МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

«БРАТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра управления в технических системах

УТВЕРЖДАЮ:
Проректор по учебной работе
___________ Е.И. Луковникова
«_____» ____________ 201   г.

РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Б1.В.18

НАПРАВЛЕНИЕ ПОДГОТОВКИ

27.03.04 Управление в технических системах

ПРОФИЛЬ ПОДГОТОВКИ

Управление и информатика в технических системах

Программа академического бакалавриата

Квалификация выпускника: бакалавр

СОДЕРЖАНИЕ ПРОГРАММЫ **Стр.**

1. ПЕРЕЧЕНЬ ПЛАНИРУЕМЫХ РЕЗУЛЬТАТОВ ОБУЧЕНИЯ ПО ДИСЦИПЛИНЕ, СООТНЕСЕННЫХ С ПЛАНИРУЕМЫМИ РЕЗУЛЬТАТАМИ ОСВОЕНИЯ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ …..…… 3

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ ………………………………………………………….……….. 3

3. РАСПРЕДЕЛЕНИЕ ОБЪЕМА ДИСЦИПЛИНЫ ……………………………….. 4

3.1 Распределение объёма дисциплины по формам обучения……………...……… 4

3.2 Распределение объёма дисциплины по видам учебных занятий и трудоемкости ………………………………………………………………..……… 4

4. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ …………………………………..………… 5

4.1 Распределение разделов дисциплины по видам учебных занятий ……………… 5

4.2 Содержание дисциплины, структурированное по разделам и темам …..…………5

4.3 Лабораторные работы………………………………………………………………67

4.4 Семинары / практические занятия…..……………………………….……… 68

4.5. Контрольные мероприятия: курсовой проект (курсовая работа), контрольная работа, РГР, реферат……………………………………………………….……… 68

5. МАТРИЦА СООТНЕСЕНИЯ РАЗДЕЛОВ УЧЕБНОЙ ДИСЦИПЛИНЫ К ФОРМИРУЕМЫМ В НИХ КОМПЕТЕНЦИЯМ И ОЦЕНКЕ РЕЗУЛЬТАТОВ ОСВОЕНИЯ ДИСЦИПЛИНЫ …………………………………… 69

6. ПЕРЕЧЕНЬ УЧЕБНО-МЕТОДИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ70

7. ПЕРЕЧЕНЬ ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ, НЕОБХОДИМОЙ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ…………………………… 70

8. ПЕРЕЧЕНЬ РЕСУРСОВ ИНФОРМАЦИОННО – ТЕЛЕКОММУНИКАЦИОННОЙ СЕТИ «ИНТЕРНЕТ» НЕОБХОДИМЫХ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ ……………………………………………. 70

9. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ…………………………………………………………………… 71

9.1. Методические указания для обучающихся по выполнению лабораторных работ / семинаров / практических работ ……………………………………….……… 71

9.2. Методические указания по выполнению контрольной работы…………………… 76

10. ПЕРЕЧЕНЬ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, ИСПОЛЬЗУЕМЫХ ПРИ ОСУЩЕСТВЛЕНИИ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ ……………………………………………………………..…… 77

11. ОПИСАНИЕ МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЙ БАЗЫ, НЕОБХОДИМОЙ ДЛЯ ОСУЩЕСТВЛЕНИЯ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ …………………………………………………………...…… 78

Приложение 1. Фонд оценочных средств для проведения промежуточной аттестации обучающихся по дисциплине……………………………79

Приложение 2. Аннотация рабочей программы дисциплины ……………………. 83

Приложение 3. Протокол о дополнениях и изменениях в рабочей программе ………… 84

1. ПЕРЕЧЕНЬ ПЛАНИРУЕМЫХ РЕЗУЛЬТАТОВ ОБУЧЕНИЯ ПО ДИСЦИПЛИНЕ, СООТНЕСЕННЫХ С ПЛАНИРУЕМЫМИ РЕЗУЛЬТАТАМИ ОСВОЕНИЯ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Вид деятельности выпускника

Дисциплина охватывает круг вопросов, относящихся к проектно-конструкторскому виду профессиональной деятельности выпускника в соответствии с компетенцией и видами деятельности, указанными в учебном плане.

Цель дисциплины

Формирование у обучающихся знаний и навыков по использованию современных технологий и методов разработки программных систем для решения практических задач с использованием современных инструментальных средств, необходимых в дальнейшем, при проектировании и эксплуатации систем управления и автоматизации.

Задачи дисциплины

Освоение принципов и методов объектно-ориентированного программирования с использованием интегрированной среды разработки (ИСР) C++ Builder.

Код компетенции	Содержание компетенций	Перечень планируемых результатов обучения по дисциплине
1	2	3
ОПК-9	способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности	**знать:** - требования информационной безопасности к программным системам; **уметь:** - использовать ИСР при проектировании и эксплуатации систем управления и автоматизации; **владеть:** - методами информационных технологий.
ПК-2	способность проводить вычислительные эксперименты с использованием стандартных программных средств с целью получения математических моделей процессов и объектов автоматизации и управления	**знать:** - основные принципы и методологию разработки прикладного программного обеспечения; **уметь:** - проводить вычислительные эксперименты с использованием стандартных программных средств; **владеть:** - методами получения математических моделей процессов и объектов автоматизации и управления.

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Дисциплина Б1.В.18 «Технологии программирования» относится к вариативной части.

Дисциплина «Технологии программирования» базируется на знаниях, полученных при изучении таких учебных дисциплин, как Б1.Б.14 Программирование и основы алгоритмизации, Б1.В.7 Информатика, Б1.В.15 Структуры и алгоритмы обработки данных.

Дисциплина «Технологии программирования» представляет основу для изучения дисциплин: Б1.В.17 Системное программное обеспечение, Б1.В.ДВ.04.01 Прикладное программирование.

Такое системное междисциплинарное изучение направлено на достижение требуемого ФГОС уровня подготовки по квалификации бакалавр.

3. РАСПРЕДЕЛЕНИЕ ОБЪЕМА ДИСЦИПЛИНЫ

3.1. Распределение объема дисциплины по формам обучения

Форма обучения	*Курс*	*Семестр*	*Трудоемкость дисциплины в часах*						*Курсовая работа (проект), контрольная работа, реферат, РГР*	*Вид промежуточной аттестации*
			Всего часов	*Аудиторных часов*	*Лекции*	*Лабораторные работы*	*Практические занятия*	*Самостоятельная работа*		
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Очная	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
Заочная	3	-	108	15	5	10	-	89	+	зачёт
Заочная (ускоренное обучение)	-	-	-	-	-	-	-	-	-	-
Очно-заочная	-	-	-	-	-	-	-	-	-	-

3.2. Распределение объема дисциплины по видам учебных занятий и трудоемкости

Вид учебных занятий	*Трудоемкость (час.)*	*в т.ч. в интерактивной, активной, инновациионной формах, (час.)*	*Распределение по семестрам, час*
			5
1	**2**	**3**	**4**
I. Контактная работа обучающихся с преподавателем (всего)	15	5	15
Лекции (Лк)	5	2	5
Лабораторные работы (ЛР)	10	3	10
Контрольная работа	+	-	+
Групповые (индивидуальные) консультации	+	-	+
II.Самостоятельная работа обучающихся (СР)	89	-	89
Подготовка к лабораторным работам	50	-	50
Подготовка к зачёту	4	-	4
Выполнение контрольной работы	35		35
III. Промежуточная аттестация зачёт	+	-	+
Общая трудоемкость дисциплины ….... час.	108	-	108
зач. ед.	3	-	3

4. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

4.1. Распределение разделов дисциплины по видам учебных занятий

- для заочной формы обучения:

№ раздела	*Наименование раздела*	*Трудоемкость, (час.)*	*Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.)*		
			учебные занятия		*самостоятельная работа обучающихся*
			лекции	*лабораторные работы*	
1	2	3	4	6	7
1.	**Базовые средства языка С++**	**19**	**1**	**2**	**16**
2.	**Функции и управление памятью**	**17,5**	**0,5**	**1**	**16**
3.	**Введение в технологии программирования**	**16,5**	**0,5**	**1**	**15**
4.	**Классы в С++**	**18**	**1**	**2**	**15**
5.	**Наследование**	**19**	**1**	**2**	**16**
6.	**Шаблоны и обработка исключительных ситуаций**	**18**	**1**	**2**	**15**
	ИТОГО	**108**	**5**	**10**	**93**

4.2. Содержание дисциплины, структурированное по разделам и темам

Раздел 1:
БАЗОВЫЕ СРЕДСТВА ЯЗЫКА С++

1.1. Алфавит и лексемы языка

Алфавит С++ включает:
- прописные и строчные латинские буквы и знак подчеркивания;
- арабские цифры от 0 до 9;
- специальные (символы) знаки:
" { } | , [ ] () + - / % * . \ ´ : ? < = > ! & # ~ ; ^;
- пробельные символы: пробел, символ табуляции, символ перехода на новую строку.

Из символов алфавита формируются лексемы языка. Лексема или элементарная конструкция - минимальная единица языка, имеющая самостоятельный смысл.

К лексемам относятся:
- идентификаторы;
- ключевые (зарезервированные) слова;
- константы;
- разделители (скобки, точка, запятая, пробельные символы).

1.1.1. Идентификаторы

Идентификатор – это имя программного объекта (переменной, функции, типа и т.п.). В идентификаторе могут использоваться латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, sysop, SySop, SYSOP – три различных имени. Первым символом идентификатора может быть буква или знак подчеркивания. Длина идентификатора по стандарту не ограничена, но некоторые компиляторы и компановщики налагают на неё ограничения. Идентификатор не должен совпадать с ключевыми словами.

1.1.2. Ключевые слова

Ключевые слова – зарезервированные идентификаторы, которые имеют специальные назначения для компилятора. Примеры ключевых слов: *auto*, *class*, *const*, *double*, *if*, *mutable*, *operator*, *return*, *throw*, *union*, *virtual*, *void*, *while*.

1.1.3. Знаки операций

Знак операции – это один или более символов, определяющих действия над операндами. Операции делятся по количеству участвующих в них операндов на унарные, бинарные и тернарные. Один и тот же знак может интерпретироваться по-разному в зависимости от контекста. В табл. 1.1 приведен список основных операций в соответствии с их приоритетами (по убыванию приоритетов, операции с разными приоритетами разделены чертой).

Таблица 1.1

Основные операции языка С++

Операция	Краткое описание
1	2
Унарные операции	
::	разрешение области видимости
++	увеличение на 1 (инкремент)
- -	уменьшение на 1 (декремент)
sizeof	размер
~	поразрядное отрицание
!	логическое отрицание
-	унарный минус
+	унарный плюс
&	взятие адреса
*	разадресация
new	выделение памяти
delete	освобождение памяти
(type)	преобразование типа
Бинарные и тернарные операции	
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
==	равно
!=	не равно
&	поразрядная конъюнкция (И)
^	поразрядное исключающее ИЛИ
|	поразрядная дизъюнкция (ИЛИ)
&&	логическое И
||	логическое ИЛИ
?:	условная операция (тернарная)
=	присваивание

*=	умножение с присваиванием
/=	деление с присваиванием
%=	остаток от деления с присваиванием
+=	сложение с присваиванием
-=	вычитание с присваиванием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное И с присваиванием
|=	поразрядное ИЛИ с присваиванием
^	поразрядное исключающее ИЛИ с присваиванием

Рассмотрим некоторые операции подробнее.

Операции уменьшения и увеличения на 1. Эти операции имеют две формы записи – префиксную, когда операция записывается перед операндом, и постфиксную. В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

Рассмотрим пример на использование префиксной и постфиксной форм записи.

Пример.

```
#include <stdio.h>
void main()
{
int x=3, y=3;
printf ("Значение префиксного выражения:%d\n", ++x);
printf ("Значение постфиксного выражения:%d\n",y++);
printf ("Значение x после приращения:%d\n", x);
printf ("Значение y после приращения:%d\n", y);
}
```

Результаты работы программы:

Значение префиксного выражения: 4
Значение постфиксного выражения: 3
Значение x после приращения: 4
Значение y после приращения: 4

Замечания о вводе/выводе. В языке С++ нет встроенных средств ввода/вывода. Ввод/вывод осуществляется с помощью функций, типов и объектов, содержащихся в стандартных библиотеках. Используется два способа: функции, унаследованные из языка С и объекты С++.

Основные функции ввода/вывода в стиле С:

```
int scanf (const char*format,...)        //ввод
int printf (const char*format,...)       //вывод
```

Они выполняют форматированный ввод и вывод произвольного количества величин в соответствии со строкой формата format.

Рассмотрим два примера.

Пример 1.

Программа, использующая функции ввода/вывода в стиле С.

```
#include <stdio.h>
void main()
{
int i;
// Выводит приглашение и переходит на новую строку в
// соответствие с \n
printf ("Введите целое число\n");
// Заносит введенное с клавиатуры целое число в
```

```
// переменную i (& означает операцию получения адреса)
scanf ("%d",&i);
// Выводит сообщение, заменив спецификацию значением
// этого числа
printf ("Вы ввели число%d",i);
}
```

Пример 2.

Та же программа, но с использованием библиотеки классов С++:

```
// Файл iosream.h содержит описание набора классов
// для управления вводом/вывода
#include <iostream.h>
void main()
{
int i;
// Объект-поток cout для вывода на экран
cout<<"Введите целое число\n";
// Объект-поток cin для ввода с клавиатуры
cin>>i;
cout<<"Вы ввели число”<<i ;
}
```

Операндом операции инкремента в общем случае является так называемые L-значения (L-value). Так обозначается любое выражение, адресующее некоторый участок памяти, в который можно занести значение.

Операции отрицания (-, !, ~). Арифметическое отрицание (-) изменяет знак операнда целого или вещественного типа на противоположный. Логическое отрицание (!) дает в результате значение 0, если операнд есть истина (не нуль), и значения 1, если операнд равен 0. Поразрядное отрицание (~) называется побитовым, инвертирует каждый разряд в двоичном представлении целочисленного операнда.

Деление (/) и остаток от деления (%). Операция деления применима к операндам арифметического типа. Если оба операнда целочисленные, результат операции округляется до целого числа, в противном случае тип результата определяется правилами преобразования. Операция остатка от деления применяется только к целочисленным операндам.

Операции сдвига (<<, >>) применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом. При сдвиге влево освободившиеся разряды обнуляются. При сдвиге вправо освободившиеся биты заполняются нулями, если операнд беззнакового типа, и знаковым разрядом в противном случае.

Операции отношения (<, <=, >, >=, =, !=) сравнивают первый операнд со вторым. Операнды могут быть арифметического типа или указателями. Результатом операции является значение true или false.

Логические операции (&&, ||). Операнды логических операций могут иметь арифметический тип или быть указателями. Результат операции логическое И (&&) имеет значение true, если оба операнда имеют значения true. Результат операции логическое ИЛИ (||) имеет значение true, если хотя бы один из операндов имеет значение true. Логические операции выполняются слева направо.

*Операции присваивания (=, *=, /=, %=,+=, -=, <<=, >>=, &=, |=, ^=)* Могут использоваться в программе как законченные операторы.

Формат операции простого присваивания (=):

операнд_1 = операнд_2, где

операнд_1 – L-значение;

операнд_2 – выражение.

Рассмотрим пример на использование операции присваивания.

Пример.

```
#include <iostream.h>
void main()
{
int a=3, b=5, c=7;
a=b; b=a; c=c+1;
cout<<"a="<<a;
cout<<"\tb="<<b;
cout<<"\tc="<<c;
}
```

Результаты работы программы:

a=5 b=5 c=8

В сложных операциях присваивания (+=, *=, /= и т.п.) при вычислении выражения, стоящего в правой части, используются L-значения из левой части.

Например, a+=b является более компактной записью выражения a=a+b.

Условная операция (?:).

Формат условной операции:

операнд_1? операнд_2: операнд_3

Если результат операнда_1 равен true, то результатом условной операции будет значение второго операнда, иначе – третьего операнда. Условная операция является сокращенной формой условного оператора if.

Рассмотрим пример.

Пример.

```
#include <stdio.h>
void main()
{
int a=11, b=4, max;
max=(b>a)?b:a;
printf ("Наибольшее число:%d", max);
}
```

Результат работы программы:

Наибольшее число: 11

1.1.4. Константы

Константами называют неизменяемые величины. Различают целые, вещественные, символьные и строковые константы. Символьная константа представляет собой один или два символа, заключённых в апострофы. Строковая константа – последовательность символов, заключенная в кавычки.

1.2. Типы данных

Тип данных определяет:

- внутреннее представление данных в памяти компьютера;
- множество значений, которые могут применять величины этого типа;
- операции и функции, которые можно применять к величинам этого типа.

Все типы языка С++ можно разделить на основные и составные. К основным (базовым) типам относятся: целочисленные, символьный, логический, типы с плавающей точкой. На основе этих типов программист может вводить описание составных типов. К ним относятся массивы, перечисления, ссылки, указатели, структуры, объединения и классы.

Рассмотрим базовые типы.

1.2.1. Логический тип

Логический тип данных – *bool*. Переменная типа *bool* может принимать только одно из двух значений – *true* и *false*.

При преобразовании в целочисленные типы значение *true* преобразовывается в 1, а значение *false* – в 0. При преобразовании целых типов в тип *bool* нулевое значение преобразуется в *false*, а все ненулевые значения – в *true*.

1.2.2. Символьные типы

Переменная типа *char* может хранить один из набора символов компьютера. Для переменной такого типа отводится 8 бит.

При преобразовании символьного типа в целочисленный получается числовое значение символа. Числовые значения *char* могут быть интерпретированы как знаковые (от -128 до 127) или беззнаковые (от 0 до 255). Язык позволяет явным образом определить указание диапазона: тип *signed char* означает знаковый символьный тип, *unsigned char* – беззнаковый.

Символьный литерал представляет собой символ, заключенный в апострофы. Литерал с особым значением может состоять из нескольких символов, при этом первым символом всегда идет “\”. Эта последовательность символов называется управляющей или *escape* – последовательностью. В табл. 1.2 приведены их допустимые значения.

Таблица 1.2

Управляющие последовательности

Название	Литерал
Перевод строки	\n
Горизонтальная табуляция	\t
Вертикальная табуляция	\v
Забой	\b
Возврат каретки	\r
Перевод страницы (формат)	\f
Звуковой сигнал	\a
Обратная косая черта	\\
Вопросительный знак	\?
Апостроф	\’
Кавычка	\”
Шестнадцатеричный код символа	\0xddd

1.2.3. Целочисленные типы

Целочисленные типы представлены типом *int*, который может применяться со спецификатором знаковости – *signed* или *unsigned* и со спецификатором размера *short* или *long*. Используя спецификатор, можно не указывать тип *int*. В случае отсутствия спецификатора знаковости целочисленный тип всегда интерпретируется как знаковый.

Таблица 1.3

Целочисленные типы

Тип	Размер, бит	Минимальное значение	Максимальное значение
signed char	8	-128	127
unsigned char	8	0	255
short int	16	-32768	32767
unsigned short int	16	0	65535
long int	32	-2147483648	2147483647
unsigned long int	32	0	4294967295
long long	64	0	18446744073709551615
unsigned long long	64	-9223372036854775808	9223372036854775807

1.2.4. Типы с плавающей точкой

Типы с плавающей точкой представлены тремя видами с разным размером – *float*, *double* и *long double*. Конкретный размер данных типов, диапазон и точность представления чисел зависят от конкретной реализации.

Например, тип *float* может иметь размер 32 бита, *double* – 64 бита, *long double* – 80 бит.

Конкретные размеры различных типов зависят от реализаций. В С++ размеры типов выражаются в размерах *char*. Получить размер конкретного типа или объекта в С++ можно при помощи операции sizeof. По определению, *sizeof(char)=1*.

1=sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)

1<=sizeof(bool)<=sizeof(long)

sizeof(float)<=sizeof(double)<=sizeof(long double)

sizeof(N)=sizeof(signed N)<=sizeof(unsigned N)

Здесь, *N- char, short int, int или long int*.

1.2.5. Тип void

Тип *void* синтаксически является базовым, но он может быть только частью других составных типов. В основном тип *void* применяется для указания на то, что функция не возвращает значения (*void func();*), и в качестве базового для указателей на объекты неизвестного типа (*void *ptr*).

1.3. Структура программы

Программа на языке С++ состоит из функций, описаний и директив препроцессора.

Рассмотрим коротко функции. Одна из функций должна иметь имя *main*. Выполнение программы начинается с первого оператора этой функции. Определение функции имеет следующий формат:

тип _ возвращаемого _ значения имя([параметры])

{

тело функции

}

Функция используется для вычисления какого-либо значения, поэтому перед именем функции указывается его тип. Каждый оператор тела функции заканчивается точкой с запятой. Тело функции является блоком и поэтому заключается в фигурные скобки.

Пример.

Структура программы, содержащей функции main, f1, f2.

```
директивы препроцессора
int main()
{
операторы главной функции
}
int f1()
{
операторы функции f1
}
int f2()
{
операторы функции f2
}
```

1.3.1. Директивы препроцессора

Препроцессором называется первая фаза компилятора. Инструкции препроцессора называют директивами. Они должны начинаться с символа #. Рассмотрим одну из наиболее часто используемых директив *include*.

Директива препроцессора *#include <имя файла>* заставляет заменить её текстом указанного в ней файла:

#include “myfile”

При включении заголовочных файлов стандартной библиотеки вместо кавычек используются угловые скобки, что заставляет искать указанный файл в каталоге стандартной библиотеки, а не в текущем каталоге. Наличие пробелов внутри <> существенно, и препроцессор может не найти включаемый файл, если директива имеет вид *#include < iostream >*, а не *#include <iostream>*.

Заголовочные файлы обычно имеют расширение *.h* и могут содержать:

- определение типов, констант, встроенных функций, шаблонов, перечислений;
- объявления функций, данных, имен, шаблонов;
- пространства имен;
- директивы препроцессора;
- комментарии.

Комментарий либо начинается с двух символов // и заканчивается символом перехода на новую строку, либо заключается между символами /* и */. Рекомендуется использовать для пояснений // - комментарии, а скобки /* */ применять для временного исключения блоков кода при отладке.

При указании заголовочных файлов стандартной библиотеки расширение *.h* можно опускать.

1.4. Переменные и выражения

В любой программе требуется производить вычисления. Для вычисления значений используются выражения, которые состоят из операндов, знаков операций и скобок.

1.4.1. Переменные

Переменная – это именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти,

в которой хранится значение. Перед использованием любая переменная должна быть описана.

Общий вид оператора описания переменных:

[класс памяти] [const] тип имя [инициализатор];

Рассмотрим правила задания составных частей этого оператора.

Класс памяти определяет время жизни и область действия (видимости) переменной. Если класс памяти не указан явным образом, он определяется компилятором, исходя из контекста объявления. Время жизни переменной может быть постоянным (в течение выполнения программы) и временным (в течение выполнения блока). Область действия – это часть программы, в которой идентификатор можно использовать для доступа к связанной с ним областью памяти. В зависимости от области действия переменная может быть локальной или глобальной.

Если переменная определена внутри блока, она называется локальной, область её действия – от точки описания до конца блока, включая все вложенные блоки. Если переменная определена вне любого блока, она называется глобальной и областью её действия считается файл, в котором она определена, от точки описания до её конца.

Рассмотрим пример на определение глобальных и локальных переменных.

Пример.

```
int a;           // Глобальная переменная a
void main()
{
int b;           // Локальная переменная b
extern int x;    // Переменная x определена в другом месте
static int c;    // Локальная статическая переменная c
a=1;             // Присваивание значения глобальной переменной
int a;           // Локальная переменная a
a=2;             // Присваивание значения локальной переменной
}
int x=4;         // Определение и инициализация x
```

Необязательный класс памяти может принимать одно из значений:

auto – автоматическая переменная. Память под неё выделяется в стеке и при необходимости инициализируется каждый раз при выполнении оператора, содержащего её определение. Освобождение памяти происходит при выходе из блока, в котором описана переменная. Для глобальных переменных этот спецификатор не используется, а для локальных он принимается по умолчанию.

extern означает, что переменная определяется в другом месте программы (в другом файле или дальше по тексту). Используется для создания переменных, доступных во всех модулях программы, в которых они объявлены.

static – статическая переменная. Время жизни – постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. В зависимости от расположения оператора описания статические переменные могут быть глобальными и локальными.

register аналогично *auto*, но память по возможности выделяется в регистрах процессора. Если такой возможности у компилятора нет, переменные обрабатываются как *auto*.

const. Модификатор *const* показывает, что значение переменной изменять нельзя. Такую переменную называют *именованной константой* или просто *константой*.

Инициализатор. При описании можно присваивать переменной начальное значение, это называется инициализацией. Инициализатор можно записывать в двух формах – со знаком равенства:

=значение

или в круглых скобках:

(значение)

Константа должна быть инициализирована при объявлении. Если при определении начальное значение переменных явным образом не задается, компилятор присваивает гло-

бальным и статическим переменным нулевое значение соответствующего типа. Автоматические переменные не инициализируются.

В одном операторе можно описывать несколько переменных одного типа, разделяя их запятыми.

Пример.

```
short int a=1;
const char c='c';
char s, sf='f';
char t(54);
float c=0.22, x(3), sum;
```

1.4.2. Выражения

Выражения состоят из операторов, знаков операций и скобок.

Примеры выражений:

```
(a+0.12)/6
x&&y||!z
(t*sin(x)-1.05e4)/((2*k+2)*(2*k+3))
```

В выражение могут входить операнды различных типов. Если операнды имеют одинаковый тип, результат будет иметь тот же тип. Если операнды различного типа, перед вычислениями выполняются преобразования типов по определенным правилам, обеспечивающим преобразования более коротких типов в более длинные для сохранности значимости и точности.

1.5. Программирование алгоритмов различных структур

Программы, разработанные в соответствии с алгоритмами, могут иметь различные структуры. Среди выполняемых последовательностей операторов выделяют типовые структуры, из которых состоят программы: линейная, разветвлённая, циклическая, вложенных циклов. Сложные программы включают в себя все типы структур.

1.5.1. Программирование алгоритмов линейной структуры

Программы линейной структуры не содержат условий, поэтому их операторы выполняются в той последовательности, которая определяется алгоритмом. Для организации программы линейной структуры используются операторы присваивания, ввода данных и вывода результатов обработки.

Оператор присваивания служит для вычисления значения выражения и присваивания его имени результата.

```
a = b+1;          //(b+1) присваивается a
```

Язык C++ допускает несколько присваиваний в одной инструкции.

```
a = b+(c=3);
```

что равнозначно

```
c = 3; a = b+c;
```

C++ предлагает операторы присваивания, комбинирующие присваивание с какой-либо операцией:

```
a+=b;           // равнозначно a=a+b
a*= a+b;        // равнозначно a=a*(a+b)
```

Операторы ввода и вывода данных. В языке C++ нет специальных операторов ввода/вывода. Ввод/вывод обеспечивается стандартной внешней библиотекой. Информация, необходимая программе для использования этой библиотеки, находится в файле *iostream.h*. В этом файле описан набор классов для управления вводом/выводом. Стандартные объекты – потоки *cin* используются для ввода с клавиатуры, а *cout* для вывода на экран. А также применяются операции помещения в поток <<*(вывод на экран)* и извлечение из потока >>*(ввод с клавиатуры)*.

Рассмотрим пример программы линейной структуры.

Пример.

Вычислить расстояние между двумя точками с координатами (x1,y1) и (x2,y2).

```
#include <iostream.h>
#include <math.h>
void main()
{
float x1, y1,x2, y2;
cout<<"Введите координаты первой точки"<<endl;
cin>>x1>>y1;
cout<<Введите координаты второй точки"<<endl;
cin>>x2>>y2;
double r=sqrt(pow(x2-x1,2)+pow(y2-y1,2));
cout<<"Расстояние между двумя точками="<<r;
}
```

Для перехода на следующую строку используется манипулятор *endl*. Он управляет стандартным объектом *cout*.

Язык С++ содержит большое число встроенных математических функций. При работе с ними необходимо подключить заголовочный файл *math.h*, т.е. ввести в начало файла директиву *#include <math.h>*.

Таблица 1.4

Тригонометрические функции

Значение	Функция
Синус	sin(n)
Косинус	cos(n)
Тангенс	tan(n)
Арккосинус	acos(n)
Арктангенс	atan(n)

Величина угла тригонометрической функции должна быть выражена в радианах. Для преобразования величины угла из градусов в радианы используется формула *(n*3.1415256)/180*, где *n* – величина угла в градусах.

Таблица 1.5

Математические функции

Название	Синтаксис
Натуральный логарифм n	log(n)
Экспонента n	exp(n)
Абсолютное значение n	abs(n)
Корень квадратный из n	sqrt (n)
Округление вверх	ceil (n)
Округление вниз	floor (n)
Возведение в степень	pow(аргумент, степень)

1.5.2. Организация программ разветвлённой структуры

Программа данной структуры обязательно содержит условия, в зависимости от которых предусматривают выбор одной из нескольких последовательностей операторов (ветвей).

Для организации разветвлений в программах реализуются операторы перехода, условий и выбора.

Оператор перехода.

goto метка;

Метка – это идентификатор. С помощью оператора перехода управление передается оператору, помеченному *меткой*. Далее выполняются операторы, стоящие за ним.

Например,

```
goto error;
…….
error:cerr<< "Ошибка: Деление на ноль!";
```

cerr – стандартный объект – поток диагностики.

Условный оператор.

```
if (условие)
  оператор;
if (условие)
  оператор1;
else
  оператор2;
```

Условие – это выражение или объявление с инициализацией, с помощью которого выбирается логика выполнения.

Рассмотрим пример программы с оператором *if*.

Пример.

Даны действительные числа x, y, z. Вычислить $\min^2\left(\frac{x+y+z}{2},\ x*y*z\right)+1$.

```
#include <iostream.h>
#include <math.h>
void main()
{
float x, y, z, min;
cout<<"Введите три числа:"<<endl;
cin>>x>>y>>z;
float m1=(x+y+z)/2;
float m2=x*y*z;
if (m1<m2)
min=pow(m1,2)+1;
else
min=pow(m2,2)+1;
cout<<"min="<<min;
}
```

Составная инструкция в С++ – это последовательность инструкций (операторов), заключенных в фигурные скобки. Составные инструкции применяются для группировки простых инструкций в функциональные звенья программы.

Оператор выбора.

Обеспечивает организацию разветвлений путем выбора одного из нескольких операторов. В общем виде оператор выбора выглядит так:

switch (условие)

инструкция,

где *инструкция* – составная инструкция, содержащая метки *case* и необязательную метку *default*. *switch* включает несколько *case*. Условие определяет, какой из *case* будет выполняться, если это вообще произойдет.

Рассмотрим пример на использование оператора выбора *switch*.

Пример.

Программа, имитирующая работу микрокалькулятора.

```
#include <iostream.h>
void main()
{
float x, y ,z;
int operation;
cout<<"Введите операнды:"<<endl;
cin>>x>>y;
cout<<"Введите код операции:"<<endl;
//Код операции: 1-сложение, 2-вычитание, 3-умножение, 4-деление
cin>>operation;
switch (operation)
{
case 1:
z=a+b; break;
case 2:
z=a-b; break;
case 3:
z=a*b; break;
case 4:
z=a/b; break;
default
cout<<"Неправильно введён знак операции"<<endl;
}
cout<<"Результат операции ="<<z;
}
```

Работа оператора *switch* заключается в следующем:

1. Вычисляется выражение в круглых скобках, стоящее за *switch*;
2. Выполняется метка *case*, совпадающая с тем значением, которое было найдено на этапе 1; если ни одна из *case* не соответствует этому значению, выполняется метка *default*; если метки *default* нет, *switch* прерывается;
3. Выполнение *switch* прерывается, когда включается оператор *break* или когда достигается конец *switch*.

1.5.3. Программирование алгоритмов циклической структуры

Программы циклической структуры могут быть организованы с помощью условного оператора. Но в языке С++ имеются специальные операторы цикла *for* и *while*. Оператор *for* служит для организации цикла с известным числом повторений (регулярного), а *while* –для организации цикла с числом повторений, зависящих от результата проверки заданного условия (итерационного).

Оператор for.

```
for (начальная_инструкция; условие; выражение)
операторы тела цикла
следующий_оператор
```

Сначала выполняется *начальная_инструкция*, она инициализирует переменную, используемую в цикле. Затем проверяется условие. Если оно истинно, то выполняются операторы тела цикла, вычисляется выражение, и управление передается обратно в начало цикла *for* с той разницей, что *начальная_инструкция* уже не выполняется. Это продолжается до тех пор, пока условие не станет ложно, после чего управление передается *следующему_оператору*.

Начальная_инструкция может быть оператором присваивания или просто объявлением. Объявленная переменная имеет область видимости оператора *for*.

Для инициализации более одной переменной могут быть использованы выражения с запятой.

Например,

```
for (factorial=n, i=n-1;i>1;--i)
factorial *=i;
```

Любое или все выражения в операторе *for* могут отсутствовать. Вместо пропущенного параметра ставится точка с запятой.

Рассмотрим пример на организацию регулярного цикла.

Пример.

Выяснить, имеются среди n целых данных чисел положительные числа, кратные 5 и подсчитать их количество.

```
#include <iostream.h >
void main()
{
int n=20;
int a, i ,k;
k=0;
for (i=1; i<=n; ++i)
{
cout<<"Введите число"<<endl;
cin>>a;
if ((a>0)&&(a/5=a%5)) ++k;
}
cout<<endl<<"Количество чисел, кратных 5="<<k;
}
```

Чтобы прервать нормальное выполнение цикла, можно использовать операторы *break* и *continue*. Оператор *continue* вызывает остановку текущей итерации цикла и немедленный переход к началу очередной итерации.

Рассмотрим пример, иллюстрирующий использование оператора *break*.

Пример.

```
for (i=0; i<10; ++i)
{
cin>>x;
if (x<0.0)
{
cout<<"Всё!"<<endl;
break; // Выход из цикла, если значение отрицательно
}
cout<<sgrt(x);
}
// break переходит сюда
.....
```

Следующий пример, иллюстрирует использование оператора *continue*. Фрагмент обрабатывает все символы, кроме цифр.

```
for (i=0; i<max; ++i)
{
cin.get(c);
if (isdeget(c))
continue;
.....        // Обработка других символов
// continue переходит сюда
}
```

Когда выполняется оператор *continue*, управление переходит в точку закрывающейся фигурной скобкой, что приводит к выполнению цикла с начала.

Оператор *while.*

while (условие)

оператор

Сначала вычисляется *условие*. Если оно истинно, то выполняется оператор, и управление передается обратно в начало цикла *while*. В результате тело цикла *while* выполняется до тех пор, пока условие не станет ложно. С этого момента управление передается оператору, следующему за циклом.

Рассмотрим пример на организацию итерационного цикла.

Пример.

Пусть даны числа a,b (a>1) и надо получить все числа бесконечной последовательности a, a2, a3…, меньше числа b.

```
#include <iostream.h>
void main()
{
int a, b;
double c;
cout<<"Введите значения a и b:"<<endl;
cin>>a>>b;
c=a;
while c<b
 {
   cout<<endl<<c<<endl;
   c*=a;
 }
}
```

1.5.4. Программирование алгоритмов со структурой вложенных циклов

Структура программы с вложенными циклами образуется в том случае, когда внутри одного цикла находится один или несколько других циклов. При этом область действия внутреннего цикла располагается внутри охватывающего его цикла.

При организации вложенных циклических структур необходимо обратить внимание на правильность выбора внешних и внутренних циклов. В некоторых постановках задач эти циклы можно поменять местами и алгоритм решения останется верным. В ряде других постановок задач замена внешнего цикла на внутренний приводит к изменению постановки задачи, а, следовательно, к получению неверного результата.

Пример.

Пусть дано натуральное n и требуется вычислить сумму степеней $\left(\frac{1}{1}\right)^n + \left(\frac{1}{2}\right)^n + \ldots \left(\frac{1}{n}\right)^n$

```
#include <iostream.h>
void main()
{
int i, j, n;
float a, s, p;
s=0.0;
cout<<"Введите степень:"<<endl;
cin>>n;
for (i=1; i<=n; ++i)
{
a=1.0/i; p=a;
```

```
for (j=2; j<=n; ++j)
{
p*=a;
}
s+=p;
}
cout<<"Сумма степеней="<<s;
}
```

Раздел 2:
ФУНКЦИИ И УПРАВЛЕНИЕ ПАМЯТЬЮ

Функция в языке C++ является основным элементом в структуре программы. Функции представляют собой программные блоки, которые могут вызываться из разных участков программы.

2.1. Функции

Всякая задача может быть разбита на подзадачи, каждую из которых можно непосредственно представить в виде кода или разбить на еще более мелкие подзадачи. Данный метод несет название *пошагового уточнения* (stepwise refinement). *Функции* в языке C++ служат для записи программного кода этих непосредственно решаемых задач. Такие функции используются другими функциями, и в конечном итоге – функцией *main()* для решения исходной задачи. Функция *main()* используется как точка входа для выполнения программы. Обычно функция *main()* неявно возвращает значение 0, что означает нормальное завершение программы. Другие возвращаемые значения нужно задавать явно (с помощью ключевого слова *return*).

Оператор *return* используется для двух целей:

- когда он выполняется, управление программой передаётся немедленно обратно в вызывающее окружение;
- если за конечным словом *return* следует какое-либо выражение, то его значение также передается в вызывающее окружение.

Вот некоторые примеры:

```
return;
return 3;
return (a+b).
```

2.1.1. Определение функции

В C++ код, описывающий, что делает функция, называется определением функции (*function definition*). Выглядит это так:

```
заголовок_функции
{
операторы – тело определения функции
}
```

Заголовок функции – это

Тип имя(список_объявлений_параметров).

Спецификация *тип*, стоящая перед именем функции, является *возвращаемым типом*. Он определяет тип значения, возвращаемого функцией (если оно вообще возвращается).

Пример.

Программа, которая издает звуковой сигнал.

```
#include <iostream.h>
```

```
const char bell='\a';
void ring()
{
cout<<bell;
}
void main()
{
ring();
}
```

Для генерации звукового сигнала использован специальный символьный литерал. Поскольку функция не возвращает значения, возвращаемый тип этой функции – *void*.

Список_объявлений_параметров – это идентификаторы, они могут использоваться внутри тела функции. Параметры в определении функции также называют *формальными параметрами*.

Формальные параметры – это параметры, вместо которых будут подставлены фактические значения, передаваемые функции в момент вызова. После вызова функции значение аргумента, соответствующее формальному параметру, используется в теле выполняемой функции. В C++ такие параметры являются *вызываемыми по значению* (call-by-value). Когда применяется вызов по значению, переменные передаются функции как аргументы, их значения копируются в соответствующие параметры функции, а сами переменные не изменяются в вызывающем окружении.

Пример.

Программа, которая издаёт звуковой сигнал заданное количество раз.

```
#include <iostream.h>
const char bell='\a';
void ring(int k)
{
int i;
for (i=0; i<k; ++i)
cout<<bell;
}
void main()
{
int n;
cout <<"Введите положительное целое:"<<endl;
cin>>n;
ring(n);
}
```

Рассмотрим пример на использование оператора *return*.

Пример.

Программа, которая вычисляет наименьшее из двух целых чисел.

```
#include <iostream.h>
int min(int x, int y)
{
if (x<y)
return x;
else
return y;
}
void main()
{
int j, k, m;
cout<<"Введите два целых числа:"<<endl;
cin>>j>>k;
```

```
m=min(j,k);
cout<<endl<<m<<"Наименьшее из"<<j<< "и"<<k<<endl;
}
```

2.1.2. Прототипы функций

Возвращаемый тип, идентификатор функции и типы параметров составляют прототип функции. *Прототип функции* (function prototype) – это объявление функции, но не её определение.

Функция может быть объявлена до того, как она определена. Определение функции может идти позже в этом же файле, браться из библиотеки или из указанного пользователем файла.

Прототип функции имеет следующую форму:

тип имя(список_объявлений_аргументов);

Список_объявлений_аргументов может быть пустым, содержать единственное объявление или несколько объявлений, разделенных запятыми. Такая информация позволяет компилятору отслеживать совместимость типов.

В последнем примере использована функция *min()*. Её прототип будет выглядеть следующим образом:

int min(int, int);

Возвращаемый тип функции и тип аргументов в списке указываются явно. Определение функции *min()* должно соответствовать этому объявлению. Прототип функции может также содержать имена аргументов. В этом случае *min()* будет выглядеть так:

int min (int x, int y);

Пример.

Программа, вычисляющая сумму и среднее трех целых чисел.

```
#include <iostream.h>
int add3(int, int, int);  // Прототип функции add3
double average(int);  // Прототип функции average
void main()
{
int score_1, score_2, score_3, sum;
cout<<"Введите три оценки:";
cin>>score_1>>score_2>>score_3;
sum=add3(score_1, score_2, score_3);
cout<<endl<<"Их сумма="<<sum;
cout<<endl<<"Их среднее="<<average(sum);
}
int add3(int a, int b, int c)
{
return (a+b+c);
}
double average(int s)
{
return (s/3.0);
}
```

2.1.3. Перегрузка функций

Перегрузка функций (function overloading) является одной из особенностей ООП, отражающей свойство полиморфизма. Перегрузка функций позволяет определять несколько функций с одним и тем же именем, если функции имеют различное количество и (или) типы аргументов. Перегрузка функций обычно используется для создания функций, предназначенных для выполнения однотипных задач, оперирующих с различными структурами и ти-

пами данных. При вызове перегруженной функции компилятор определяет адрес вызова нужной функции путём анализа количества, типа и порядка следования аргументов.

Пример.

Программа нахождения среднего значения элементов массива.

```
#include <iostream.h>
double avg_arr(const int a[] , int size)
{
int sum=0;
for (int i=0; i<size; i++)
sum+=a[i];
return (sum/size);
}
double avg_arr(const double a[], int size)
{
double sum=0;
for (int i=0; i<size; i++)
sum+=a[i];
return (sum/size);
}
```

Следующий код демонстрирует обращение к функции *avg_arr()*.

```
void main()
{
int w[5]={1,2,3,4,5};
double x[5]={1.2,2.2,3.3,4.4,5.5};
cout<<avg_arr(w,5)<<"Среднее для int"<<endl;
cout<<avg_arr(x,5)<<"Среднее для double"<<endl;
}
```

С перегрузкой функций связана еще одна возможность С++. Она называется *аргумент по умолчанию*. Аргумент по умолчанию позволяет дать параметру значение по умолчанию, если при вызове соответствующий аргумент не задан. Если в программе предварительно описывается прототип функции, то аргументы по умолчанию указываются в этом прототипе.

Например,

```
void f(int i=0, int j=1); // прототип функции f(...)
...
void f(int i, int j)
{
... // тело функции
}
```

Теперь эту функцию можно вызывать тремя способами:

```
f(); // i по умолчанию равно 0, j по умолчанию равно 1
f(10); // i равно 10, j по умолчанию равно 1
f(11, 22); // i равно 11, j равно 22.
```

Задать значение j, установив i по умолчанию, нельзя. Все параметры по умолчанию должны находиться правее аргументов, передаваемых обычным путем.

Аргументами по умолчанию могут быть только константы или глобальные переменные.

Применение аргумента по умолчанию является простейшей формой перегрузки функций. Функция при этом остается одна, следовательно, и алгоритм, реализуемый при вызове этой функции, - один. А в перегруженных функциях, могут быть реализованы различные алгоритмы.

2.1.4. Встраиваемые функции

Встраиваемые функции в языке С++ задаются с помощью спецификатора *inline*. Если функция имеет спецификатор *inline*, то тело такой функции встраивается в блок после места её вызова. Преимуществом таких функций является быстродействие, заключающееся в экономии времени процессора на вызов функции и обеспечения механизма возврата из неё.

Следующая программа определяет функции max и min как inline.

```
#include <iostream.h>
inline int max(int a, int b)
{
if (a>b) return (a);
  else return (b);
}
inline int min(int a, int b)
{
if (a<b) return (a);
  else return (b);
}
void main()
{
cout<<"Минимум из 1001 и 2002 равен"<<min(1001,2002)<<endl;
cout<<"Максимум из 1001 и 2002 равен"<<max(1001,2002)<<endl;
}
```

В данном случае компилятор С++ заменит каждый вызов функции соответствующими операторами функции. Производительность программы увеличивается без ее усложнения.

Функция должна быть короткой. Компилятор может игнорировать директиву *inline*, если функция слишком длинная.

Встраиваемые функции можно перегружать.

2.1.5. Локальные и глобальные переменные

При объявлении переменных им можно задавать конкретные инициализирующие значения. Такое свойство языка программирования называется *динамической инициализацией*. В программах иногда возникают ситуации, когда имя глобальной переменной совпадает с именем локальной переменной, объявленной внутри функции или блока. Для того чтобы получить доступ из блока, где объявлена локальная переменная, к глобальной с таким же именем, в языке С++ применяется операция *«область видимости»*. Обозначается она ::.

Пример.

```
#include <iostream.h>
int i=0; //глобальная переменная
void function()
{
   int i=1; //локальная переменная 1
   {
      int i=2; //локальная переменная 2
      ::i++;  // операция области видимости
      i++;
      cout<<"Global = "<<::i<<"\n";
      cout<<"Local 2 = "<<i<<"\n";
   }
   i++;
   cout<<"Local 1 = "<<i<<"\n";
}
void main()
{
   function();
```

```
}
Global = 1
Local 2 = 3
Local 1 = 2
```

Все локальные переменные являются переменными с локальным временем жизни. Это означает, что доступ к локальным переменным возможен лишь внутри функции, где они объявлены. Вне пределов функции локальные переменные недоступны. Все локальные переменные имеют по умолчанию атрибут *auto*. При входе в функцию выделяется память под такие переменные, а после выхода из блока - они исчезают. Любую переменную или функцию можно задать с таким атрибутом, который обеспечивает глобальное время жизни. Такие переменные называются статическими и имеют атрибут *static*. Они начинают свое существование с самого начала работы программы. Если программист явно не проинициализировал статическую переменную, то компилятор по умолчанию присваивает ей значение 0. Локальные статические переменные имеют область видимости лишь в той функции, где они объявлены, и сохраняют свои значения на протяжении всего жизненного цикла программы. Рассмотрим пример, иллюстрирующий это.

Пример.

```
#include <iostream.h>
void function()
{
   static int n=0;
   n++;
   cout<<n<<" start of function\n";
}
void main()
{
   function();
   function();
   function();
}
```

Результат работы программы:

```
1 start of function
2 start of function
3 start of function
```

2.2. Выделение динамической памяти

Динамическая память - это предоставляемая системой область памяти для объектов, время жизни которых напрямую управляется программистом.

Динамически распределяемая память – это область памяти, разделенная на множество пронумерованных ячеек, в которых записана информация. В отличие от стека переменных, ячейкам свободной динамической памяти нельзя присвоить имя, а доступ к ним осуществляется посредством указателя, хранящего адрес нужной ячейки. Важной особенностью динамической памяти является то, что выделенная в ней область памяти не может использоваться до тех пор, пока явно не будет освобождена. Еще одной особенностью динамической памяти является то, что доступ к данным можно получить только из функции, в которой есть доступ к указателю, хранящему нужный адрес.

В языке С++ для выделения и освобождения динамической памяти используются следующие два оператора *new* и *delete*. Данные операторы являются стандартом языка С++ и не требуют подключение какой-либо библиотеки. Синтаксис операторов выделения и освобождения памяти для одной переменной определенного типа выглядит следующим образом:

```
указатель_на_тип = new тип;
delete указатель_на_тип;
```

Если по каким-либо причинам *new* не может выделить память (недостаточно свободной памяти), то оператор возвратит указатель на 0. В противном случае возвращается указатель на выделенный блок памяти. Оператор *new* автоматически выделяет требуемое количество памяти для хранения объекта заданного типа, и автоматически возвращает указатель на заданный тип. При выделении памяти под переменные можно автоматически производить их инициализацию, а также выделять память под массивы и структуры данных – в том числе и объектов.

Синтаксис операторов выделения и освобождения памяти под массивы выглядит следующим образом:

указатель_на_тип = new тип[размерность];

delete [] указатель_на_тип.

Пример.

Выделение и освобождение памяти под двумерный массив.

```
#include <iostream.h>
void main()
{
   int *ptr=new int[5,5];
   for (int i=0 ;i<5 ; i++)
   {
      for (int j=0 ;j<5; j++)
      {
         if (i==j) ptr[i,j]=i;
         else ptr[i,j]=0;
        cout<<ptr[i,j];
      }
      cout<<"\n";
   }
   delete [] ptr;
}
```

Результат работы программы:

```
00000
01000
00200
00030
00004
```

В этом примере *ptr*– указатель, а **ptr* – значение переменной, на которую указывает *p*. Прямое значение *ptr* – это адрес памяти, **ptr* – это непрямое значение *ptr*, т.е. значение, находящееся по адресу, который хранится в указателе *p*. * – операция определения значения по адресу (разадресация).

2.3. Ссылки

В большинстве языков программирования аргументы в функциях передаются либо при помощи ссылки (*by reference*) или по значению (*by value*). В первом случае функция работает непосредственно с переменной, во втором только с её значением. Различие состоит в том, что переменную, переданную по ссылке, функция может модифицировать, а переданную по значению - нет. В стандартном языке C параметры в функции передаются только по значению, и для того чтобы модифицировать ее внутри функции, необходимо в качестве параметра передать указатель на переменную. Это демонстрирует следующий пример:

```
int inc(int *var)
{
   (*var)++;
}
void main()
```

```
{
   int n=0;
   inc(&n);
}
```

В приведенном примере, для изменения переменной *n* в функции *inc* необходимо передать в нее указатель на переменную, а при вызове функции указать, что в качестве аргумента в функцию передается адрес переменной *n*.

При использовании ссылок в языке C++ нет необходимости в передаче адреса переменной в функции и в использовании операции * для модификации переменной внутри тела функции.

Рассмотрим этот же пример, но только с использованием ссылки.

```
int inc(int &var)
{
   var++;
}
void main()
{
   int n=0;
   inc(n);
}
```

Раздел 3:
ВВЕДЕНИЕ В ТЕХНОЛОГИЮ ПРОГРАММИРОВАНИЯ

Лекция проводится в интерактивной форме с текущим контролем (0,5 час.)

3.1. Общие положения

Под *технологией программирования* (ТП) будем понимать совокупность производственных процессов, приводящую к созданию требуемых программных средств (ПС), а также описание этой совокупности процессов. ТП будем понимать в широком смысле как технологию разработки ПС, включая в нее все процессы, начиная с момента зарождения идеи этого средства, и, в частности, связанные с созданием необходимой программной документации.

Используется в литературе и близкое к ТП понятие *программной инженерии*, определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения ПС. Главное различие между технологией программирования и программной инженерией как дисциплинами заключается в способе рассмотрения и систематизации материала. В ТП акцент делается на изучении процессов разработки ПС и порядке их прохождения – методы и инструментальные средства разработки ПС используются в этих процессах. Тогда как в программной инженерии изучаются различные методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей – эти методы и средства могут использоваться в разных ТП.

Не следует также путать ТП с *методологией программирования*. В ТП методы рассматриваются «сверху» – с точки зрения организации технологических процессов, а в методологии программирования методы рассматриваются «снизу» – с точки зрения основ их построения.

Имея ввиду, что надежность является неотъемлемым атрибутом ПС, будем рассматривать ТП как технологию разработки надежных ПС. Это означает, что:

– будем рассматривать все процессы разработки ПС, начиная с момента возникновения замысла ПС;

– будут интересовать не только вопросы построения программных конструкций, но и вопросы описания функций и принимаемых решений с точки зрения их человеческого (неформального) восприятия;

– в качестве продукта технологии принимается надежная ПС.

3.2 Технологии программирования и информатизация общества

Этапы развития ТП:

-1 этап. В 50-е годы мощность компьютеров (первого поколения) была невелика, а программирование для них велось в машинном коде. Решались, главным образом, научно-технические задачи (счет по формулам), задание на программирование содержало, как правило, достаточно точную постановку задачи. Родилась фундаментальная для ТП концепция модульного программирования, первые языки программирования высокого уровня (Фортран);

- 2 этап. В 60-е годы происходит широкое использование языков программирования высокого уровня (Алгол 60, Фортран, Кобол) и др. Появление в компьютерах 2-го поколения прерываний привело к развитию мультипрограммирования и созданию больших программных систем. Широко стала использоваться коллективная разработка;

- 3 этап. В 70-е годы получили широкое распространение информационные системы и базы данных. Началось интенсивное развитие ТП в следующих направлениях:

- обоснование и широкое внедрение нисходящей разработки и структурного программирования;

- развитие абстрактных типов данных и модульного программирования;

- исследование проблем обеспечения надежности и мобильности ПС;

- создание методики управления коллективной разработкой ПС;

- появление инструментальных ПС поддержки ТП;

- 4 этап. 80-е годы характеризуются широким внедрением персональных компьютеров (ПК) во все сферы человеческой деятельности и тем самым созданием обширного и разнообразного контингента пользователей ПС. Это привело к бурному развитию пользовательских интерфейсов и созданию четкой концепции качества ПС. Появляются языки программирования, учитывающие требования ТП. Начинается бурный процесс стандартизации технологических процессов и, прежде всего, документации, создаваемой в этих процессах. Создаются различные инструментальные среды разработки и сопровождения ПС. Развивается концепция компьютерных сетей;

- 5 этап. 90-е годы знаменательны широким охватом всего человеческого общества международной компьютерной сетью, ПК стали подключаться к ней как терминалы. Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. Стали бурно развиваться CASE-технологии разработки ПС и связанные с ней формальные методы спецификации программ.

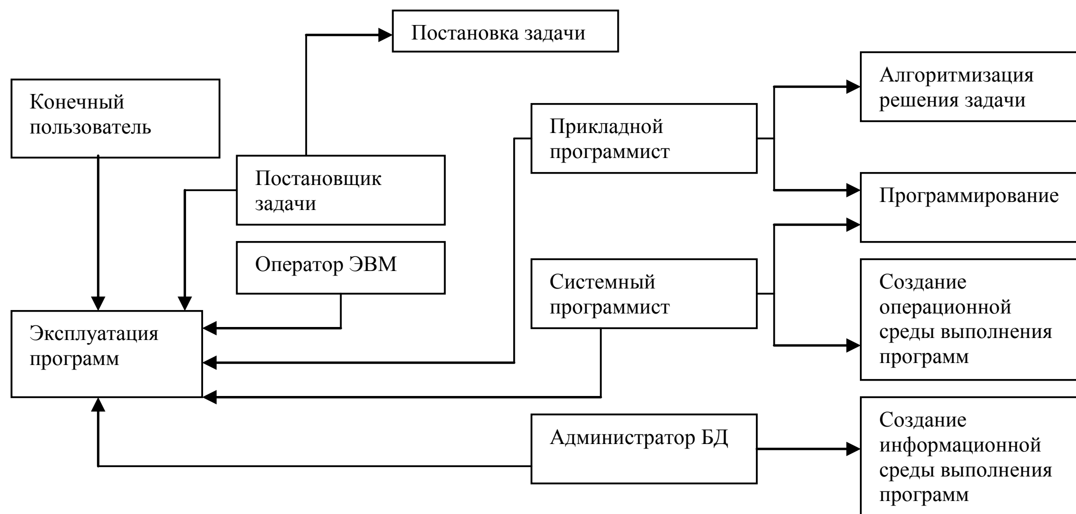

Рисунок 1.1 – Схема взаимодействия специалистов, занятых разработкой и эксплуатацией программного обеспечения

3.3. Ключевые понятия ООП

Любая программа в окончательном варианте представляет собой набор инструкций процессора. Всё, что написано на любом языке программирования, – более удобная, упрощённая запись этого набора инструкций. Чем выше уровень языка, тем более в простой форме записываются одни и те же действия.

С ростом объёма программы невозможно удержать в памяти все детали и становится необходимым структурировать информацию, выделять главное и отбрасывать несущественное. Этот процесс называется повышением степени абстракции программы.

Первый шаг к повышению абстракции – использование функций, позволяющее после написания и отладки функций отвлечься от деталей её реализации, поскольку для вызова функции требуется знать только её интерфейс.

Следующий шаг – описание собственных типов данных, позволяющих структурировать и группировать информацию, представляя её в более естественном виде. Для работы с собственными данными требуются специальные функции. Нужно сгруппировать эти функции с описанием типов данных в одном месте программы. При этом для использования этих типов и функций не требуется полного знания того, как они написаны – необходимы только описания интерфейсов. Объединение в модули описаний типов данных и функций, предназначенных для работы с ними, со скрытием от пользователя модуля несущественных деталей, является дальнейшим развитием структуризации программы.

Все три описанных выше метода повышения абстракции преследуют цель упростить структуры программы, т. е. представить её в виде меньшего количества более крупных блоков и минимизировать связи между ними. Это позволяет управлять большим объёмом информации и успешно отлаживать сложные программы.

Класс является типом данных, определяемых пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные (внутреннее представление, множество значений, операции и функции).

Идея классов является основой ООП. Основные концепции ООП были разработаны в языках Simula-63, Smalltalk, но в то время не получили широкого применения из-за трудностей освоения и низкой эффективности реализации. В языке C++ эти концепции реализованы красиво и непротиворечиво, что явилось основой успешного распространения этого языка.

ООП часто называют новой парадигмой программирования. В программировании этот термин используется для определения модели вычислений, т.е. способа структурирования информации, организации вычислений и данных. Объектно-ориентированная программа строится в терминах объектов и их взаимосвязей.

Объект – это осязаемая сущность, которая чётко проявляет своё поведение. Объект состоит из следующих трёх частей:

- имя объекта;
- состояние (переменные состояния);
- методы (операции).

Интерфейс объекта с его окружением определён полностью его методами, так как к его состоянию нет другого доступа извне, как через методы.

Объект сохраняет своё состояние от обращения к обращению. Изменение состояний производится только через вызов методов этого объекта. Этим существенно ограничивается возможность введения объекта в недопустимое состояние и/или несанкционированное разрушение объекта. Возможность управлять состояниями объекта через вызов методов в конечном итоге будет определять его поведение.

Объект может посылать сообщения другим объектам и принимать сообщения от них. *Сообщение* – это совокупность данных определённого типа, передаваемых объектом отправителем объекту получателю, имя которого указывается в сообщении. Получатель реагирует или никак не реагирует (защита) на сообщение выполнением некоторой операции (метода), имя которой также может быть указано в сообщении.

По своему смыслу объект является представителем некоторой реальной сущности – реального объекта, процесса, ситуации, которая:

- поддаётся хранению и обработке;
- способна воздействовать на другие объекты и вычислительную среду, посылая сообщения, и реагировать на принимаемы сообщения.

Совокупность объектов в системе ООП образует среду, в которой выполняются вычисления путём обмена сообщениями между объектами. Состояние вычислительной среды в ООП оказывается разделенным на состояния объектов. Это в принципе отличает объектно-ориентированные вычисления от вычислений, заданных на процедурно-ориентированных языках. Процедуры выполняются в общей памяти, в то время как объекты выполняют свои операции с учётом данных одного сообщения и своего собственного состояния.

Объекты с одинаковыми свойствами, т.е. с одинаковыми наборами переменных состояния и методов, образуют класс.

Основными свойствами ООП являются инкапсуляция, наследование и полиморфизм.

Инкапсуляция – это объединение в едином объекте данных и кодов, оперирующих с этими объектами. В терминологии ООП данные называются *полями класса*, а коды – *объектными методами*. Инкапсуляция позволяет в максимальной степени изолировать объект от внешнего воздействия. Она существенно повышает надёжность разрабатываемых программ, т.к. локализованные в объекте функции обмениваются с программой небольшими объёмами данных, причём количество и тип этих данных контролируются. В результате замена или модификация функций и данных, инкапсулированных в объект, не влечёт за собой плохо отслеживаемых последствий для программы в целом. Инкапсуляция даёт также возможность лёгкого обмена объектами, переноса их из одной программы в другую.

Наследование – это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объём программы. Наследование выполняет в ООП несколько важных функций:

- моделирует концептуальную структуру предметной области;
- экономит описания, позволяя использовать их многократно для задания разных классов;
- обеспечивает пошаговое программирование больших систем путём многократной конкретизации классов.

Полиморфизм – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемые действия во время выполнения программы.

Примеры полиморфизма:

- перегрузка функций, когда из нескольких вариантов выбирается наиболее подходящая функция по соответствию её прототипа передаваемым параметрам;

- использование шаблонов функций, когда один и тот же код видоизменяется в соответствии с типом, переданным в качестве параметра.

Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

Благодаря тому, что программа представляется в терминах поведения объектов, при программированиии используются понятия, более близкие к предметной области, следовательно, программа легче читается и понимается. Это является преимуществом ООП.

Таким образом, ООП = ИНКАПСУЛЯЦИЯ + НАСЛЕДОВАНИЕ + ПОЛИМОРФИЗМ.

Раздел 4: КЛАССЫ В С++

Лекция проводится в интерактивной форме с разбором конкретных ситуаций (1 час.)

4.1. Описание объектов при помощи классов

Основой языка С++ являются классы. *Класс* является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними. Классы позволяют в программах группировать данные объекта и функции объекта (методы), которые оперируют с этими данными, в одной переменной.

Определение класса начинается с ключевого слова *class*.

```
class имя_класса
{
  //функции
  //данные
};
```

Каждая функция и элемент данных, описанные внутри класса имеют атрибут доступа. В языке С++ существуют три атрибута доступа:

- *public* – открытый (общий). Доступ к элементам класса может быть осуществлен как из самого класса, так и извне класса;
- *private* – закрытый (частный). Доступ к элементам класса может быть осуществлен только из функций самого класса. Однако доступ к элементам с атрибутом *private* можно осуществлять и не только внутри класса;
- *protected* – защищенный.

По умолчанию все элементы класса имеют атрибут доступа *private*. Атрибут задается ключевым словом и символом “:”. Действие атрибута сохраняется до следующего атрибута или до закрывающей фигурной скобки в объявлении класса.

```
class C
{
  int X;
  public:
  void function();
  private:
  int Y;
};
```

Для доступа к элементам класса используется символ:

- «.»

```
C obj;
obj.function();
```

- *«–>», если используется не имя объекта, а указатель*

```
C *obj;
obj–>function();
```

Описание функций (*методов*) класса можно проводить как внутри описания класса, так и за его пределами. В первом случае компилятор воспримет функцию как *inline*-функцию, во втором – как обычную.

```
class C
{
  public:
  void f1() {cout<<"Inline function\n";}
  void f2();
};
void C::f2()
{  cout<<"Ordinary function\n";}
void main()
{
  C obj;
  obj.f1(); obj.f2();
}
```

Результаты работы программы:

Inline function
Ordinary function

Для описания функции за пределами класса внутри тела класса описывается только прототип функции. Для описания тела такой функции за пределами класса необходимо использовать механизм привязки функции.

Использование методов внутри класса осуществляется при помощи имени метода, за пределами класса – через имя объекта, которому принадлежит метод. Методы класса могут быть перегружены, но только в пределах описания класса.

```
#include <iostream.h>
class C
{
public:
      void f() {cout<<"Inline\n";}
      void f(int);
};
void C::f(int n)
{     cout<<"Number="<<n<<"\n";}
void main()
{
      C obj;
      obj.f(); obj.f(5);
}
```

Результаты работы программы:

Inline
Number=5

4.2. Конструкторы и деструкторы

Внутри класса нельзя явно инициализировать переменные класса. Инициализацию можно проводить вне пределов класса, если переменная является общедоступной или при помощи функций класса. Для инициализации переменных предусмотрен специальный механизм использования функций-конструкторов (или просто – *конструкторов*). Данная функция является элементом класса и вызывается всякий раз при создании объекта данного класса. Это означает, что если внутри конструктора расположены операции по инициализации каких-либо переменных, то такая инициализация будет происходить автоматически каждый раз при создании новых объектов. Конструктор может быть описан как обычная функция класса,

т.е. внутри и вне тела класса. Имя конструктора всегда совпадает с именем класса. Конструктор не может возвращать какое-либо значение.

Пример.

```
#include <iostream.h>
class C
{
	char *str;
	int num;
public:
	C() {cout<<"Constructor\n";str='a';num=0;}
	C(char *);
	C(int);
	void print();
};
C::C(char *s) {str=s; num=0; cout<<"String init\n";}
C::C(int n) {num=n; str="Nothing";cout<<"Number init\n";}
void C::print()
{  cout<<"String="<<str<<"Number="<<num<<"\n";}
void main()
{
	C obj1, obj2("Object2"), obj3(3);
	obj1.print();
	obj2.print();
	obj3.print();
}
```

Результаты работы программы:

```
Constructor
String init
Number init
String=a Number=0
String=Object2 Number=0
String=Nothing Number=3
```

При создании объектов может использоваться один из трёх вариантов:

- создание объектов по умолчанию;
- создание объектов со специальной инициализацией;
- создание объектов путём копирования других объектов.

Обратной смысл конструктора принадлежит специальной функции класса – *деструктору*, который если объявлен внутри класса будет вызываться всякий раз при уничтожении объекта (окончание программы, выход из области видимости объекта, освобождение памяти, выделенной под объект). Имя деструктора, как и конструктора, совпадает с именем класса, но перед именем деструктора ставится специальный символ – тильда «~». Деструктор не может возвращать значения, а также в деструктор нельзя передавать аргументы. В деструкторе осуществляются действия, связанные с освобождением динамической памяти, которая была выделена некоторым структурам данных, описанных внутри класса. Конструкторы и деструктор должны иметь атрибут доступа *public*, в противном случае объект не будет создан или уничтожен.

Пример.

```
#include <iostream.h>
class C
{
public:
	C() {cout<<"Constructor\n";}
	~C() {cout<<"Destructor\n";}
};
```

```
void main()
{
  C obj;
}
```

Результаты работы программы:

Constructor
Destructor

Рассмотрим пример, в котором вызываются конструкторы и деструкторы объектов.

Пример.

```
#include <iostream.h>
class C
{
	int id;
public:
	C(int n) {id=n; cout<<"Constructor "<<id<<"\n";}
	~C() {cout<<"Destructor "<<id<<"\n";}
} ;
function() { static C obj5(5);}
C obj1(1);
void main()
{
  cout<<"BEGIN\n";
  C obj2(2);
	{
		C obj3(3);
	}
  C obj4(4);
  function();
  function();
  function();
  cout<<"END\n";
}
```

Результаты работы программы:

Constructor 1
BEGIN
Constructor 2
Constructor 3
Destructor 3
Constructor 4
Constructor 5
END
Destructor 5
Destructor 4
Destructor 2
Destructor 1

Конструкторы глобальных объектов вызываются в первую очередь до выполнения первого действия в функции *main()*. Деструкторы глобальных объектов вызываются самыми последними, после того как отработала функция *main()*. Жизнь локальных объектов, как и локальных переменных, ограничивается областью видимости. Конструкторы таких объектов вызываются в месте создания объекта, деструкторы – после выхода из блока (области видимости). Конструкторы статических объектов вызываются один раз при первом достижении команды создания объекта, а деструкторы – после завершения функции *main()*. В общем слу-

чае, если несколько объектов создаются в одном блоке, то конструкторы таких объектов вызываются в порядке создания объектов, а деструкторы – в обратном порядке.

4.3. Конструкторы копирования

Пример.

```
#include <iostream.h>
class C
{
public:
	int *ptr;
	C(int number) {ptr=new int(number); cout<<"Constructor\n";}
	~C() {delete ptr; cout<<"Destructor\n";}
	print() {cout<<"*ptr="<<(*ptr)<<"\n";}
};
void main()
{
	C obj1(5);
	{
		C obj2=obj1;
		obj2.print();
	}
	C obj3(10);
	obj1.print();
}
```

Результаты работы программы:

```
Constructor
*ptr=5
Destructor
Constructor
*ptr=10
Destructor
Destructor
```

Первый раз конструктор вызывается для объекта *obj1*. При создании объекта *obj2* конструктор не вызывается, т.к. происходит побитное копирование объекта *obj1* в объект *obj2*. Это означает, что указатели на целое *ptr* в объектах *obj1* и *obj2* равны. При выходе из области видимости объекта *obj2* вызывается деструктор, который освобождает память под переменную целого типа, но в этом случае как для временного объекта *obj2*, так и для объекта *obj1*. В данной программе сразу же после уничтожения объекта *obj1* происходит автоматическое выделение памяти под объект *obj3*, который размещается на месте объекта *obj1*. Для предотвращения таких нежелательных действий, в языке C++ применяется механизм, называемый *конструктором копирования*. В общем случае конструктор копирования имеет следующий синтаксис:

```
имя_класса (const имя_класса &имя_объекта){....}
```

Теперь, если в предыдущем примере описать следующий конструктор копирования:

```
C(const C &obj) {ptr=new int (*obj.ptr); cout<<"Copy constructor \n";},
```

то при явной инициализации локального объекта *obj*2 произойдет вызов конструктора копирования, при котором выделится память под целое, а значение в выделенную память скопируется из объекта *obj*, который присваивается текущему объекту.

Результат работы предыдущей программы будет выглядеть следующим образом:

```
Constructor
```

Copy constructor
**ptr=5*
Destructor
Constructor
**ptr=5*
Destructor
Destructor

Следует отметить, что конструктор копирования вызывается только при явной инициализации объектов, когда одному объекту присваивается другой. В противном случае компилятор воспринимает оператор (=) как обыкновенную операцию присваивания и производит побитное копирование содержимого одного объекта в другой. Существует еще одна ситуация, при которой вызывается конструктор копирования. Если описана функция, которая возвращает временный объект, созданный внутри функции.

Пример.

```
#include <iostream.h>
class C
{      int n;
       public:
       C(int number) {n=number; cout<<"Constructor\n";}
       C(const C &obj ) {n=obj.n; cout<<"Copy constructor\n";}
       ~C() {cout<<"Destructor\n";}
       print() {cout<<"n="<<n<<"\n";}
};

C fnul()
{       cout<<"BEGIN:fnul\n";
        C tmp(0);
        cout<<"END:fnul\n";
        return tmp;
}

void main()
{       cout<<"BEGIN:main\n";
        C obj1(5); obj1.print(); obj1=fnul(); obj1.print();
        cout<<"END:main\n";
}
```

Результаты работы программы:

```
BEGIN:main
Constructor
n=5
BEGIN:fnul
Constructor
END:fnul
Copy constructor
Destructor
Destructor
n=0
END:main
Destructor
```

При вызове функции *fnul* создается временный объект, предназначенный для хранения возвращаемого объекта. Далее, когда происходит инициализация временного объекта локальным объектом *tmp* внутри функции *fnul()*, происходит вызов конструктора копирования. После этого происходит последовательный вызов деструкторов временного объекта и объек-

та *tmp*. Значение временного объекта побитно копируется в объект *obj1* без вызовов конструктора копирования и деструктора.

4.4. Статические компоненты класса

С помощью модификатора *static* можно описывать статические поля и методы класса. Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемые всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, т.е. не дублируются.

Пример.

```
#include <iostream.h>
class C
{
  static int n;    // объявление в классе
  public:
  C() {n++; cout<<"Object #"<<n<<"\n";}
};
int C::n=0;     // определение в глобальной области
void main()
{
  C obj1,obj2,obj3;
}
```

Результаты работы программы:

```
Object #1
Object #2
Object #3
```

Данный пример заключается в определении статической переменной внутри класса и повторного определения этой же переменной за пределами класса с явной инициализацией или без нее. В последнем случае компилятор произведет сам инициализацию этой переменной значением 0.

4.5. Указатели, ссылки и массивы объектов. Инициализация объектов

В языке С++ можно создавать массивы объектов. Синтаксис создания массивов объектов аналогичен синтаксису создания обыкновенных объектов в С++. Если задан произвольный класс *C*, то для того чтобы создать одномерный массив объектов, можно воспользоваться следующей записью:

```
class C {...};
void main()
{
  C objs[4]; // массив объектов с параметрами по умолчанию
}
```

С созданием массивов объектов связано одно ограничение. При объявлении объекта вызывается его конструктор. Если у соответствующего класса есть только конструкторы с параметрами, то необходимо при описании массива явно указывать передаваемые параметры.

Пример.

```
#include <iostream.h>
class C
{
```

```
    int n;
    public:
    C(int nn) {n=nn; cout<<"Constructor "<<n<<" \n";}
    ~C() {cout<<"Destructor\n";}
    print() {cout<<"n="<<n<<"\n";}
};
void main()
{
    C objs[5]={ C(0), C(1), C(2), C(3), C(4)};
    objs[2].print();
}
```

Результаты работы программы:

Constructor0
Constructor1
Constructor2
Constructor3
Constructor4
n=2
Destructor

Если конструктор имеет всего один аргумент, то данная запись может быть сокращена без использования явного вызова конструктора – обычным перечислением аргументов через запятую. Доступ к открытым членам класса при описании массивов объектов производиться через явное указание элемента массива (объекта) и точки.

К компонентам классов можно обращаться с помощью указателей и ссылок. Если в программе объявлен указатель на объект, то такой указатель можно применять для доступа к компонентам объектов, только в том случае, если имена классов, используемые для описания указателя и объекта, совпадают. Так, если в предыдущем примере описать указатель на класс C, то можно использовать следующую запись для вызова метода print() второго объекта в массиве:

```
C *ptr;
ptr=&objs[2];
ptr->print();
```

При помощи данного указателя можно выделить динамическую память под объект.

```
C *ptr;
ptr=new C;
ptr->print();
delete ptr;
```

Если конструктор имеет аргументы, то можно при выделении памяти под объект производить динамическую инициализацию путем передачи аргументов конструктору.

```
C *ptr;
ptr=new C(1);
ptr->print();
delete ptr;
```

При использовании ссылок на объекты в качестве аргументов функций встречаются ситуации, при которых нежелательны какие-либо изменения открытых компонентов классов. Для этого при описании ссылки ставится спецификатор *const*, говорящий о том, что компоненты данного объекта изменить нельзя.

Пример:

```
#include <iostream.h>
class C
{   public:
```

```
    int n;
    C(int nn) {n= nn;}
    print() {cout<<"n="<<n<<"\n";}
};
void function(const C &obj)
{   obj.n++; //ERROR!!!
    obj.print();
}
void main()
{ C obj(5); function(obj); }
```

Для каждого экземпляра класса существует специальный уникальный указатель. Например, при создании двух объектов одного и того же класса происходит вызов метода этих объектов. Компилятор выбирает определенную версию метода для конкретного объекта путем использования специального указателя на объект. Такой указатель называется указателем *this*. Запись **this* (разадресация указателя) и есть сам объект. Указатель *this* неявно используется внутри метода для ссылок на компоненты объекта. В явном виде этот указатель применяется для возвращения указателя (*return this;*) или ссылки (*return *this;*) на вызвавший объект. Указатель *this* можно также применять для идентификации поля класса в том случае, когда его имя совпадает с именем формального параметра метода.

Пример.

```
#include <iostream.h>
class C
{
    int n;
    public:
    C(int n) {this->n=n; cout<<"Address="<<this<<"\n";}
    print() {cout<<"n="<<n<<"\n";}
    ~C( ) {(*this).print();}
};
void main()
{
    C obj(15);
    obj.print();
}
```

Результаты работы программы:

```
Address=124506
n=15
```

4.6. Другие способы описания объектов

В языке С++ можно описывать объекты также при помощи структур и объединений. Применение структур для описания объекта скорее используется для поддержания совместимости языка С и С++ для тех разработчиков, которые переносят свои старые разработки в стиле языка С на С++. Для любой структуры в С++ можно описать любой метод, в том числе конструктор и деструктор. Одно отличие классов и структур выражается в том, что по умолчанию все компоненты структур имеют атрибут открытого доступа *public*.

Описывать объекты в С++ можно и при помощи объединений. Объединение представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания такой же, как у структуры, только используется ключевое слово *union*. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в переменной типа объединения хранится только одно значение. Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно не требуется.

Рассмотрим пример использования объединений для определения объектов.

```
#include <iostream.h>
union word
{
  unsigned N;
  unsigned char B[1];
  word(int N) {this->N=N; cout<<"Address="<<this<<"\n";}}
  convert() {cout<<int(B[1])<<" - "<<int(B[0])<<"\n";}
};
void main()
{  word W(3);
  W.convert();
}
```

Результаты работы программы:

Address=1245064
0-35

В одной и той же области памяти располагается переменная *N* (1 слово=2 байтам) и два байта в виде массива *char*.

Замечание: при использовании типа char символы русского алфавита кодируются отрицательными числами (-127…+128). Чтобы коды символов однозначно соответствовали кодировке ASCII, следует использовать тип unsigned char (0...255).

Компилятор разделяет общую память внутри *union* только под переменные. Если же внутри *union* описать указатель на внешнюю функцию, то он будет восприниматься как обычная переменная, которая будет разделять с другими общую память.

Существуют ограничения на использование объединений для описания объектов:

- объединения не могут наследовать какие-либо объекты;
- компоненты объединений не могут иметь атрибут *static*.

4.7. Дружественные функции

При написании программ часто встречаются ситуации, при которых необходимо иметь доступ к закрытым компонентам классов. При использовании обычных внешних функций это невозможно. Для решения данной задачи в языке C++ применяются дружественные функции (*friend functions*). Отличительная особенность дружественных функций заключается в том, что такие функции не являются компонентами классов, однако могут обращаться к закрытым компонентам. Дружественные функции задаются при помощи спецификатора *friend*, и их прототипы задаются в теле описания класса. Тело дружественной функции может быть описано за пределами описания класса и внутри класса. Но лучше производить описание тела дружественной функции за пределами класса, тем самым, подчеркивая, что она не является компонентом класса.

Пример.

```
#include <iostream.h>
class C
{
  int n;
  public:
  C(int n) {this->n=n;}
  print() {cout<<"n="<<n<<"\n";}
  friend void add(C &);
};
void add(C &obj)
{
  obj.n++;
```

```
}
int main()
{
  C obj(4);
  obj.print();
  add(obj);
  obj.print();
  return 0;
}
```

Результаты работы программы:

n=4
n=5

Чтобы дружественная функция осуществляла доступ к компонентам класса, необходимо в качестве аргумента передавать указатель или ссылку на объект. Объявлять прототип дружественной функции можно в любом месте описания класса. Дружественная функция не является компонентом класса, поэтому её нельзя вызвать с использованием имени объекта или указателя на объект. Вызов осуществляется обычным способом. Дружественные функции обычно применяются для доступа к компонентам сразу же нескольких классов. Рассмотрим пример, в котором описана дружественная функция, производящая сравнение закрытых переменных двух классов.

```
#include <iostream.h>
class B; // ссылка вперед
class A
{
    int i;
    public:
    A(int i) {this->i=i;}
    friend bool equal(A &, B &);
};
class B
{
    int i;
    public:
    B(int i) {this->i=i*i;}
    friend bool equal(A & , B &);
};
bool equal(A &obj1, B &obj2)
{
    if (obj1.i==obj2.i) return true;
    return false;
}
int main()
{
    A obj1(100); B obj2(10);
    cout<<equal(obj1,obj2);
    return 0;
}
```

Результат работы программы:

1

Дружественные функции могут быть перегружены.

Любой метод одного класса может быть дружественным по отношению к другому классу. В этом случае, для доступа к компонентам классов в качестве аргументов необходимо передать ссылки или указатели на объекты, для которых функция является дружественной.

Предположим, что в предыдущем примере функция *equal* является компонентом класса *A* и дружественной по отношению к классу *B*. Тогда описания этой функции будут выглядеть следующим образом:

B в классе *A*
...
bool equal(B &);
...
B в классе *B*
...
friend bool A::equal(B &);
...
Последняя запись сообщает компилятору, что функция equal() является дружественной к классу *B* и является компонентом класса *A*.

4.8. Перегрузка операций

Язык C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса, они выполняли заданные функции. Это даёт возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в C++, за исключением: .,::, sizeof, ?:.

Перегрузка операций осуществляется с помощью методов специального вида (*функций-операций*) и подчиняется следующим правилам:

1. при перезагрузке операций сохраняется количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
2. для стандартных типов данных переопределять операции нельзя;
3. функции-операции не могут иметь аргументов по умолчанию;
4. функции-операции наследуются;
5. функции-операции не могут определяться как *static*.

Функцию-операрацию можно определить тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией.

Если перегруженный оператор является членом какого-либо класса, то его протип задается следующим образом:
возвращаемый_тип <имя_класса> ::operator XX (аргументы);

Вместо записи *XX* непосредственно записывается оператор, который подлежит перегрузке.

4.8.1. Перегрузка бинарных операций

Рассмотрим механизмы перегрузки операций, которые являются членами классов. Если для какого-либо класса осуществляется перегрузка бинарного операции, то данный оператор будет иметь только один аргумент. Вторым аргументом по умолчанию является экземпляр этого класса. При этом нет необходимости передавать его отдельным аргументом. Доступ к нему осуществляется с помощью указателя *this*.

Определим класс, на примере которого будем рассматривать перегрузку всех операций. Пусть этот класс будет копией встроенного типа *int*, однако он реализован как пользовательский.

Пример:
class Number
{

```
        int n;
        public:
        Number(int n=0) {this->n=n;}
        print();
    };
    Number::print() {cout<<"n="<<n<<"\n";}
```

Перегрузим бинарные операторы «+» и «-» таким образом, чтобы можно было их использовать следующим образом:

```
Number N1(5), N2;
N2=N1+5;
N1=N2-4;
```

В этом случае у бинарных операций будет только один аргумент типа *int* (целое число), а второй – сам объект, по отношению к которому применяется перегруженная операция. Эти операции должны возвращать объекты, так как, после того как отработают перегруженные операции, результатом должен быть объект, который присваивается другому объекту. Для того чтобы не возникало никаких побочных эффектов при возврате объекта из перегруженной операции, необходимо наличие конструктора копирования. Запись обоих операций будет выглядеть следующим образом:

```
Number Number::operator + (const int arg)
{
    Number tmp;
    tmp.n=this->n+arg;
    return tmp;
}

Number Number::operator - (const int arg)
{
    Number tmp;
    tmp.n=this->n-arg;
    return tmp;
}
```

Функция main() будет такой:

```
void main()
{
   Number N1(5),N2;
   N2=N1+5;
   N2.print();
}
```

В классе необходимо описать прототип операторной функции:

```
Number operator + (const int);
```

Если бы перегруженные операции работали без временных объектов и возвращали бы ссылки на текущие объекты, то использование таких операций было бы некорректным. Так, в случае операции *N2=N1+5* сначала бы сработал оператор «+», который изменил бы объект *N1*, а затем изменённый объект был бы присвоен объекту *N2*, т.е. произошло бы изменение сразу же двух объектов. Использовать перегруженную операцию можно только в данной записи, если будет попытка использовать его следующим образом *N2=5+N1*, то компилятор такой записи не пропустит. При такой записи для компилятора появляются два явных аргумента – это целое число и экземпляр класса. Избежать этого можно, определив перегружаемую операцию как дружественную к данному классу. В этом случае передача второго аргумента (ссылка на объект) является оправданной.

...

```
friend Number operator - (const int, Number &);
...

Number operator - (const int arg, Number &obj)
{
	Number tmp;
	tmp.n=arg-obj.n;
	return tmp;
}
```

В языке С++ доступна запись *N2+=5*, упрощающая более длинную *N2=N2+5*. Используется это благодаря бинарному перегруженному оператору «+=», у которого имеется два аргумента – целое число и объект. При этом операция должна возвращать ссылку на тот объект, который используется в качестве аргумента, т.е. ссылку сам на себя. Реализация такой операции представлена в следующем фрагменте:

```
Number Number::operator += (const int arg)
{
	this->n+=arg;
	return (*this);
}
```

Были рассмотрены случаи, когда вторым аргументом бинарной операции является другой тип, в данном случае встроенный *int*. Однако можно перегружать бинарные операции таким образом, чтобы оба аргумента были объектами данного класса. Рассмотрим пример перегрузки бинарной операции сложения двух объектов.

```
Number Number::operator + (const Number &obj)
{
	Number tmp;
	tmp.n=this->n+obj.n;
	return tmp;
}
```

Теперь, после определения такого оператора возможна следующая запись:

```
Number N1(2), N2(8), N3;
N3=N1+N2;
```

4.8.2. Перегрузка логических операций и операций отношения

Все операции данных типов являются бинарными. Если они перегружаются как члены класса, то должны иметь один аргумент – объект или ссылку на него, а в качестве возвращаемого типа должен быть один из встроенных типов, который говорил бы о верности или неверности выражения. Это позволит при проверке с помощью операций отношения использовать длинные записи, такие как:

```
if((obj1>obj2)||(obj2<obj3)) ...
```

Рассмотрим перегрузку операции отношения «==» для класса *Number*.

```
int Number::operator == (const Number &obj) // перегрузка как члена класса
{
	if (this->n==obj.n) return 1;
	return 0;
}
```

Аналогичным образом можно перегрузить все остальные операции отношения и логические операции.

4.8.3. Перегрузка унарных операций

Все унарные операции имеют только один аргумент. При перегрузке унарных операций этим аргументом является экземпляр класса. В случае если унарная операция перегружается как член класса, то такой оператор не будет иметь аргументов. Рассмотрим пример перегрузки унарного знака «-» для класса *Number*.

```
Number Number::operator - ()
{
    n=-n;
    return (*this);
}
```

Следует обратить внимание на перегрузку таких унарных операций как «++» и «--». Многие компиляторы *C++* при перегрузке данных операций эквивалентно воспринимают как постфиксную, так и префиксную их запись. В *ANSI*-стандарте языка C++ можно различать постфиксную и префиксную записи. Осуществляется это путем использования второго фиктивного аргумента типа *int*. Если необходимо подчеркнуть, что операция будет использоваться в постфиксной записи, то необходимо наличие этого аргумента, который по умолчанию компилятором устанавливается в 0.

```
Number &Number::operator ++ (int)
{
    cout<<"Postfix\n";
    n++;
    return (*this);
}
Number &Number::operator ++ ()
{
    cout<<"Prefix\n";
    ++n;
    return (*this);
}
```

Раздел 5:
НАСЛЕДОВАНИЕ КЛАССОВ

5.1. Одиночное наследование

Наследование позволяет строить иерархию классов, переходя от более общих к более специальным. Приведем пример, пусть имеется один базовый класс, который описывает многоугольники. Пусть у данного класса определены некоторые свойства, такие как количество, множество координат вершин. На базе данного класса можно создать несколько производных классов, например четырёхугольник. Сам производный класс четырёхугольник может быть в свою очередь быть базовым для таких классов, как ромб или прямоугольник и т.д. Производный класс получается надстройкой над базовым классом, определяя в себе все, или почти все, свойства базового класса и имея свои собственные свойства. Рассмотрим, как можно определить наследование одного класса другим.

```
class имя_производного_класса : атрибут_наследования имя_базового класса
{
   // тело класса
};
```

Атрибут наследования определяется тремя известными ключевыми словами *private*, *public*, *protected*, которые использовались для доступа к внутренним компонентам классов. Дан-

ный атрибут при наследовании определяет видимость компонент базового класса внутри производного. Рассмотрим, как атрибут наследования влияет на видимость компонент базового класса. Это показано на рисунке 5.1.

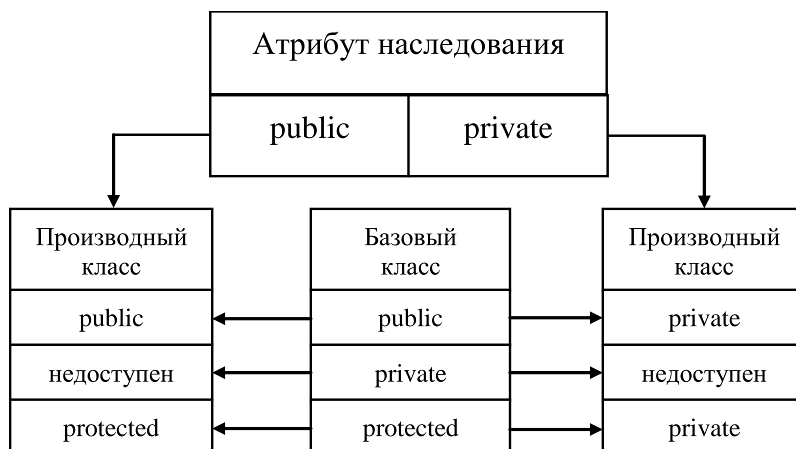

Рисунок 5.1 – Области видимости компонент класса

Модификатор *protected* используется тогда, когда необходимо, чтобы некоторые компоненты базового класса оставались закрытыми, но были бы доступны для производного класса. *Protected* эквивалентен атрибуту *private*, но существует одно искючение: защищённые компоненты базового класса доступны для членов всех производных классов этого класса. Как базовый класс, так и производные ему классы могут иметь конструкторы и деструкторы. Существует правило вызова конструкторов и деструкторов при одиночном наследовании. При создании экземпляра производного класса сначала выполняется конструктор базового, а затем производного класса. Деструкторы выполняются в обратном порядке. При создании объектов базового класса выполняются конструктор и деструктор только данного базового класса. Конструкторы производных классов и базового могут иметь аргументы. Если при создании экземпляра производного класса нет необходимости в передаче каких-либо аргументов конструктору базового класса, то данный конструктор должен быть описан либо как конструктор без параметров, либо иметь все аргументы по умолчанию. Производный класс может производить фиктивную передачу аргументов конструктору базового класса без их использования. Для того чтобы осуществить передачу аргументов для конструктора базового класса необходимо воспользоваться следующей записью:

конструктор_производного_класса(аргументы) : конструктор_базового_класса(аргументы)

Количество и тип аргументов для конструкторов производных классов и базового могут не совпадать. Рассмотрим пример использования всех вышеперечисленных свойств.

Пример.

```
#include <iostream.h>
class A
{
    protected:
    int n;
    public:
    A(int n=0) {this->n=n;cout<<"Constructor A\n";}
    ~A() {cout<<"Destructor A\n";}
};

class B:protected A
{
    public:
    print() {cout<<"n="<<n<<"\n";}
    B():A(5) {cout<<"Constructor B\n";}
```

```
    ~B() {cout<<"Destructor B\n";}
};

void main()
{
    B obj;
    obj.print();
}
```

Результаты работы программы:

Constructor A
Constructor B
n=5

Если конструктор базового класса имеет аргументы, то для передачи этих аргументов их должен содержать конструктор производного класса.

Пример.

```
#include <iostream.h>
class A
{
    int n;
    public:
    A(int n) {this->n=n; cout<<"Constructor A\n n="<<n<<"\n";}
    ~A() {cout<<"Destructor A\n";}
};

class B:public A
{
    int m;
    public:
    B(int m,int n):A(n)
{this->m=m;cout<<"Constructor B\n m="<<m<<"\n";}
    ~B() {cout<<"Destructor B\n";}
};

void main()
{
    B obj(1,2);
}
```

Результаты работы программы:

Constructor A\n n=2
Constructor B\n m=1
Destructor B
Destructor A

В случае если конструктор базового класса имеет все аргументы по умолчанию, то такой конструктор можно явно не задавать при описании конструктора производного класса.

Если в программе определен указатель на базовый класс, то его можно использовать для доступа к компонентам объектов базового класса, а также для доступа к компонентам базового класса, которые наследуются в производном классе. Если указатель указывает на экземпляр производного класса, то с помощью такого указателя можно получить доступ к тем компонентам, которые описаны только внутри производного класса. Если же описан указатель на производный класс, то при помощи такого указателя можно осуществлять доступ к открытым компонентам данного класса, в том числе и к компонентам базового класса, которые открыто наследуются в производном классе. Нельзя использовать указатель производно-

го класса для доступа к компонентам объектов базового класса. Рассмотрим пример, в котором используются указатели на производный и базовый классы.

Пример.

```
#include <iostream.h>
class A
{
	int n;
	public:
	A(int n=0)
	{this->n=n; cout<<"Constructor A\n n="<<n<<"\n";}
	~A() {cout<<"Destructor A\n";}
	print_A() {cout<<"n="<<n<<"\n";}
};

class B:public A
{
	int m;
	public:
	B(int m,int n):A(n)
	{this->m=m;cout<<"Constructor B\n m="<<m<<"\n";}
	~B() {cout<<"Destructor B\n";}
	print_B() {cout<<"m="<<m<<"\n";}
};

void main()
{
	A obj1(5);
	B obj2(1,2);
	A *ptr;		// указатель на базовый класс
// указатель ссылается на объект базового класса
	ptr=&obj1;
	ptr->print_A();
	ptr=&obj2;
// вызов метода объекта производного класса
	ptr->print_A();
	B *ptr2;
	ptr=&obj2;
	ptr2->print_A();
	ptr2->print_B();
}
```

Результаты работы программы:

```
Constructor A\n n=5
Constructor A\n n=2
Constructor B\n m=1
n=5
n=2
n=2
m=1
Destructor B
Destructor A
Destructor A
```

5.2. Виртуальные функции

Пример.

```
#include <iostream.h>
class A
{
	int n;
	public:
	A(int n ){this->n=n;}
	void print() {cout<<"Base n="<<n<<"\n";}
};

class B: public A
{
	char c;
	public:
	B(char c):A(1) {this->c=c;}
	void print() {cout<<"Derived c="<<c<<"\n";}
};

void main()
{
	A obj1(1); B obj2('W');
	obj1.print();
	obj2.print();
}
```

Результаты работы программы:

Base n=1
Derived c=W

В производном и базовом классах определены функции, отвечающие за вывод внутренних компонент. Вызов данных функций происходит с явным указанием экземпляров этих классов. Рассмотрим этот же пример, но вызов функций *print()* будет осуществляться через указатель на базовый класс.

Пример.

```
void main()
{
	A obj1(1); B obj2('W');
	A *ptr;
	ptr=&obj1;
	ptr->print();
	ptr=&obj2;
	ptr->print();
}
```

Результаты работы программы:

Base n=1
Base n=1

Указатель, объявленный на базовый класс, может быть использован в качестве указателя на любой производный класс. В примере, когда указатель на базовый класс указывает на экземпляр базового класса, происходит вызов функции *print()* базового класса. Указатель на базовый класс может быть использован только для доступа к компонентам, которые описаны в базовом классе. Для доступа к компонентам из производных классов необходимо исполь-

зовать указатель на производный класс. В случае использования указателя на базовый класс эта задача решается путем использования виртуальных функций (методов).

Виртуальный метод – это метод, который, будучи описан в потомках, замещает собой соответствующий метод везде, даже в методах, описанных для предка, если он вызывается для потомка.

Виртуальная функция задается точно также как и обычная, только в начале определения такой функции ставится ключевое слово *virtual*. Виртуальная функция объявляется внутри базового класса. Если виртуальная функция переопределяется в производных классах, то она автоматически в них становится виртуальной, и в этом случае нет необходимости использовать ключевое слово *virtual*.

С понятием виртуальных функций тесно связано понятие полиморфизма. *Полиморфизм* – это свойство родственных объектов (объектов, классы которых являются производными от одного родителя) вести себя по-разному в зависимости от ситуации, возникшей в момент выполнения программы. На рассмотренном примере это будет выглядеть следующим образом: если используется указатель на базовый класс, в котором определена виртуальная функция, и эта функция переопределена в производных классах, то при адресации указателя базового класса на экземпляры производных, будет вызываться функция, соответствующая каждому производному классу. Виртуальные функции немного отличаются от перегруженных функций. Виртуальная функция должна полностью повторяться в производных классах, т.е. список аргументов и возвращаемое значение обязательно должны совпадать, иначе такая функция будет считаться просто перегруженной функцией.

5.3. Чисто виртуальные функции

Часто возникают ситуации, при которых виртуальные функции, определенные в базовых классах не используются, а иногда и не содержат никаких действий, а являются лишь прототипами (шаблонами) для конкретных реализаций виртуальных функций в производных классах. Для того чтобы подчеркнуть, что в программе не предусматривается вызов виртуальной функции для базового класса, используют чисто виртуальные функции. Для их определения используют следующую запись:

virtual возвращаемый_тип имя(аргументы) =0;

Если в базовом классе определен прототип чисто виртуальной функции, то она должна быть обязательно определена для всех производных классов. Если в базовом классе определена хотя бы одна чисто виртуальная функция, то такой класс называется *абстрактным базовым классом*.

5.4. Ранее и позднее связывание

Под процессами *раннего связывания* понимают те процессы, которые могут быть предопределены на стадии компиляции. Например, при использовании вызовов обычных функций. Компилятор заранее знает адрес вызываемой функции, и этот адрес помещает в место вызовова этой функции.

Иначе дело обстоит при вызове виртуальных функций – *позднее связывание*. Процессы, относящиеся к позднему связыванию, определяются на стадии выполнения программы. Например, компилятор заранее может не предугадать вызов виртуальной функции для конкретных экземпляров производных классов. Адрес виртуальной функции известен только в момент выполнения программы. Когда происходит вызов виртуальной функции, её адрес берётся из таблицы виртуальных методов своего класса. Нужная версия виртуальной функции выбирается на стадии выполнения программы. Такой процесс называется *поздним связыванием*.

5.5. Виртуальные деструкторы

Виртуальные деструкторы необходимы в случаях использования указателей на базовые классы, когда производится выделение динамической памяти под объекты производных классов. Рассмотрим на примере программы, как бы вызывались деструкторы производных и базового классов в случае виртуальных и не виртуальных деструкторов.

Пример.

```
#include <iostream.h>
class Something
{
	public:
	Something(char *what) {cout<<"This is "<<what<<endl;}
	void virtual print()=0;
	virtual ~Something() {cout<<"Base destructor\n";}
};

class Number:public Something
{
	int n;
	public:
	Number(int n, char *id="number"):Something(id)
	{this->n=n;}
	void print() {cout<<"n="<<n<<endl;}
	~Number() {cout<<"Number destructor\n";}
};

class Strng:public Something
{
	char *str;
	public:
	Strng(char *str, char *id="string"):Something(id)
	{this->str=str;}
	void print() {cout<<"str="<<str<<endl;}
	~Strng() {cout<<"String destructor\n";}
};

int main()
{
  // описываются указатели на базовый класс
	Something *ptr1,*ptr2;
  // указатель ссылается на объект производного класса
	ptr1=new Number(5);
	ptr2=new Strng ("Hello!");
	ptr1->print();
	ptr2->print();
	delete ptr1;
	delete ptr2;
	return 0;
}
```

Результаты работы программы:

```
This is number
This is string
n=5
str=Hello!
```

Number destructor
Base destructor
String destructor
Base destructor

В случае не виртуальных деструкторов при удалении объектов производных классов вызывался бы деструктор только базового класса. Если деструктор базового класса объявлен как виртуальный, все деструкторы производных классов автоматически становятся виртуальными. Если в производном классе не описан деструктор, то при удалении объекта будет вызван деструктор базового класса. Если деструктор описан, то в начале произойдет вызов деструктора производного, а затем базового классов.

5.6. Множественное наследование

В языке С++ существует возможность наследовать более одного класса в качестве базового. Такой механизм наследования называется *множественным наследованием*. При множественном наследовании объекты производных классов могут использовать данные и методы нескольких классов, а также в свою очередь быть базовыми классами для других. Рассмотрим, как задаются производные классы от более чем одного базового класса.

class derived: атрибут_наследования base1,...,атрибут_наследования baseN
{...};

Атрибуты наследования и их смысл тот же, что и при одиночном наследовании. Данная последовательность базовых классов определяет порядок вызова конструкторов базовых классов. При создании экземпляра производного класса в первую очередь вызовется конструктор базового класса, имя которого определено первым в списке наследуемых классов. Далее по порядку. Последним вызовется конструктор производного класса. При уничтожении объекта производного класса деструкторы вызываются в обратном порядке вызовам конструкторов. При описании конструктора производного класса задаются те конструкторы базовых классов, которые имеют аргументы. Если базовый класс не имеет аргументов или все аргументы такого конструктора используются по умолчанию, то имя такого конструктора можно не задавать в описании производного класса.

конструктор_производного_класса (аргументы) : base1(аргументы), ... ,baseN(аргументы)
{...}

Список конструкторов базовых классов не влияет на порядок их вызова. Он определяется только списком имен базовых классов в начале определения производного класса.

5.7. Виртуальные базовые классы

В языке С++ запрещено непосредственно передавать базовый класс в производный более одного раза, т.е. имя класса в списке базовых классов не может повторяться.

```
class A {...};
class B: public A, protected A {...};
```

Однако могут возникать ситуации, когда один производный класс косвенно может наследовать один и тот же базовый класс через другие базовые классы. Например,

```
class A
{
  public: int n;
  ...
};
class B:public A
```

```
{...};
class C : public A, public B
{...};
```

При данной схеме наследования классов получается следующая структура объектов (см. рисунок 5.2).

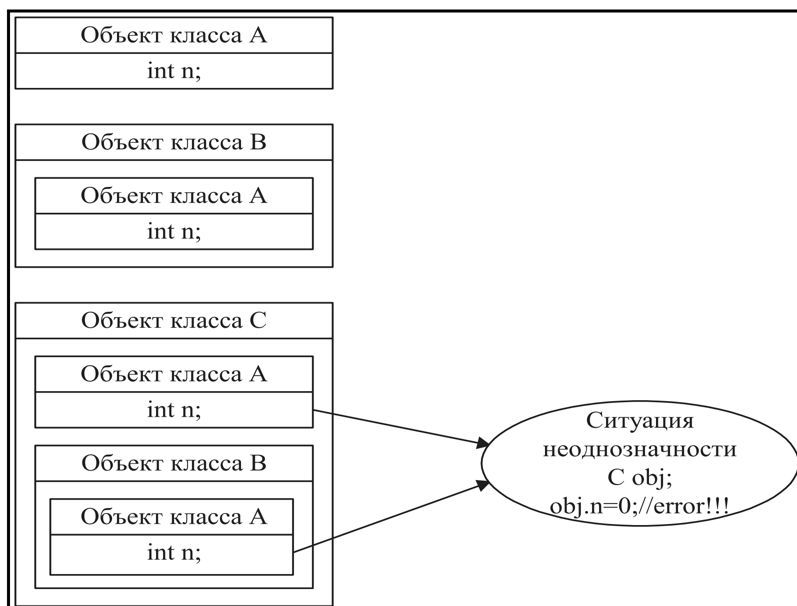

Рисунок 5.2 – Схема наследования классов

Такое наследование вполне возможно и компилятор не выдаст сообщение об ошибке, а только предупредит, что наследуемый класс *A* также находиться и в наследуемом классе *B*. Однако в данном примере ошибка возникнет на стадии компиляции. Компилятор не различит, какую именно переменную *int n* базового класса необходимо изменить – переменную, которая непосредственно является компонентом класса *A* или переменную, которая доступна из наследуемого класса *B*. Решить такого рода проблему в C++ можно путём использования виртуальных базовых классов. Если один базовый класс косвенно наследуется в одном производном классе и наследуется с атрибутом *virtual*, то в экземпляр производного класса будет помещена только одна копия базового класса. Базовый класс объявляется как виртуальный только при наследовании в списке базовых классов с указанием спецификатора *virtual*:

```
class A {...};
class B: virtual public A {...};
class C: virtual public A, public B {...};
```

Если базовый класс наследуется производным как виртуальный, то его компоненты доступны в производном классе. Отличие между обычным и виртуальным наследованием заключается в том, что когда класс наследует базовый класс более одного раза, то он будет содержать только одно вхождение базового класса.

Пример:

```
#include <iostream.h>
class A
{
   public:
   int a;
   A(int a) {this->a= a; cout<<"Constructor A\n";}
   ~A ( ) {cout<< "Destructor A\n";}
   void print( ) {cout<<"a= "<<a<<endl;}
};
class B
```

```
{
    public:
    int b;
    B(int b) {this->b= b; cout<<"Constructor B\n";}
    ~B( ){cout<<"Destructor B\n";}
    void print ( ) {cout<<"b= "<<b<<endl;}
};
class C : virtual public A
{
    public:
    int c;
    C(int c) : A(c) {this->c= c; cout<<"Constructor C\n";}
    ~C( ) {cout<<Destructor C\n";}
    void print( ) {cout<<"c="<<c<<endl;}
};
class D : public B, virtual public C, virtual public A
{
    public:
    int d;
    D(int d) : A(d+1), B(d+2), C(d+3)
    {this->d= d; cout<<"Constructor D\n";}
    ~D( ) {cout<<"Destructor D\n";}
    void print( )
    {
        cout<<"d= "<<d<<endl;
        A::print( );
        B::print( );
        C::print( );
    }
};
int main ( )
{
    D obj(1);
    obj.print( );
    return 0;
}
```

Результаты работы программы:

```
Constructor A
Constructor C
Constructor B
Constructor D
d= 1
a= 2
b= 3
c= 4
Destructor D
Destructor B
Destructor C
Destructor A
```

При виртуальном наследовании базовых классов в первую очередь вызываются конструкторы виртуальных базовых классов в порядке наследования. В примере первым будет вызван конструктор виртуального класса *A*, затем конструктор виртуальног класса *C*, затем конструктор базового класса *B* и последним будет вызван конструктор производного класса *D*. Деструкторы вызываются в обратном порядке. В случае, если базовый класс *C* наследо-

вался бы как не виртуальный, порядок вызова конструкторов был бы следующим: коструктор *A*, конструктор класса *B*, конструктор класса *C*, конструктор класса *D*. Если базовый класс *A* наследовался бы как не виртуальный, то это привело бы к ошибке на стадии компиляции.

Рассмотрим пример программы, которая использует виртуальные функции. Предположим, что необходимо создать список элементов разных типов, например, *int* и *char* *. Любой элемент вне зависимости от его типа может принадлежать какому-то множеству. Необходимо также создать функции, которые выводили бы значения элементов, принадлежащих заданному множеству, а также удаляли бы из списка элементы заданного множества. Принадлежность элемента тому или иному множеству задается при помощи идентификатора множества.

Пример.

```
#include <iostream.h>
class Base            //базовый класс
{
protected:
      Base *next;
      Base *prev;
      int num;      //идентификатор подмножества
      static Base *Begin;
      static int NumObj;
public:
      Base(int);
      virtual void print()=0;
      static void printall(int);
      static void del(int);
};

Base *Base::Begin=NULL;
int Base::NumObj=0;
Base::Base(int k)      //при создании объекта класса Base
                       //он сам себя помещает в стек
{
num=k;
      NumObj++;
      next=Begin;
      if (Begin!=NULL) Begin->prev=this;
      Begin=this;
      Begin->prev=NULL;
}
void Base::printall(int k)      //печатает значения всех элементов
                     //с заданным идентификатором подмножества
{
      Base *p=Begin;
      for (int i=0; i<NumObj; p=p->next, i++)
            if (p->num==k) p->print();
}

void Base::del(int k)          //удаляет все элементы с заданным
                               //идентификатором подмножества
{
      int i=NumObj;
      for (Base *tmp=Begin; i>0; tmp=tmp->next,i--)
if (tmp->num==k)
            {
```

```
			if (tmp==Begin) Begin=tmp->next;
				else tmp->prev->next= tmp->next;
			delete tmp;
		}
}

class Set			//класс для работы с подмножествами
{
	static int Lastident;
	int ident;
public:
	Set() {ident=++Lastident;}
	~Set() {Base::del(ident);}
	int getident() {return ident;}
	friend ostream &operator<<(ostream &,Set1&);
};

int Set::Lastident=0;
ostream &operator<<(ostream &os, Set1 &s)
{
	Base::printall(s.ident);
	return os<<endl;
}
class Integer:public Base  //производный класс с типом элементов int
{
	int i;
public:
	Integer(int i, int num):Base(num) {this->i=i;}
	void print() {cout<<i<<' ';}
	int& get() {return i;}
};

class String:public Base //производный класс с типом элементов *char
{
	char *str;
public:
	String(char *str,int num):Base(num) {this->str=str;}
	void print() {cout<<str<<' ';}
};
int main()
{
	Set s1;
	int id1=s1.getident(),id2;
	Integer t1(1,id1),t2(5,id1);
	String t3("string1",id1),t4("string2",id1);
	{
		Set s2;
		id2=s2.getident();
		String t5("Hello",id2);
		Integer t6(12,id2);
		cout<<s1<<s2;
	}
	Base::printall(id1);
	return 0;
}
```

Результаты работы программы:
string2 string1 5 1
12 Hello
string2 string1 5 1

Раздел 6: ШАБЛОНЫ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Часть лекции проводится в интерактивной форме с разбором конкретных ситуаций (0,5 час.)

6.1. Шаблоны функций

Шаблоны функций являются логическим продолжением механизма перегрузки функций. Обычно перегрузку функций осуществляют для различных типов данных, причем все перегруженные функции осуществляют идентичные действия. Так, например, для сортировки и вывода на экран одномерных массивов данных различных типов, приходится реализовывать различные перегруженные функции, содержимое которых порой отличается только используемыми типами данных. Для возможности сортировки массивов *int* и *char** приходится реализовывать две перегруженные функции *sort(int,int)* и *sort(char*,int)*. Механизм шаблонов функций в языке C++ позволяет более легким и гибким путём решить эту проблему. Достаточно описать один шаблон функции сортировки и вывода на экран, и компилятор сгенерирует код такой функции в соответствии с используемым типом данных при вызове такой функции.

Пример.

```
#include <iostream.h>
void print(int array[],int d)
{
	for (int i=0; i<d; i++)
	cout<<array[i];
}
void print(char array[],int d)
{
	for (int i=0; i<d; i++)
	cout<<array[i];
}
int main()
{
	char A1[6]="HELLO!";
	int A2[5]={1,2,3,4,5};
	print(A1,6);
	cout<<"\n";
	print(A2,5);
	return 0;
}
```

Результаты работы программы:
HELLO!
12345

В данном примере реализованы две идентичные функции, которые выполняют одни и те же операции и различаются только типом одного аргумента. Рассмотрим, как это можно реализовать с помощью шаблонов. В общем случае шаблон функции задается следующим образом:

template <class тип_параметра> возвращаемый_тип

```
имя_функции(аргументы)
{
...
}
```

Определение каждого типа параметра начинается с ключевого слова *class*. Если используется функция более одного типа, то они разделяются запятой. При компиляции компилятор вместо типа параметра подставляет необходимый тип, который был задан при вызове функции. Определим шаблон функции *print()*. В данном случае в описании шаблона будет присутствовать только один тип параметра – тип одномерного массива.

Пример.

```
#include <iostream.h>
template <class Array_Type> void print(Array_Type array[],int d)
{
	for (int i=0; i<d; i++)
	cout<<array[i];
}
int main()
{
	char A1[6]="HELLO!";
	int A2[5]={1,2,3,4,5};
	print(A1,6);
	cout<<"\n";
	print(A2,5);
	cout<<"\n";
	return 0;
}
```

Результаты работы программы:

```
HELLO!
12345
```

В качестве типа параметра могут быть заданы не только встроенные типы, а также типы, определенные пользователем, в том числе и классы. Рассмотрим пример шаблона функции, аргументом которого является класс.

Пример.

```
#include <iostream.h>
class Integer1
{
	int n;
	public:
	Integer1(int n=0) {this->n=n;}
	int get() {return n;}
};

class String1
{
	char *str;
	public:
	String1(char *str="Nothing") {this->str=str;}
	char* get() {return str;}
};

template <class Class_Type> void print_it(Class_Type Obj)
{
	cout<<Obj.get()<<endl;
```

```
}
int main()
{
	Integer1 num(12);
	String1 buf;
	print_it(num);
	print_it(buf);
	return 0;
}
```

Результаты работы программы:

12
Nothing

Шаблоны функций могут быть перегружены. Если в шаблоне описаны несколько типов параметров, то они должны быть все обязательно описаны в списке аргументов функции.

```
template <class Class_Type1,class Class_Type2>
void print_it(Class_Type1 Obj1, Class_Type2 Obj2)
{
	cout<<"Obj1="<<Obj1.get()<<endl;
	cout<<"Obj2="<<Obj2.get()<<endl;
}
```

Последний пример шаблона функции показывает, что при его использовании порядок аргументов не имеет значения, т.е. можно использовать этот шаблон двумя способами:

- *print_it(num,buf)*;
- *print_it(buf,num)*.

6.2. Шаблоны классов

Помимо шаблонов функций в C++ можно использовать и шаблоны классов. Идея использования аналогична. Описывается шаблон класса, который осуществляет идентичные действия с различными типами данных. Используемый тип данных определяется на стадии компиляции. Общая форма описания шаблона класса:

```
template <class Class_Type1,...,class Class_type2> class имя_класса
{
...
};
```

Экземпляры шаблонов классов описываются так:

имя_класса <тип> объект;

Все функции-компоненты шаблонов классов автоматически становятся шаблонами, значит при их описании необязательно задавать ключевое слово *template*. Статические переменные шаблонов классов необходимо инициализировать для каждого используемого типа данных.

Пример.

```
#include <iostream.h>
template <class Type> class Class_type
{
	static int Nobj;
	Type element;
	public:
	Class_type(Type element) {this->element=element; Nobj++;}
	Type &get() {return element;}
	friend ostream &operator<<(ostream &os,Class_type &Obj)
```

```
        {
            return
                os<<"Element"<<Nobj<<"="<<Obj.element<<endl;
        }
};

int Class_type<int>::Nobj=0;
int Class_type<char*>::Nobj=0;

int main()
{
        Class_type<int> Num(5);
        Class_type<char*> Str("Message");
        cout<<Num<<Str;
        Num.get()++;
        Str.get()="Hello";
        cout<<Num<<Str;
        Class_type<int> Obj(12);
        cout<<Obj;
        return 0;
}
```

Результаты работы программы:

```
Element1=5
Element1=Message
Element1=6
Element1=Hello
Element2=12
```

Шаблоны классов, как и классы, поддерживают механизм наследования. Все основные идеи наследования при этом остаются неизменными, что позволяет построить иерархическую структуру шаблонов, аналогичную иерархии классов.

6.3. Библиотека стандартных шаблонов

Стандартная библиотека С++ предоставляет большой набор шаблонов для различных способов организации хранения и обработки данных.

При рассмотрении вопроса, связанного с библиотекой стандартных шаблонов (*Standard Templates Library – STL*) необходимо выделить некоторые ключевые понятия: контейнеры, итераторы и алгоритмы. Под *контейнером* понимают объекты, предназначенные для хранения объектов других (произвольных) типов. Тип контейнера может быть любым. В *STL* описаны следующие типы контейнеров:

- *bitset* – множество битов;
- *deque* – двунаправленная очередь;
- *list* – линейный список;
- *map* – ассоциативный список (ключ -> одно значение);
- *multimap* – ассоциативный список (ключ -> несколько значений);
- *multiset* – множество;
- *priority_queue* – очередь с приоритетом;
- *queue* – очередь;
- *set* – множество уникальных элементов;
- *stack* – стек;
- *vector* – вектор (динамически изменяемый стек).

Ассоциативный контейнер отличается от остальных типов тем, что значение или значения, хранящиеся в контейнере, могут быть доступны только по определенному ключу (функ-

ционирование аналогично ассоциативному запоминающему устройству). Каждому контейнеру принадлежит набор уникальных функций, отвечающих за манипулирование с содержимым контейнера. В каждом контейнере определены функции, отвечающие за добавление элемента в контейнер и удаление элемента из контейнера. *Итераторы* – указатели на контейнер, используются для доступа к элементам контейнера. Выделяют итераторы следующих 5 типов:

- *random access* (*RandIter*) – считывание и запись элементов с произвольным доступом;
- *bidirectional* (*BiIter*) – считывание и запись элементов, два направления прохода контейнера;
- *forward* (*ForIter*) – считывание и запись элементов, однонаправленный проход контейнера;
- *input* (*InIter*) – считывание элементов, однонаправленный проход контейнера;
- *output* (*OutIter*) – запись элементов, однонаправленный проход контейнера.

Использование итераторов аналогично использованию обычных указателей на типы, т.е. итераторы, как и указатели, можно инкрементировать и т.д. *Алгоритмы* осуществляют операции над элементами контейнеров (сортировка, поиск, замена и т.д.). Для каждого контейнера определен так называемый *allocator* (распределитель памяти), который управляет процессом выделения памяти под конкретный контейнер. По умолчанию распределитель памяти для каждого контейнера является экземпляром класса *allocator*. Можно определить и свой собственный распределитель памяти. Рассмотрим более подробно механизмы использования контейнеров на конкретном примере – контейнер *vector*.

Контейнер vector.

Контейнер *vector* определяет динамически изменяемый стек. Прототип шаблона класса vector выглядит следующим образом:

template <class Type, class Allocator = allocator <Type>> class vector,

где

Type – тип хранящихся данных в контейнере;

Allocator – распределитель памяти (по умолчанию – стандартного типа *allocator*).

Класс *vector* имеет несколько перегруженных конструкторов:

- конструктор, создающий пустой вектор

  vector (const Allocator &alloc = Allocator());

- конструктор, создающий вектор с инициализацией каждого элемента

  vector (size_type number, const Type &what = Type(), const Allocator &alloc = Allocator ());

- конструктор, создающий вектор из объектов одинакового типа

  vector (const vector <Type, Allocator> &object);

- конструктор, создающий вектор по диапазону

  template <class InIter> vector (InInter Begin, InInter End, const Allocator &alloc = Allocator ()).

Если элементом вектора является экземпляр класса, то в описании такого класса обязательно должен присутствовать конструктор по умолчанию, и перегружены необходимые операторы. Для встроенный типов все операторы перегружены по умолчанию. В самом шаблоне *vector* перегружены операции сравнения <,>,<=,>=, !=, ==. Рассмотрим подробнее некоторые функции-члены шаблона класса *vector*. Функциями-членами шаблона класса *vector* являются следующие:

- *template <class InInter> void assign (InInter Begin, InInter End)* – присваивает вектору последовтельность, определенную итераторами *Begin* и *End*;
- *reference at(size_type i)* – возвращает ссылку на i-ый элемент контейнера;
- *reference back()* – возвращает ссылку на последний эле-

мент;

- *reference front()* – возвращает ссылку на первый элемент;
- *iterator begin()* – возвращает итератор первого элемента;
- *iterator end()* – возвращает итератор последнего элемента;
- *void clear()* – удаляет все элементы вектора;
- *iterator erase(iterator i)* – удаляет элемент, на который указывает итератор i;
- *reference operator [] (size_type i) const* – возвращает ссылку на i-ый элемент;
- *void pop_back()* – удаляет последний элемент вектора;
- *void push_back(const Type &value)* – добавляет в конец вектора элемент;
- *size_type size() const* – возвращает текущее количество элементов.

В описании шаблона *vector* присутствуют переопределенные типы. Ниже приведены расшифровки некоторых из них:

- *reference* – ссылка на элемент вектора;
- *const_reference* – константная ссылка на элемент вектора;
- *iterator* – итератор;
- *const_iterator* – константный итератор;
- *value_type* – тип элемента вектора;
- *allocator_type* – тип распределителя памяти (по умолчанию *allocator*);
- *key_type* – тип ключа для ассоциативных контейнеров.

Рассмотрим пример использования стандартного шаблона vector.

Пример 1.

```
#include <iostream.h>
#include <vector.h>        //подключение стандартной библиотеки
                           //vector
void main()
{
 vector<int> m;                 //пустой вектор элементов типа int
 vector<int>::iterator ptr;  //итератор
 int buf=0;
 while (cout<<"Input:",cin>>buf,buf!=0) m.push_back(buf); //добавление элементов
 cout<<"Vector size:"<<m.size()<<endl; //вывод количества элементов
 cout<<"Vector value:";
 for(int i=0; i<m.size(); cout<<m[i++]); cout<<endl; //вывод самих элементов
 cout<<"Vector value, part#2:";
 ptr=m.begin();             //итератор указывает на начало вектора
 while (ptr!=m.end()) cout<<*ptr++; cout<<endl; //вывод при помощи итератора
 cout<<"Delete all elements\n";
 m.erase(m.begin(),m.end());        //удаление всех элементов
 cout<<"Vector size:"<<m.size()<<endl; //вывод количества элементов
}
```

Результаты работы программы:

```
Input:1
Input:3
Input:6
Input:9
Input:0
Vector size:4
Vector value:1369
Vector value, part#2:1369
```

Delete all elements
Vector size:0

Теперь рассмотрим пример использования контейнера *vector* для хранения экземпляров классов.

Пример 2.

```
#include <iostream.h>
#include <vector.h>
class Number
{
 int n;
 public:
 Number() {n=0;}
 Number(int n) {this->n=n;}
 friend ostream &operator << (ostream&,Number&);
};

ostream &operator << (ostream &os, Number &obj)
{
 return (os<<" "<<hex<<obj.n);
}

void main()
{
 vector<Number> N(5,0xff);
 vector<Number>::iterator ptr;
 cout<<"Vector size:"<<N.size()<<endl;
 ptr=N.begin();
 for(int i=0; i<5; cout<<N[i++]);
 cout<<"\nVector value:";
 while (ptr!=N.end()) cout<<*ptr++;
 cout<<endl;
}
```

Результаты работы программы:

Vector size:5
 ff ff ff ff ff
Vector value:ff ff ff ff ff

Каждый контейнер содержит алгоритмы, необходимые для работы с элементами. Все алгоритмы представляют собой шаблоны функций. Для использования стандартных алгоритмов *STL* необходимо подключить библиотеку *<algorithm.h>*. Описанные в ней алгоритмы применимы к любому типу контейнеров. Рассмотрим пример использования стандартных алгоритмов для первого примера.

Пример 3.

```
#include <iostream.h>
#include <vector.h>
#include <algorithm> //<algo.h>
void main()
{
vector<int> m;
vector<int>::iterator ptr;
int buf=0;
while (cout<<"Input:",cin>>buf,buf!=0) m.push_back(buf);
cout<<"Vector size:"<<m.size()<<endl;
cout<<"Vector value:";
```

```
for (int i=0 ;i<m.size(); cout<<m[i++]); cout<<endl;
random_shuffle(m.begin(),m.end()); //случайное перемешивание
                                   //элементов определенного диапазона
cout<<"Vector value, part#2:";
ptr=m.begin();
while (ptr!=m.end()) cout<<*ptr++; cout<<endl;
ptr=max_element(m.begin(),m.end());//возвращает итератор максимального элемента в диапазоне
cout<<"Maximal element:"<<*ptr<<endl;
cout<<"Delete all elements\n";
m.erase(m.begin(),m.end());
cout<<"Vector size:"<<m.size()<<endl;
}
```

Результаты работы программы:

```
Input:1
Input:3
Input:6
Input:9
Vector size:4
Vector value:1369
Vector value, part#2:3196
Delete all elements
Vector size:0
```

6.4. Обработка исключительных ситуаций

Одним из последних нововведений в стандарте языка C++ является механизм обработки ошибок или обработки исключительных ситуаций. К исключительным ситуациям относят такие события, происходящие во время выполнения программы, которые могут привести к ее сбою, например: деление на 0, выделение динамической памяти размером более допустимого, выход индексов массивов за дозволенные пределы, и т.д. Обработка исключительных ситуаций введена только в последние версии стандарта языка C++ (GCC, Watcom, и т.д.). Механизм обработки исключительных ситуаций реализован с помощью трех ключевых слов в C++: *try* (попытаться), *catch* (поймать), *throw* (бросить). В общих чертах этот механизм работает так. Функция пытается выполнить фрагмент кода. Если в коде содержится ошибка, функция бросает (генерирует) сообщение об ошибке, которое должно поймать (перехватить) вызывающая функция. *Try* и *catch* являются блоками. В блоке *try* располагаются те операторы, которые необходимо проверить на исключительные ситуации. Если исключительная ситуация имеет место быть в блоке *try*, то сообщение о ней генерируется оператором *throw*. Перехватывается и обрабатывается сообщение об исключительной ситуации блоком *catch*, в котором располагаются соответствующие операторы – это чаще всего сообщения о возникновении исключительных ситуациях.

Блок *try* должен содержать ту часть программы, в которой необходимо отслеживать ошибки. Это могут быть как несколько операторов внутри одной функции, так и все операторы функции *main()*. Оператор *throw* должен выполняться либо внутри блока *try*, либо в любой функции, вызов которой происходит внутри блока *try*. Любая исключительная ситуация должна перехватываться блоком *catch*, который располагается непосредственно за блоком *try*. Для одного блока *try* может существовать несколько блоков *catch*. В этом случае выполняется тот блок *catch*, аргумент которого соответствует типу сгенерированного сообщения об исключительной ситуации с помощью оператора *throw*. Общая форма записи блоков *try* и *catch* выглядит следующим образом:

```
try
{
...
```

```
throw ...;
f(); // throw
}
catch(type1 arg) {...}
catch(typeN arg) {...}
```

Тип сообщения об ошибке может быть любым, в том числе и пользовательским. Записывается оператор *throw* следующим образом:

```
throw сообщение;
throw 1;
throw "Error!";
throw 'E';
```

Если в программе нет блока *catch*, аргумент которого соответствует сообщению об ошибке, то это приводит к ненормальному завершению программы (*abnormal program termination*).

Рассмотрим несколько примеров, чтобы понять, как работают операторы.

Пример 1.

```
#include <iostream.h>
void main()
{
  cout<<"Begin \n";
try
 {
    cout<<"Block try\n";
    throw 1;
    cout<<"Not\n";
 }
catch(char *msg)
  {
    cout<<msg<<endl;
  }
catch(int n)
  {
    cout<<"Error number "<<n<<endl;
  }
cout<<"End \n";
}
```

Результаты работы программы:

```
Begin
Block try
Error number 1
End
```

Из примера видно, было перехвачено сообщение типа *int*, а блок *catch* с типом *char** был проигнорирован.

Исключительная ситуация может быть вызвана из невходящего в блок *try* оператора, если он входит в функцию, которая вызывается из блока *try*.

Пример 2.

```
#include <iostream.h>
void TEST(int val)
{
   cout<<"Inside TEST "<<val<<"\n";
   if (val) throw val;
}
```

```
void main()
{
   try
   {
         TEST(0);
         TEST(1);
         TEST(2);
   }
   catch(int i)
   {
         cout<<"Error !!\n";
         cout<<i<<"\n";
   }
}
```

Результаты работы программы:

Inside TEST 0
Inside TEST 1
Error !!
1

Блок *try* можно располагать внутри функции. В этом случае при каждом входе в функцию обработчик исключительной ситуации устанавливается снова.

Пример 3.

```
void TEST(int val)
{
   try
   {
         if (val) throw val;
   }
   catch(int i)
   {
         cout<<"Error  #"<<i<<"\n";
   }
}
void main()
{
   TEST(1);
   TEST(0);
   TEST(2);
}
```

Результаты работы программы:
Error #1
Error #2

В некоторых случаях необходимо перехватывать все исключительные ситуации независимо от их типа. Для этого используют следующую форму блока *cath*:

```
cath(...)
   {
         //обработка всех исключительных ситуаций
   }
```

Многоточие соответствует любому типу данных.

Для функции, вызываемой из блока *try*, можно ограничить число типов исключительных ситуаций, которые способна сгенерировать функция. Можно даже запрещать генерировать некоторые типы исключительных ситуаций.

возвращаемый_тип имя_функции (аргументы) throw(список_типов)
```
{

}
```

Генерация любого другого типа исключительной ситуации приведет к аварийному завершению программы. Если необходимо, чтобы функция не генерировала никаких исключительных ситуаций, то можно список типов сделать пустым.

Оператор *cath(...)* удобно использовать в качестве последнего оператора в группе операторов *catch*. Он будет перехватывать все остальное. Путём перехвата всех исключительных ситуаций предотвращается аварийное завершение программы из-за необработанной исключительной ситуации.

Если генерируется исключительная ситуация, для которой нет соответствующего оператора *catch*, то произойдет вызов функции *terminate()*, которая в свою очередь вызовет функцию *abort()*.

Если необходимо повторно сгенерировать исключительную ситуацию из процедуры обработки исключительной ситуации (*catch*), то можно использовать оператор *throw* без параметров. Это приведет к тому, что текущая исключительная ситуация передастся внешней последовательности *try/catch*.

Пример 4.
```
void TEST()
{
   try
   {
         throw "Error!\n";
   }
   catch(char*)
   {
         cout<<"Intercept error inside TEST\n";
         throw;
   }
}
void main()
{
   try
   {TEST();}
   catch(char*)
   {cout<<"Intercept error inside main\n";}
}
```
Результаты работы программы:
Intercept error inside TEST
Intercept error inside main

4.3. Лабораторные работы

№ п/п	***Номер раздела дисциплины***	***Наименование лабораторной работы***	***Объем (час.)***	***Вид занятия в интерактивной, активной, инновационной формах, (час.)***
1	**1.**	Обработка одномерных массивов и данных символьного и строкового типов	**2**	-
2	**2.**	Использование прототипа функции. Перегрузка фукции	**1**	-

3	**3.**	Работа с файлами	**1**	разбор конкретных ситуаций (1 час)
4	**4.**	Работа со структурами	**2**	тренинги в малой группе (1 час)
5	**5.**	Работа с классами	**2**	разбор конкретных ситуаций (1 час)
6	**6.**	Перегрузка операций	**2**	-
		ИТОГО	**10**	**3**

4.4. Практические занятия

Учебным планом не предусмотрено.

4.5. Контрольные мероприятия: контрольная работа

Цель: овладеть навыками программирования классических структур данных, заполнения структур данными из файла, обработки структрур данных.

Структура: каждое индивидуальное задание предполагает выпонение обучающимся следующих разделов:

- описание структуры с заданным именем, содержащую поля в соотвествии с индивидуальным заданием (не менее трёх);
- создание текстового файла, содержащего записи структуры (не менее пяти);
- разработка программы, выполняющей действия в соответсвии с индивидульным заданием.

Основная тематика: разработка структуры записей.

Рекомендуемый объём: поятснительная записка объёмом 10 страниц должна содержать титульный лист, цель работы, индивидуальное задание, листинг файла реализаци, скриншоты работы консольного приложения с описанием.

Выдача задания, приём контрольной работы проводится в соответствии с календарным учебным графиком.

Оценка	Критерии оценки контрольной работы
зачтено	Контрольная работа сдана в установленные сроки. Пояснительная записка оформлена в соответсвии с правилами ФГБОУ ВО «БрГУ». Листинг программы не содерджит ошибок. К скриншотам работы приложения приведены грамотные комментарии.
не зачтено	Контрольная работа не сдана в установленный срок.

5. МАТРИЦА СООТНЕСЕНИЯ РАЗДЕЛОВ УЧЕБНОЙ ДИСЦИПЛИНЫ К ФОРМИРУЕМЫМ В НИХ КОМПЕТЕНЦИЯМ И ОЦЕНКЕ РЕЗУЛЬТАТОВ ОСВОЕНИЯ ДИСЦИПЛИНЫ

Компетенции / *№, наименование разделов дисциплины*	*Кол-во часов*	*Компетенции*		*Σ комп.*	$t_{ср}$, *час*	*Вид учебных занятий*	*Оценка результатов*
		ОПК	*ПК*				
		9	**2**				
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1. Базовые средства языка С++	**19**	+	+	2	9,5	Лк, ЛР, СРС	Зачёт
2. Функции и управление памятью	**17,5**	+	+	2	8,75	Лк, ЛР, СРС	Зачёт
3. Введение в технологии программирования	**16,5**	+	+	2	8,25	Лк, ЛР, СРС	Зачёт
4. Классы	**18**	+	+	2	9,0	Лк, ЛР, СРС	Зачёт
5. Наследование	**19**	+	+	2	9,5	Лк, ЛР, СРС	Зачёт
6. Шаблоны и обработка исключительных ситуаций	**18**	+	+	2	9,0	Лк, ЛР, СРС	Зачёт
всего часов	**108**	**108**		**2**	**54**		

6. ПЕРЕЧЕНЬ УЧЕБНО-МЕТОДИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. Иванова, Г. С. Технология программирования : учебник / Г. С. Иванова. - М. : КНОРУС, 2011. - 336 с.

7. ПЕРЕЧЕНЬ ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ, НЕОБХОДИМОЙ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

№	***Наименование издания*** *(автор, заглавие, выходные данные)*	***Вид заня-тия***	***Количество экземпляров в библиоте-ке,*** **шт.**	***Обеспечен-ность,*** **(экз./ чел.)**
1	2	3	4	5
Основная литература				
1.	Ашарина, И. В. Объектно-ориентированное программирование в С++: лекции и упражнения : учеб. пособие для вузов / И.В. Ашарина. - М. : Горячая линия-Телеком, 2008. - 320 с.	Лк, ЛР	20	1,0
2.	Пахомов, Б. И. С/С++ и Borland C++ Builder для начинающих : учебное пособие / Б. И. Пахомов. - Санкт-Петербург : БХВ- Петербург, 2007. - 640 с.	ЛР	10	0,8
3.	Подбельский, В. В. Язык СИ++ : учебное пособие для вузов / В. В. Подбельский. - 5-е изд. - М. : Финансы и статистика, 2007. - 559 с.	ЛР	21	1,0
4.	Самохина, М. И. Объектно-ориентированное программирование на языке С++ : учеб. пособие / М. И. Самохина. - Братск : БрГУ, 2007. - 97 с. - Б. ц.	Лк	18	1,0
5.	Хорев, П. Б. Технологии объектно-ориентированного программирования : учеб. пособие для вузов / П. Б. Хорев. - 2-е изд., стереотип. - М. : Академия, 2008. - 448 с. - (Высшее профессиональное образование).	Лк	25	1,0
Дополнительная литература				
6.	Самохина, М.И., Крумин, О.К. Объектно-ориентированное программирование на языке С++: учеб. пособие. – Братск: Изд-во БрГУ, 2017. – 129 с.	Лк	13	1,0
7.	Самохина, М. И. С++. Объектно-ориентированное программирование : лабораторный практикум / М. И. Самохина, Н. А. Барковская. - Братск : БрГУ, 2008. - 62 с. - Б. ц.	ЛР	67	1,0
8.	Терехов, А. Н. Технология программирования : учеб. пособие для вузов / А.Н.Терехов. - 2-е изд. - М. : ИНТУИТ.РУ, 2007. - 148 с. - (Информационные технологии от первого лица).	Лк	5	0,4
9.	Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие / П. Б. Хорев. - 4-е изд., стереотип. - М. : Академия, 2012. - 448 с. - (Высшее профессиональное образование. Бакалавриат).	Лк, ЛР	6	0,5

8. ПЕРЕЧЕНЬ РЕСУРСОВ ИНФОРМАЦИОННО - ТЕЛЕКОММУНИКАЦИОННОЙ СЕТИ «ИНТЕРНЕТ» НЕОБХОДИМЫХ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

1.Электронный каталог библиотеки БрГУ
http://irbis.brstu.ru/CGI/irbis64r_15/cgiirbis_64.exe?LNG=&C21COM=F&I21DBN=BOOK&P21DBN=BOOK&S21CNR=&Z21ID=.

2. Электронная библиотека БрГУ

http://ecat.brstu.ru/catalog .
3. Электронно-библиотечная система «Университетская библиотека online» http://biblioclub.ru .
4. Электронно-библиотечная система «Издательство «Лань» http://e.lanbook.com .
5. Информационная система "Единое окно доступа к образовательным ресурсам" http://window.edu.ru .
6. Научная электронная библиотека eLIBRARY.RU http://elibrary.ru .
7. Университетская информационная система РОССИЯ (УИС РОССИЯ) https://uisrussia.msu.ru/ .
8. Национальная электронная библиотека НЭБ http://xn--90ax2c.xn--p1ai/how-to-search /.

9. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

9.1. Методические указания для обучающихся по выполнению лабораторных работ

Лабораторная работа №1
Обработка одномерных массивов и данных символьного и строкового типов

Цель работы:
Овладеть навыками обработки одномерных массивов и данных символьного и строкового типов, используя стандартные библиотечные функции.

Задание (один из возможных вариантов):
1. Вывести последовательность символов *ABBCCC…ZZ…Z*;
2. Дано натуральное число n ($n \le 99$). Выяснить, верно ли, что n^2 равно кубу суммы цифр числа n;
3. Даны действительные $a_1,..,a_{15},b_1,\dots,b_{15}$. Вычислить $(a_1+b_{15})*(a_2+b_{14})*\dots*(a_{15}+b_1)$.

Порядок выполнения:
Соответствует пунктам 1 – 3 задания.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Распечатанные изображения форм (если приложение неконсольное) и программный код файла реализации.

Задания для самостоятельной работы:
Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в первом разделе данной дисциплины.

Основная литература

1. Ашарина, И. В. Объектно-ориентированное программирование в С++: лекции и упражнения : учеб. пособие для вузов / И.В. Ашарина. - М. : Горячая линия-Телеком, 2008. - 320 с.

Дополнительная литература

2. Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие / П. Б.

Хорев. - 4-е изд., стереотип. - М. : Академия, 2012. - 448 с. - (Высшее профессиональное образование. Бакалавриат).

Контрольные вопросы для самопроверки
1. Дайте определение массива. Чем характеризуется массив?
2. Расскажите об описании и инициализации массива.
3. Дайте определние строки.
4. Перечислите функции обработки строк типа char.

Лабораторная работа №2
Использование прототипа функции. Перегрузка функции

Цель работы:
понять на конкретном примере назначение прототипа и перегрузки фукции.

Задание (один из возможных вариантов):
1. Даны функции $\begin{matrix} f(x)=x^2+2/x-\ln x^2 \\ f(y)=e^y+2\sqrt{y}+2 \end{matrix}$. Определить $z=\begin{cases} f(x+b)\cdot f(y), & \text{если} \quad x \text{ и } y>b \\ f(y+b)\cdot f(x), & \text{если} \quad x \text{ и } y \le b \end{cases}$;
2. Написать перегруженную функцию Dohod, которая вычисляет доход по вкладу. Исходными данными для функции являются: величина вклада, процентная ставка (годовых) и срок вклада (количество дней).

Порядок выполнения:
Соответствует пунктам 1 – 2 задания.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Распечатанные изображения форм (если приложение неконсольное) и программный код файла реализации.

Задания для самостоятельной работы:
Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным во втором разделе данной дисциплины.

Основная литература
1. Пахомов, Б. И. C/C++ и Borland C++ Builder для начинающих : учебное пособие / Б. И. Пахомов. - Санкт-Петербург : БХВ- Петербург, 2007. - 640 с.
Дополнительная литература
2. Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие / П. Б. Хорев. - 4-е изд., стереотип. - М. : Академия, 2012. - 448 с. - (Высшее профессиональное образование. Бакалавриат).

Контрольные вопросы для самопроверки
1. Дайте определение функции.
2. Почему функция является основным элементом программы на C++?
3. Дайте определение прототипа. Что даёт программисту использование прототипа в программе?

Лабораторная работа №3
Работа с файлами

Занятие проводится в интерактивной форме с разбором конкретных ситуаций

Цель работы:
Овладеть навыками программирования файловых структур данных, вывода данных в файл, чтения данных из файла, обработки файловых данных.

Задание (один из возможных вариантов):
1. создать файл, содержащий сведения для городской справочной службы. Структура записи: фамилия, имя, отчество, дата рождения, адрес, телефон;
2. написать программу, позволяющую по заданному номеру определить фамилию владельца телефона.

Порядок выполнения:
Соответствует пунктам 1 – 2 задания.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Распечатанные изображения форм (если приложение неконсольное) и программный код файла реализации.

Задания для самостоятельной работы:
Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в третьем разделе данной дисциплины.

Основная литература

1. Подбельский, В. В. Язык СИ++ : учебное пособие для вузов / В. В. Подбельский. - 5-е изд. - М. : Финансы и статистика, 2007. - 559 с.

Дополнительная литература

2. Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие / П. Б. Хорев. - 4-е изд., стереотип. - М. : Академия, 2012. - 448 с. - (Высшее профессиональное образование. Бакалавриат).

Контрольные вопросы для самопроверки
1. Дайте определения файла, потока.
2. Перечислите потоки ввода-вывода.
3. Перечислите классы файловых потоков ввода-вывода.

Лабораторная работа №4
Работа со структурами

Занятие в интерактивной форме с проведением тренингов в малой группе.

Цель работы:
Приобрести навыки и умения использования структур для работы со списком данных.

Задание (один из возможных вариантов):
1. описать структуру с именем *aeroflot*, содержащую следующие поля:
- название пункта назначения рейса;
- номер рейса;
- тип самолёта.
2. создать файл, содержащий 7 записей типа *aeroflot*;
3. написать программу, выполняющую следующие действия:
- выводит на экран номера рейсов и типов самолётов, вылетающих в пункт назначения, название которого совпало с названием, введённым с клавиатуры;
- если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Порядок выполнения:
Соответствует этапам 1 – 3 задания.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Распечатанные изображения форм (если приложение неконсольное) и программный код файла реализации.

Задания для самостоятельной работы:
Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в четвёртом разделе данной дисциплины.

Основная литература

1. Самохина, М. И. С++. Объектно-ориентированное программирование : лабораторный практикум / М. И. Самохина, Н. А. Барковская. - Братск : БрГУ, 2008. - 62 с. - Б. ц.

Дополнительная литература

2. Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие / П. Б. Хорев. - 4-е изд., стереотип. - М. : Академия, 2012. - 448 с. - (Высшее профессиональное образование. Бакалавриат).

Контрольные вопросы для самопроверки
1. Дайте определение структуры.
2. Как объявить структуру? Приведите пример.
3. Как определить объект структуры? Приведите пример.

Лабораторная работа №5
Работа с классами

Занятие проводится в интерактивной форме с разбором конкретных ситуаций.

Цель работы:
Овладеть навыками программирования классов данных; использования открытых и закрытых данных классов; создания массива указателей и массивов области динамического обмена.

Задание (один из возможных вариантов):
1. описать класс *star*, содержащий следующие элементы:
- конструктор;

- деструктор;
- методы доступа к закрытым переменным;
- две закрытые переменные, в одной из которых хранится имя звезды, а в другой её расстояние до Солнечной системы;

2. создать массив из 10 указателей на объекты класса *star*, хранящие в динамической памяти информацию о типе звезды. Предоставить пользователю возможность заполнить массив;
3. создать в области динамической памяти массив, содержащий 10 записей о расстоянии до Солнечной системы. Заполнить массив по формуле *i*-е значение = $20 * i^2 + 15$.

Порядок выполнения:
Соответствует пунктам 1 – 3 задания.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Распечатанные изображения форм (если приложение неконсольное) и программный код файла реализации.

Задания для самостоятельной работы:
Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в пятом разделе данной дисциплины.

Основная литература
1. Самохина, М. И. С++. Объектно-ориентированное программирование : лабораторный практикум / М. И. Самохина, Н. А. Барковская. - Братск : БрГУ, 2008. - 62 с. - Б. ц.
Дополнительная литература
2. Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие / П. Б. Хорев. - 4-е изд., стереотип. - М. : Академия, 2012. - 448 с. - (Высшее профессиональное образование. Бакалавриат).

Контрольные вопросы для самопроверки
1. Дайте определение указателя.
2. Как создать указатель?
3. В каких случаях используется операция разыменования?

Лабораторная работа №6
Перегрузка операций

Цель работы:
Овладеть навыками программирования перегрузки стандартных операций применительно к членам классов.

Задание (один из возможных вариантов):
1. создать класс, содержащий:
- закрытую переменную типа float;
- методы доступа, позволяющие изменить значение закрытой переменной;
- конструктор и деструктор;
- перегруженный конструктор, позволяющий инициализировать объект с заданным значением закрытой переменной.

2. создать два объекта этого класса А и В, а затем перегрузить операцию деления таким об-

разом, чтобы при помощи него вычислять выражение вида $C = \frac{sin(A)}{cos(B)}$.

Порядок выполнения:
Соответствует пунктам 1 – 2 задания.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Распечатанные изображения форм (если приложение неконсольное) и программный код файла реализации.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в шестом разделах данной дисциплины.

Основная литература
1. Самохина, М. И. С++. Объектно-ориентированное программирование : лабораторный практикум / М. И. Самохина, Н. А. Барковская. - Братск : БрГУ, 2008. - 62 с. - Б. ц.
Дополнительная литература
2. Хорев, П. Б. Объектно-ориентированное программирование : учебное пособие / П. Б. Хорев. - 4-е изд., стереотип. - М. : Академия, 2012. - 448 с. - (Высшее профессиональное образование. Бакалавриат).

Контрольные вопросы для самопроверки:
1. Расскажите о назначении перегрузки операций.
2. Перечислите правила перегрузки операций.
3. Что ещё можно перегружать в языке С++ и для чего?

9.2. Методические указания по выполнению контрольной работы

Структуры и члены структур

Структура – это совокупность переменных, скомпонованных с набором связанных функций. Структура может состоять из любых комбинаций типов данных, а также типов других структур. Переменные в структуре называются переменными-членами (данные-члены).

Переменные-члены относятся к той структуре, в которой они объявлены. Функции в структуре выполняют действия над переменными-членами. Они определяют функциональные возможности структуры.

Для объявления структуры используется ключевое слово *struct,* после которого указывается имя структуры и список данных-членов и методов структуры, которые заключаются в фигурные скобки.

Пример.

```
struct Cat
{
 int Age;
 int Weight;
void Meow();
};
```

При объявлении структуры, память под нее не резервируется, т.е. объявление сообщает компилятору о существовании структуры и о том, какие переменные и методы в ней используются.

Объявление объекта структуры

Объект нового типа создается тем же самым способом, что и обычная переменная:

```
unsigned int x; // определение переменной
Cat Frisky;
```

В данном случае определяется объект *Frisky*, который является объектом структуры *Cat* (т.е. имеет тип *Cat).*

Объект – это отдельный экземпляр некоторой структуры.

После определения реального объекта, может возникнуть необходимость в получении доступа к членам этого объекта. Для этого используется операция переменного доступа – точка (.). Например, чтобы присвоить число 5 переменной-члена *Weight* структуры *Cat* объекта *Frisky* необходимо записать:

```
Frisky.Weight = 5;
```

Аналогично, для вызовов метода структуры надо использовать запись:

```
Frisky.Meow();
```

Ограничение доступа к членам структуры

В объявлении структуры используются такие ключевые слова, как *public* (открытый) и *private* (закрытый). Все члены структуры, т.е. данные и методы являются открытыми по умолчанию. К закрытым же членам можно получить доступ только с помощью методов самой структуры. Открытые члены доступны для всех других функций программы.

Пример.

```
struct Cat
    {
    private:
    int Age;
    public:
    int Weight;
    void Meow();
    };
```

Например, если имеется структура *Cat,* приведенная выше, то обращение к ней переменной *Age* вызовет ошибку компиляции.

Далее рассматриваются пример формы и листинга программы для каждого пункта контрольной работы.

10. ПЕРЕЧЕНЬ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, ИСПОЛЬЗУЕМЫХ ПРИ ОСУЩЕСТВЛЕНИИ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

1. ОС Windows 7 Professional.
2. Microsoft Office 2007 Russian Academic OPEN No Level.
3. Антивирусное программное обеспечение Kaspersky Security.

11. ОПИСАНИЕ МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЙ БАЗЫ, НЕОБХОДИМОЙ ДЛЯ ОСУЩЕСТВЛЕНИЯ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

Вид занятия	***Наименование аудитории***	***Перечень основного оборудования***	***№ Лк или ЛР***
1	**2**	**3**	**4**
Лк	Лекционная аудитория	Меловая, маркерная доска	Лк 1-6
ЛР	Дисплейный класс	AMD Athlon 64 (5GHz/250Gb/2Gb/DD-RW), 2 ядра	ЛР 1-6
СР	ЧЗ №3	Оборудование 15 - CPU 5000/RAM 2Gb/HDD (Монитор TFT 19 LG 1953S-SF);принтер HP LaserJet P3005	

Приложение 1

ФОНД ОЦЕНОЧНЫХ СРЕДСТВ ДЛЯ ПРОВЕДЕНИЯ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. Описание фонда оценочных средств (паспорт)

№ компетенции	Элемент компетенции	Раздел	Тема	ФОС
ОПК-9	Способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности	**1.** Базовые средства языка С++	1.1. Алфавит и лексемы языка	Вопрос к зачёту 1.1
			1.3. Структура программы	Вопрос к зачёту 1.3
			1.5. Программирование алгоритмов различных структур	Вопрос к зачёту 1.5
		2. Функции и управление памятью	2.1. Функции	Вопрос к зачёту 2.1
		3. Введение в технологии программирования	3.1. Общие положения технологий программирования	Вопрос к зачёту 3.1
			3.2. Технологии программирования и информатизация общества	Вопрос к зачёту 3.2
		4. Классы	4.1. Описание объектов при помощи классов	Вопрос к зачёту 4.1
			4.3. Статические компоненты класса	Вопрос к зачёту 4.3
			4.5. Инициализация объектов	Вопрос к зачёту 4.5
		5. Наследование	5.1. Одиночное наследование	Вопрос к зачёту 5.1
			5.2. Виртуальные функции. Чисто виртуальные функции	Вопрос к зачёту 5.2
			6.3. Библиотека стандартных шаблонов	Вопрос к зачёту 6.3
			6.4. Обработка исключительных ситуаций	Вопрос к зачёту 6.4
ПК-2	способность проводить вычислительные эксперименты с использованием стандартных программных средств с целью получения математических моделей процессов и объектов автоматизации и управления	**1.** Базовые средства языка С++	1.2. Типы данных	Вопрос к зачёту 1.2
			1.4. Переменные и выражения	Вопрос к зачёту 1.4
		2. Функции и управление памятью	2.2. Выделение динамической памяти	Вопрос к зачёту 2.2
			2.3. Ссылки	Вопрос к зачёту 2.3
		3. Введение в технологии программирования	3.3. Ключевые понятия объектно-ориентированного программирования	Вопрос к зачёту 3.3
		4. Классы	4.2. Конструкторы и деструкторы	Вопрос к зачёту 4.2
			4.4. Указатели, ссылки и массивы объектов	Вопрос к зачёту 4.4

			4.6. Дружественные функции	Вопрос к зачёту 4.6
		5. Наследование	5.3. Множественное наследование	Вопрос к зачёту 5.3
			5.4. Виртуальные базовые классы	Вопрос к зачёту 5.4
		6. Шаблоны и обработка исключительных ситуаций	6.1. Шаблоны функций	Вопрос к зачёту 6.1
			6.2. Шаблоны классов	Вопрос к зачёту 6.2

2. Вопросы к зачёту

№ п/п	Компетенции		ВОПРОСЫ К ЗАЧЁТУ	№ и наименование раздела
	Код	Определение		
1	2	3	4	5
1.	ОПК-9	способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать требования информационной безопасности	**1.1.** Алфавит и лексемы языка	**1.** Базовые средства языка С++
			1.3. Структура программы	
			1.5. Программирование алгоритмов различных структур	
			2.1. Функции	**2.** Функции и управление памятью
			3.1. Общие положения технологий программирования	**3.** Введение в технологии программирования
			3.2. Технологии программирования и информатизация общества	
			4.1. Описание объектов при помощи классов	**4.** Классы
			4.3. Статические компоненты класса	
			4.5. Инициализация объектов	
			5.1. Одиночное наследование	**5.** Наследование
			5.2. Виртуальные функции. Чисто виртуальные функции	
			6.3. Библиотека стандартных шаблонов	
			6.4. Обработка исключительных ситуаций	
2.	ПК-2	способность проводить вычислительные эксперименты с использованием стандартных программных средств с целью получения математических моделей процессов и объектов автоматизации и управления	**1.2.** Типы данных	**1.** Базовые средства языка С++
			1.4. Переменные и выражения	
			2.2. Выделение динамической памяти	**2.** Функции и управление памятью
			2.3. Ссылки	
			3.3. Ключевые понятия объектно-ориентированного программирования	**3.** Введение в технологии программирования
			4.2. Конструкторы и деструкторы	**4.** Классы
			4.4. Указатели, ссылки и массивы объектов	
			4.6. Дружественные функции	
			5.3. Множественное наследование	**5.** Наследование
			5.4. Виртуальные базовые классы	
			6.1. Шаблоны функций	**6.** Шаблоны и обработка исключительных ситуаций
			6.2. Шаблоны классов	

3. Описание показателей и критериев оценивания компетенций

Показатели	Оценка	Критерии
Знать (ОПК-9): - требования информационной безопасности к программным системам; (ПК-2): - основные принципы и методологию разработки прикладного программного обеспечения; Уметь (ОПК-9): - использовать ИСР при проектировании и эксплуатации систем управления и автоматизации; (ПК-2): - проводить вычислительные эксперименты с использованием стандартных программных средств; Владеть: (ОПК-9): - методами информационных технологий; (ПК-2): - методами получения математических моделей процессов и объектов автоматизации и управления.	**зачтено**	Обучающийся должен во время ответа показать знания: основных конструкций языка С++ и базовой технологии созданий программ, типовые способы организции данных и построения алгоритмов их обработки, требования информционой безопасности к программным системам, основных терминов, используемых в научно-технической литературе по объектно-ориентированному программированию. Обучающийся должен иметь навыки владения: методами построения современных проблемно-ориентированных прикладных программных продуктов, понимания материала и способности высказывания мыслей на научно-техническом языке. Обучающийся во время ответа должен продемонстрировать умения: понимать структуру объектно-ориентированной программы, использовать ИСР при проектировании и эксплуатации систем управления и автоматизации.
	незачтено	На оба вопроса обучающийся отвечает неубедительно. На дополнительные вопросы преподавателя также не может ответить.

4. Методические материалы, определяющие процедуры оценивания знаний, умений, навыков и опыта деятельности

Дисциплина «Технологии программирования» направлена на формирование у обучающихся знаний и навыков использования современных технологий и методов разработки программных систем для решения практических задач с использованием современных инструментальных средств, необходимых в дальнейшем, при проектировании и эксплуатации систем управления и автоматизации.

Изучение дисциплины предусматривает:

– лекции;
– лабораторные работы;
– контрольную работу;
– самостоятельную работу;
– зачёт.

В ходе освоения раздела 1 «Базовые средства языка С++» обучающиеся должны уяснить:

алфавит и лексемы языка C++, типы даных, программирование алгоритмов различных структур.

В ходе освоения раздела 2 «Функци и управление памятью» обучающиеся должны знать: понятие функции, операции выделения динамической памяти, назначение ссылок.

В ходе освоения раздела 3 «Введение в технологии программирования» обучающиеся должны уяснить: основные определения и ключевые понятия объектно-ориентированного программирования, пять этапов информатизации общества.

В ходе освоения раздела 4 «Классы в C++» обучающиеся должны знать: определение класса, назначение конструкторов и деструкторов, статические компоненты класса, использование указателей, ссылок и массивов объектов, инициализацию объектов, применение дружественных функций.

В ходе освоения раздела 5 «Наследование» обучающиеся должны уяснить: механизм одиночного и множественного наследования, понятие виртуальной функции, виртуального базового класса.

В ходе освоения раздела 6 «Шаблоны и обработка исключительных ситуаций» обучающиеся должны знать: механизмы шаблонов функций, классов и обработки исключительных ситуаций, библиотеку стандартных шаблонов.

В процессе проведения лабораторных работ происходит закрепление знаний, формирование умений и навыков использования ИСР C++ Builder для разработки объектно-ориентированного обеспечения.

В выполнении контрольной работы обучающиеся приобретают навыки программирования одного из типов данных объектно-ориентированного подхода – структур.

При подготовке к зачёту рекомендуется особое внимание уделить следующим вопросам: описание объектов при помощи классов, конструкторы и деструкторы, одиночное и множественное наследования.

Работа с литературой является важнейшим элементом в получении знаний по дисциплине. Прежде всего, необходимо воспользоваться списком рекомендуемой литературы. Дополнительные сведения по изучаемым темам можно найти в периодической печати и Интернете.

Предусмотрено проведение аудиторных занятий в интерактивной форме (лекции с текущим контролем, лабораторные работы с разбором конкретных ситуаций) в сочетании с внеаудиторной работой.

Приложение 2

АННОТАЦИЯ
рабочей программы дисциплины
Технологии программирования

1. Цель и задачи дисциплины

Формирование у обучающихся знаний и навыков по использованию современных технологий и методов разработки программных систем для решения практических задач с использованием современных инструментальных средств, необходимых в дальнейшем при проектировании и эксплуатации систем управления и автоматизации.

Задачей дисциплины является освоение принципов и методов объектно-ориентированного программирования с использованием интегрированной среды разработки (ИСР) C++ Builder.

2. Структура дисциплины

2.1 Распределение трудоемкости по отдельным видам учебных занятий, включая самостоятельную работу: Лк – 5 час.; ЛР – 10 час.; СР – 89 час.

Общая трудоёмкость дисциплины составляет 108 часа, 3 зачётные единицы.

2.2 Основные разделы дисциплины:

1. Базовые средства языка С++;
2. Функции управления памятью;
3. Введение в технологии программирования;
4. Классы;
5. Наследование;
6. Шаблоны и обработка исключительных ситуаций.

3. Планируемые результаты обучения (перечень компетенций)

Процесс изучения дисциплины направлен на формирование следующей компетенции:
ОПК-9 - способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности;
ПК-2 - способность проводить вычислительные эксперименты с использованием стандартных программных средств с целью получения математических моделей процессов и объектов автоматизации и управления.

4. Вид промежуточной аттестации: зачёт.

Приложение 3

Протокол о дополнениях и изменениях в рабочей программе
на 201___ - 201___ учебный год

1. В рабочую программу по дисциплине вносятся следующие дополнения:
____
____

2. В рабочую программу по дисциплине вносятся следующие изменения:
____
____
____

Протокол заседания кафедры №____ от «___» __________ 201 ____ г.,
(разработчик)

Заведующий кафедрой ______________________ ______________________
(подпись) *(Ф.И.О.)*

Программа составлена в соответствии с федеральным государственным образовательным стандартом высшего образования по направлению подготовки 27.03.04 Управление в технических системах от «20» октября 2015 г. № 1171

для набора 2014 года: и учебным планом ФГБОУ ВО «БрГУ» для заочной формы обучения от «03» июля 2017 г. № 413.

Программу составил:

Крумин О.К., доцент кафедры УТС, к.т.н. ____________________

Рабочая программа рассмотрена и утверждена на заседании кафедры УТС

от 28 декабря 2018 г, протокол № 6

Заведующий кафедрой УТС _____________ Игнатьев И.В.

СОГЛАСОВАНО:

Заведующий выпускающей кафедрой ______________ Игнатьев И.В.

Директор библиотеки ______________ Сотник Т.Ф.

Рабочая программа одобрена методической комиссией ФЭиА факультета

от 28 декабря 2018 г, протокол № 5

Председатель методической комиссии факультета __________ Ульянов А.Д.

СОГЛАСОВАНО:

Начальник
учебно-методического управления _______________________ Нежевец Г.П.

Регистрационный №________