среди них являются распределители памяти, предикаты и функции сравнения.

У каждого контейнера имеется определенный для него распределитель памяти (allocator), который управляет процессом выделения памяти для контейнера.

9.2. Классы-контейнеры

Контейнерами называются объекты библиотеки стандартных шаблонов, непосредственно предназначенные для хранения данных. В таблице 8.2.1 перечислены контейнеры, определенные в библиотеке стандартных шаблонов, а также заголовки, которые следует включить в программу, чтобы использовать тот или иной контейнер.

Таблица 8.2.1. Контейнеры, определенные в библиотеке стандартных шаблонов

Контейнер	Описание	Заголовок
bitset	Множество битов	<bitset>
deque	Двусторонняя очередь	<deque>
list	Линейный список	<list>
map	Ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано только одно значение	<map>
multimap	Ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано два или более значений	<map>
multiset	Множество, в котором каждый элемент не обязательно уникален	<set>
priority_queue	Очередь с приоритетом	<queue>
queue	Очередь	<queue>
set	Множество, в котором каждый элемент уникален	<set>
stack	Стек	<stack>
vector	Динамический массив	<vector>

Поскольку имена типов элементов, входящих в объявление класса-шаблона, могут быть самыми разными, в классах-контейнерах с помощью ключевого слова typedef объявляются некоторые согласованные версии этих типов. Эта операция позволяет конкретизировать имена типов

Согласованное имя типа	Описание
size_type	Интегральный тип, эквивалентный типу size_t
reference	Ссылка на элемент
const_reference	Постоянная ссылка на элемент
iterator	Итератор
const_iterator	Постоянный итератор
reverse_iterator	Обратный итератор
const_reverse_iterator	Постоянный обратный итератор
value_type	Тип хранящегося в контейнере значения
allocator_type	Тип распределителя памяти
key_type	Тип ключа

key_compare	Тип функции, которая сравнивает два ключа
value_compare	Тип функции, которая сравнивает два значения

9.3. Алгоритмы

Алгоритмы предназначены для разнообразной обработки контейнеров. Хотя в каждом контейнере поддерживается собственный базовый набор операций, стандартные алгоритмы обеспечивают более широкие и комплексные действия. Кроме этого, они позволяют одновременно работать с двумя контейнерами разных типов. Для доступа к алгоритмам библиотеки стандартных шаблонов в программу необходимо включить заголовок <algorithm>. В библиотеке стандартных шаблонов определяется большое число алгоритмов, которые систематизированы в табл. 8.3.1. Все алгоритмы представляют собой функции-шаблоны. Это означает, что их можно использовать с контейнерами любых типов.

Таблица 8.3.1. Алгоритмы библиотеки стандартных шаблонов

Алгоритм	Назначение
adjacent_find	Выполняет поиск смежных парных элементов в последовательности. Возвращает итератор первой пары
binary_search	Выполняет бинарный поиск в упорядоченной последовательности
copy	Копирует последовательность
copy_backward	Аналогична функции copy() за исключением того, что перемещает в начало последовательности элементы из ее конца
count	Возвращает число элементов в последовательности
count_jf	Возвращает число элементов в последовательности, удовлетворяющих некоторому предикату
equal	Определяет идентичность двух диапазонов
equal_range	Возвращает диапазон, в который можно вставить элемент, не нарушив при этом порядок следования элементов в последовательности
fill fill_n	Заполняет диапазон заданным значением

find	Выполняет поиск диапазона для значения и возвращает первый найденный элемент
find_end	Выполняет поиск диапазона для подпоследовательности. Функция возвращает итератор конца подпоследовательности внутри диапазона
find_first_of	Находит первый элемент внутри последовательности, парный элементу внутри диапазона
find_jf	Выполняет поиск диапазона для элемента, для которого определенный пользователем унарный предикат возвращает истину
for_each	Назначает функцию диапазону элементов
generate generate_n	Присваивает элементам в диапазоне значения, возвращаемые порождающей функцией
includes	Определяет, включает ли одна последовательность все элементы другой последовательности
inplace_merge	Выполняет слияние одного диапазона с другим. Оба диапазона должны быть отсортированы в порядке возрастания элементов. Результирующая последовательность сортируется
iter_swap	Меняет местами значения, на которые указывают два итератора, являющиеся аргументами функции
lexicographical_compare	Сравнивает две последовательности в алфавитном порядке
lower_bound	Обнаруживает первое значение в последовательности, которое не меньше заданного значения
make_heap	Выполняет пирамидальную сортировку последовательности (пирамида, на английском языке heap, — полное двоичное дерево, обладающее тем свойством, что значение каждого узла не меньше значения любого из его дочерних узлов)
max	Возвращает максимальное из двух значений
max_element	Возвращает итератор максимального элемента внутри диапазона

merge	Выполняет слияние двух упорядоченных последовательностей, а результат размещает в третьей последовательности
min	Возвращает минимальное из двух значений
min_element	Возвращает итератор минимального элемента внутри диапазона
mismatch	Обнаруживает первое несовпадение между элементами в двух последовательностях. Возвращает итераторы обоих несовпадающих элементов
next_permutation	Образует следующую перестановку (permutation) последовательности
nth_element	Упорядочивает последовательность таким образом, чтобы все элементы, меньшие заданного элемента E, располагались перед ним, а все элементы, большие заданного элемента E, — после него
partial_sort	Сортирует диапазон
partial_sort_copy	Сортирует диапазон, а затем копирует столько элементов, сколько войдет в результирующую последовательность
partition	Упорядочивает последовательность таким образом, чтобы все элементы, для которых предикат возвращает истину, располагались перед элементами, для которых предикат возвращает ложь
pop_heap	Меняет местами первый и предыдущий перед последним элементы, а затем восстанавливает пирамиду
prev_permutation	Образует предыдущую перестановку последовательности
push.heap	Размещает элемент на конце пирамиды
random_shuffle	Беспорядочно перемешивает последовательность
remove remove_if remove_copy remove_copy_if	Удаляет элементы из заданного диапазона

replace replace_if replace_copy replace_copy_if	Заменяет элементы внутри диапазона
reverse reverse_copy	Меняет порядок сортировки элементов диапазона на обратный
rotate rotate.copy	Выполняет циклический сдвиг влево элементов в диапазоне
search	Выполняет поиск подпоследовательности внутри последовательности
search.n	Выполняет поиск последовательности заданного числа одинаковых элементов
set_difference	Создает последовательность, которая содержит различающиеся участки двух упорядоченных наборов
setjntersection	Создает последовательность, которая содержит одинаковые участки двух упорядоченных наборов
set_symmetric_difference	Создает последовательность, которая содержит симметричные различающиеся участки двух упорядоченных наборов
set_union	Создает последовательность, которая содержит объединение (union) двух упорядоченных наборов
sort	Сортирует диапазон
sort_heap	Сортирует пирамиду внутри диапазона
stable_partition	Упорядочивает последовательность таким образом, чтобы все элементы, для которых предикат возвращает истину, располагались перед элементами, для которых предикат возвращает ложь. Разбиение на разделы остается постоянным; относительный порядок расположения элементов последовательности не меняется
stable_sort	Сортирует диапазон. Одинаковые элементы не переставляются
swap	Меняет местами два значения
swap_ranges	Меняет местами элементы в диапазоне

transform	Назначает функцию диапазону элементов и сохраняет результат в новой последовательности
unique unique.copy	Удаляет повторяющиеся элементы из диапазона
upper_bound	Обнаруживает последнее значение в последовательности, которое не больше некоторого значения

10. ДИНАМИЧЕСКАЯ ИДЕНТИФИКАЦИЯ И ПРИВЕДЕНИЕ ТИПОВ

10.1. Понятие о динамической идентификации типа

Динамическая идентификация типа не характерна для языков программирования, в которых не поддерживается полиморфизм. В языках, в которых не поддерживается полиморфизм, информация о типе объекта известен уже на этапе компиляции программы. В языках, поддерживающих полиморфизм, возможны ситуации, в которых тип объекта на этапе компиляции неизвестен, поскольку до выполнения программы не определена точная природа объекта. В C++ полиморфизм реализуется через иерархии классов, виртуальные функции и указатели базовых классов. При таком подходе указатель базового класса может использоваться либо для указания на объект базового класса, либо для указания на объект любого класса, производного от этого базового. Следовательно, не всегда есть возможность заранее узнать тип объекта, на который будет указывать указатель базового класса в каждый данный момент времени. В таких случаях определение типа объекта должно происходить во время выполнения программы, а для этого служит механизм динамической идентификации типа. Информацию о типе объекта получают с помощью оператора typeid. Для использования оператора typeid в программу следует включить заголовок <typeinfo>. Ниже представлена основная форма оператора typeid:

```
typeid  (  «объект» )
```

«объект» — этот тот объект, информацию о типе которого необходимо получить. Оператор typeid возвращает ссылку на объект типа type_info, который и описывает тип объекта объект. В классе type_info определены следующие открытые члены:

```
bool operator== ( const type_info & объект );
bool operator!= ( const type_info & объект );
bool before     ( const type_info & объект );
const char * name ( );
```

Сравнение типов обеспечивают перегруженные операторы == и !=. Функция before ( ) возвращает истину, если вызывающий объект в порядке сор-

тировки расположен раньше объекта, заданного в качестве параметра. Функция name ( ) возвращает указатель на имя типа.

Вторая форма оператора typeid, в которой в качестве аргумента указывают имя типа:

```
typeid ( «имя_типа» )
```

Обычно с помощью данной формы оператора typeid получают объект типа type_info, который можно использовать в инструкции сравнения типов. Поскольку оператор typeid чаще всего применяют к разыменованному указателю (т. е. указателю, к которому уже был применен оператор *), для обработки положения, когда разыменованный указатель равен нулю, была придумана специальная исключительная ситуация bad_typeid, которую в этом случае возбуждает оператор typeid. Динамическая идентификация типа используется далеко не в каждой программе.

10.2. Оператор dynamic jcast

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

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

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

Например, пусть дано два полиморфных класса B и D, причем класс D является производным от класса B, тогда оператор dynamic_cast всегда может привести тип указателя D* к типу указателя B*. Это возможно потому, что ука-

затель базового класса всегда может указывать на объект производного класса. Оператор dynamic_cast может также привести тип указателя B* к типу указателя D*, но только в том случае, если объект, на который указывает указатель, действительно является объектом типа D.

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

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

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

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

```
Приведение типов прошло успешно.
```

10.3. Операторы const_cast, reinterpret_cast и static_cast

Доступны еще три операторы приведения типов, их основные формы:

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

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

типов, более характерный для языка C.

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

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

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

1. В следующей программе демонстрируется использование оператора reinterpret_cast.

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

int main ( )
{
    int i;
    char * p = "Это строка";
    // приведение типа указателя к типу целого
    i = reinterpret_cast < int > ( p );
    cout << i;
    return 0;
}
```

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

2. В следующей программе демонстрируется оператор const_cast.

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

void f ( const int * p )
{
    int * v;
    // преобразование типа лишает указатель p атрибута const
    v = const_cast < int * > ( p );
    * v = 100; // теперь указатель v может изменить объект
}

int main ( )
{
    int x = 99;
    cout << "Объект x перед вызовом функции равен: " << x << endl;
    f ( & x );
    cout << "Объект x после вызова функции равен: " << x << endl;
    return 0;
}
```

Ниже представлен результат выполнения программы:

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

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

11. РАСПРЕДЕЛЕНИЕ ЧАСТЕЙ ОПИСАНИЯ КЛАССА ПО ФАЙЛАМ

Описание заголовочной части и части реализации класса можно разместить в одном фале или разнести в двух файлах.

Рекомендуется разместить описание класса в двух файлах. Для размещения заголовочной части описания класса используется файл наименованием класса и расширением «.h». Для размещения части реализации описания класса используется файл наименованием класса и расширением «.cpp». В начало файла с расширением «.cpp» надо добавить инструкцию:

#include “имя_класса.h”

Описание класса можно разместить в одном файле. Заголовочная часть должна располагаться перед описанием части реализации. Рекомендуется в качестве наименования файла использовать имя класса, а расширение «.h».

11.1. Конструктивное построение программы

Обращение к объекту для выполнения (активации) определенной реакции инициируется по:

1. Внешнему событию.
2. Внутреннему событию.
3. Сигналу (посылает вне).
4. Запросу (посылает вне и ожидает ответа).
5. Непосредственной активации (вызов метода объекта).

Элементы языка C++ для построения программ согласно парадигме ООП.

Описание объекта	Описание класса
Создание объекта	Создание объекта на бае определенного класса посредством конструктора класса

Построение конструкции объекта	Наследственность. Задействование других классов в описании.
Взаимодействие объектов в рамках конструкции	Непосредственный вызов метода объекта. Сигнал от объекта ⇒ обработчик объекта.
Архитектура приложения-системы (конструктивное построение)	Построение схемы взаимосвязи объектов в рамках приложения
Схема взаимодействия объектов в составе приложения-системы	Построение схемы интерфейсов между объектами приложения. Сигнал от объекта ⇒ обработчик объекта.
Схема взаимодействия объектов приложения внешней средой	Построение схемы интерфейсов между объектами приложения и внешней средой. Внешнее (системное) событие ⇒ обработчик объекта.

12. СПИСОК ЛИТЕРАТУРЫ

1. Иванова Г.С. , Ничушкина Т.Н. Объектно-ориентированное программирование. Учебник для студентов ВУЗов. Издательство: МГТУ им. Н.Э. Баумана. Москва, 2014г. 455 стр.
2. Герберт Шилдт. C++ базовый курс. Издательский дом «Вильямс». Москва, 2017 г. 620 стр.
3. Васильев А.Н. Объектно-ориентированное программирование на C++. Издательство: Наука и Техника. Санкт-Петербург, 2016г. 543 стр.
4. Васильев А.Н. Программирование на C++ в примерах и задачах. Издательство: Эксмо. Москва, 2017г. 368 стр.
5. Г.И. Радченко, Е.А. Захаров. Объектно-ориентированное программирование. Конспект лекций. Челябинск, Издательский центр ЮУрГУ, 2013 г. 167 стр.
6. Ашарина И.В. Объектно-ориентированное программирование на C++: лекции и упражнения. Учебное пособие для ВУЗов. Издательство: Горячая линия - Телеком. Москва, 2014г. 335 стр.
7. Ашарина И.В. Язык C++ и объектно-ориентированное программирование в C++. Лабораторный практикум. Учебное пособие для ВУЗов. Издательство: Горячая линия - Телеком. Москва, 2015г. 231 стр.
8. Стивен Прата. Язык программирования C++. Лекции и упражнения. Издательский дом «Вильямс». Москва, 2017 г. 1244 стр.
9. Бьерн Страуструп. Язык программирования C++. специальное издание. Издательство «Бином-Пресс». Москва, 2007 г. 1104 стр.