МИНИСТЕРСТВО ВЫСШЕГО И СРЕДНЕ СПЕЦИАЛЬНОГО ОБРАЗОВАНИЯ

ГУЛИСТАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

КАФЕДРА ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

ОСНОВЫ ПРОГРАММИРОВАНИЯ

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС

Область знаний: **100000 – гуманитарная**

Область образования: **130000 – Математика**

Направление образования: **5130100-Математика**

ГУЛИСТАН - 2018

Данный учебно-методический комплекс подготовлен на основе типовой учебной программы, утверждённой 18 августа 2017 года (БД-5130100-2.01) Министерством высшего и средне-специального образования.

Составитель: **Каландаров А.А.-ст.преп. кафедры “Информационные технологии”**

Рецензент: **Кулмаматов С.И.-доц. кафедры “Информационные технологии”,**

Данный учебно-методический комплекс по предмету «Информатика и информационные технологии» был обсуждён и одобрен Учебно-Методическим Советом Гулистанского государственного университета на заседании № 1 от 26.08.2017 года.

ОГЛАВЛЕНИЕ

Начальные сведения о языке

1.1 История и назначение языка Си++

Разработчиком языка Си++ является Бьерн Страуструп. В своей работе он опирался на опыт создателей языков Симула, Модула 2, абстрактных типов данных. Основные работы велись в исследовательском центре компании Bell Labs.

Непосредственный предшественник Си++ – язык Си с классами – появился в 1979 году, а в 1997 году был принят международный стандарт Си++, который фактически подвел итоги его 20-летнего развития. Принятие стандарта обеспечило единообразие всех реализаций языка Си++. Не менее важным результатом стандартизации стало то, что в процессе выработки и утверждения стандарта язык был уточнен и дополнен рядом существенных возможностей.

На сегодня стандарт утвержден Международной организацией по стандартзации ISO. Его номер ISO/IEC 14882. ISO бесплатно стандарты не распространяет. Его можно получить на узле американского национального комитета по стандартам в информационных технологиях: www.ncits.org В России следует обращаться в ВНИИ Сертификации: http://www.vniis.ru

Проекты стандарта имеются в свободном доступе: ftp://ftp.research.att.com/dist/c++std/WP/CD2/
http://www/research.att.com/~bs/bs_faq.html

Язык Си++ является универсальным языком программирования, в дополнение к которому разработан набор разнообразных библиотек. Поэтому, строго говоря, он позволяет решить практически любую задачу программирования. Тем не менее, в силу разных причин (не всегда технических) для каких-то типов задач он употребляется чаще, а для каких-то – реже.

Си++ как преемник языка Си широко используется в системном программировании. На нем можно писать высокоэффективные программы, в

том числе операционные системы, драйверы и т.п. Язык Си++ – один из основных языков разработки трансляторов.

Поскольку системное программное обеспечение часто бывает написано на языке Си или Си++, то и программные интерфейсы к подсистемам ОС тоже часто пишут на Си++. Соответственно, те программы, даже и прикладные, которые взаимодействуют с операционными системами, написаны на языке Си++.

Распределенные системы, функционирующие на разных компьютерах, также разрабатываются на языке Си++. Этому способствует то, что у широко распространенных компонентых моделей CORBA и COM есть удобные интерфейсы на языке Си++.

Обработка сложных структур данных – текста, бизнес-информации, Internet-страниц и т.п. – одна из наиболее распространенных возможностей применения языка. В прикладном программировании, наверное, проще назвать те области, где язык Си++ применяется мало.

Разработка графического пользовательского интерфейса на языке Си++ выполняется, в основном, тогда, когда необходимо разрабатывать сложные, нестандартные интерфейсы. Простые программы чаще пишутся на языках Visual Basic, Java и т.п.

Программирование для Internet в основном производится на языках Java, VBScript, Perl.

В целом надо сказать, что язык Си++ в настоящее время является одним из наиболее распространенных языков программирования в мире.

1.2 Простейшая программа на языке Си++

Самая короткая программа на языке Си++ выглядит так:

```
// Простейшая программа
int main() { return 1; }
```

Первая строчка в программе – комментарий, который служит лишь для пояснения. Признаком комментария являются два знака деления подряд (//).

main – это имя главной функции программы. С функции main всегда начинается выполнение. У функции есть имя (main), после имени в круглых скобках перечисляются аргументы или параметры функции (в данном случае у функции main аргументов нет). У функции может быть результат или возвращаемое значение. Если функция не возвращает никакого значения, то это обозначается ключевым словом void. В фигурных скобках записывается тело функции – действия, которые она выполняет. Оператор return 1 означает, что функция возвращает результат – целое число 1.

Если мы говорим об объектно-ориентированной программе, то она должна создать объект какого-либо класса и послать ему сообщение. Чтобы не усложнять программу, мы воспользуемся одним из готовых, предопределенных классов – классом ostream (поток ввода-вывода). Этот класс определен в файле заголовков "iostream.h". Поэтому первое, что надо сделать – включить файл заголовков в нашу программу:

```
#include <iostream.h>
int main() { return 1; }
```

Кроме класса, файл заголовков определяет глобальный объект этого класса cout. Объект называется глобальным, поскольку доступ к нему возможен из любой части программы. Этот объект выполняет вывод на консоль. В функции main мы можем к нему обратиться и послать ему сообщение:

```
#include <iostream.h>
int main()
{
  cout << "Hello world!" << endl;
  return 1;
}
```

Операция сдвига << для класса ostream определена как "вывести". Таким образом, программа посылает объекту cout сообщения "вывести строку Hello world!" и "вывести перевод строки" (endl обозначает новую строку). В ответ

на эти сообщения объект cout выведет строку "Hello world!" на консоль и переведет курсор на следующую строку.

1.3 Компиляция и выполнение программы

Программа на языке Си++ – это текст. С помощью произвольного текстового редактора программист записывает инструкцию, в соответствии с которой компьютер будет работать, выполняя данную программу.

Для того чтобы компьютер мог выполнить программу, написанную на языке Си++, ее нужно перевести на язык машинных инструкций. Эту задачу решает компилятор. Компилятор читает файл с текстом программы, анализирует ее, проверяет на предмет возможных ошибок и, если таковых не обнаружено, создает исполняемый файл, т.е. файл с машинными инструкциями, который можно выполнять.

Откомпилировав программу один раз, ее можно выполнять многократно, с различными исходными данными.

Не имея возможности описать все варианты, остановимся только на двух наиболее часто встречающихся.

1.4 Компилирование и выполнение программ в среде Windows

Если Вы используете персональный компьютер с операционной системой Microsoft© Windows 98™, Windows NT™ или Windows 2000™, то компилятор у Вас, скорее всего, Visual C++©. Этот компилятор представляет собой интегрированную среду программирования, т.е. объединяет текстовый редактор, компилятор, отладчик и еще ряд дополнительных программ. Мы предполагаем, что читатель работает с версией 5.0 или старше. Версии младше 4.2 изучать не имеет смысла, поскольку реализация слишком сильно отличается от стандарта языка.

В среде Visual C++ прежде всего необходимо создать новый проект. Для этого нужно выбрать в меню File атрибут New. Появится новое диалоговое окно. В закладке Projects в списке различных типов выполняемых файлов выберите Win32 Consol Application. Убедитесь, что отмечена кнопка Create

new workspace. Затем следует набрать имя проекта (например, test) в поле Project name и имя каталога, в котором будут храниться все файлы, относящиеся к данному проекту, в поле Location. После этого нажмите кнопку "OK".

Теперь необходимо создать файл. Опять в меню File выберите атрибут New. В появившемся диалоге в закладке File отметьте text file. По умолчанию новый файл будет добавлен к текущему проекту test , в чем можно убедиться, взглянув на поле Add to project. В поле Filename нужно ввести имя файла. Пусть это будет main.cpp. Расширение .cpp – это стандарт для файлов с исходными текстами на языке Си++. Поле Location должно показывать на каталог C:\Work. Нажмите кнопку "OK".

На экране появится пустой файл. Наберите текст программы.

Компиляция выполняется с помощью меню Build. Выберите пункт Build test.exe (этому пункту меню соответствует функциональная клавиша F7). В нижней части экрана появятся сообщения компиляции. Если Вы сделали опечатку, двойной щелчок мышью по строке с ошибкой переведет курсор в окне текстового редактора на соответствующую строку кода. После исправления всех ошибок и повторной компиляции система выдаст сообщение об успешной компиляции и компоновке (пока мы не будем уточнять, просто вы увидите сообщение Linking).

Готовую программу можно выполнить с помощью меню Build, пункт Execute test.exe. То же самое можно сделать, нажав одновременно клавиши CTRL и F5. На экране монитора появится консольное окно, и в нем будет выведена строка "Hello world!". Затем появится надпись "Press any key to continue". Эта надпись означает, что программа выполнена и лишь ожидает нажатия произвольной клавиши, чтобы закрыть консольное окно.

1.5 Компилирование и выполнение программ в среде Unix

Если Вы работаете в операционной системе Unix , то, скорее всего, у Вас нет интегрированной среды разработки программ. Вы будете пользоваться любым доступным текстовым редактором для того, чтобы набирать тексты

программ.

Редактор Emacs предпочтительнее, поскольку в нем есть специальный режим редактирования программ на языке Си++. Этот режим включается автоматически при редактировании файла с именем, оканчивающимся на ".cpp" или ".h". Но при отсутствии Emacs сгодится любой текстовый редактор.

Первое, что надо сделать – это поместить текст программы в файл. В редакторе следует создать файл с именем main.cpp (расширение cpp используется для текстов программ на языке Си++). Наберите текст программы из предыдущего параграфа и сохраните файл.

Теперь программу надо откомпилировать. Команда вызова компилятора зависит от того, какой компилятор Си++ установлен на компьютере. Если используется компилятор GNU C++, команда компиляции выглядит так:

gcc main.cpp

Вместо gcc может использоваться g++, c++, cc. Уточнить это можно у системного администратора. Отметим, что у широко распространенного компилятора GNU C++ есть ряд отличий от стандарта ISO.

В случае каких-либо ошибок в программе компилятор выдаст на терминал сообщение с указанием номера строки, где обнаружена ошибка. Если в программе нет никаких опечаток, компилятор должен создать исполняемый файл с именем a.out. Выполнить его можно, просто набрав имя a.out в ответ на подсказку интерпретатора команд:

a.out

Результатом выполнения будет вывод на экран терминала строки:

Hello world!

2 Имена, переменные и константы

2.1 Имена

Для символического обозначения величин, имен функций и т.п. используются имена или идентификаторы.

Идентификаторы в языке Си++ – это последовательность знаков, начинающаяся с буквы. В идентификаторах можно использовать заглавные и строчные латинские буквы, цифры и знак подчеркивания. Длина идентификаторов произвольная. Примеры правильных идентификаторов:

abc A12 NameOfPerson BITES_PER_WORD

Отметим, что abc и Abc – два разных идентификатора, т.е. заглавные и строчные буквы различаются. Примеры неправильных идентификаторов:

12X a-b

Ряд слов в языке Си++ имеет особое значение и не может использоваться в качестве идентификаторов. Такие зарезервированные слова называются ключевыми.

Список ключевых слов:

asm auto bad_cast
bad_typeid bool break
case catch char
class const const_cast
continue default delete
do double dynamic_cast
else enum extern
float for friend
goto if inline
int long namespace
new operator private
protected public register
reinterpret_cast return short
signed sizeof static
static_cast struct switch

template then this
throw try type_info
typedef typeid union
unsigned using virtual
void volatile while
xalloc

В следующем примере

```
int max(int x, int y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

max, x и y – имена или идентификаторы. Слова int, if, return и else – ключевые слова, они не могут быть именами переменных или функций и используются для других целей.

2.2 Переменные

Программа оперирует информацией, представленной в виде различных объектов и величин. Переменная – это символическое обозначение величины в программе. Как ясно из названия, **значение** переменной (или величина, которую она обозначает) во время выполнения программы может изменяться.

С точки зрения архитектуры компьютера, переменная – это символическое обозначение ячейки оперативной памяти программы, в которой хранятся данные. Содержимое этой ячейки – это текущее значение переменной.

В языке Си++ прежде чем использовать переменную, ее необходимо **объявить**. Объявить переменную с именем x можно так:

int x;

В объявлении первым стоит название типа переменной int (целое число), а затем идентификатор x – имя переменной. У переменной x есть тип – в данном случае целое число. Тип переменной определяет, какие

возможные значения эта переменная может принимать и какие операции можно выполнять над данной переменной. Тип переменной изменить нельзя, т.е. пока переменная x существует, она всегда будет целого типа.

Язык Си++ – это строго типизированный язык. Любая величина, используемая в программе, принадлежит к какому-либо типу. При любом использовании переменных в программе проверяется, применимо ли выражение или операция к типу переменной. Довольно часто смысл выражения зависит от типа участвующих в нем переменных.

Например, если мы запишем x+y, где x – объявленная выше переменная, то переменная y должна быть одного из числовых типов.

Соответствие типов проверяется во время компиляции программы. Если компилятор обнаруживает несоответствие типа переменной и ее использования, он выдаст ошибку (или предупреждение). Однако во время выполнения программы типы не проверяются. Такой подход, с одной стороны, позволяет обнаружить и исправить большое количество ошибок на стадии компиляции, а, с другой стороны, не замедляет выполнения программы.

Переменной можно присвоить какое-либо значение с помощью операции **присваивания**. Присвоить – это значит установить текущее значение переменной. По-другому можно объяснить, что операция присваивания запоминает новое значение в ячейке памяти, которая обозначена переменной.

```
int x;      // объявить целую переменную x
int y;      // объявить целую переменную y
x = 0;      // присвоить x значение 0
y = x + 1;  // присвоить y значение x + 1,
            // т.е. 1
x = 1;      // присвоить x значение 1
y = x + 1;  // присвоить y значение x + 1,
            // теперь уже 2
```

2.3 Константы

В программе можно явно записать величину – число, символ и т.п.

Например, мы можем записать выражение x + 4 – сложить текущее значение переменной x и число 4. В зависимости от того, при каких условиях мы будем выполнять программу, значение переменной x может быть различным. Однако целое число четыре всегда останется прежним. Это неизменяемая величина или константа.

Таким образом, явная запись значения в программе – это константа.

Далеко не всегда удобно записывать константы в тексте программы явно. Гораздо чаще используются символические константы. Например, если мы запишем

```
const int BITS_IN_WORD = 32;
```

то затем имя BITS_IN_WORD можно будет использовать вместо целого числа 32.

Преимущества такого подхода очевидны. Во-первых, имя BITS_IN_WORD (битов в машинном слове) дает хорошую подсказку, для чего используется данное число. Без комментариев понятно, что выражение

```
b / BITS_IN_WORD
```

(значение b разделить на число 32) вычисляет количество машинных слов, необходимых для хранения b битов информации. Во-вторых, если по каким-либо причинам нам надо изменить эту константу, потребуется изменить только одно место в программе – определение константы, оставив все случаи ее использования как есть. (Например, мы переносим программу на компьютер с другой длиной машинного слова.)

3 Операции и выражения

3.1 Выражения

Программа оперирует с данными. Числа можно складывать, вычитать, умножать, делить. Знаки можно сравнивать и т.д. То есть из разных величин можно составлять выражения, результат вычисления которых – новая величина. Приведем примеры выражений:

```
X * 12 + Y  // значение X умножить на 12 и к
            // результату прибавить значение Y
val < 3     // сравнить значение val с 3
-9          // константное выражение -9
```

Выражение, после которого стоит точка с запятой – это оператор-выражение. Его смысл состоит в том, что компьютер должен выполнить все действия, записанные в данном выражении, иначе говоря, вычислить выражение.

```
x + y – 12; // сложить значения x и y и затем
            // вычесть 12
a = b + 1;  // прибавить единицу к значению b и
            // запомнить результат в переменной a
```

Выражения – это переменные, функции и константы, называемые операндами, объединенные знаками операций. **Операции** могут быть унарными – с одним операндом, например, минус; могут быть бинарные – с двумя операндами, например сложение или деление. В Си++ есть даже одна операция с тремя операндами – условное выражение. Чуть позже мы приведем список всех операций языка Си++ для встроенных типов данных. Подробно каждая операция будет разбираться при описании соответствующего типа данных. Кроме того, ряд операций будет рассмотрен в разделе, посвященном определению операторов для классов. Пока что мы ограничимся лишь общим описанием способов записи выражений.

В типизированном языке, которым является Си++, у переменных и констант есть определенный тип. Есть он и у результата выражения. Например, операции сложения (+), умножения (*), вычитания (-) и деления (/), примененные к целым числам, выполняются по общепринятым

математическим правилам и дают в результате целое значение. Те же операции можно применить к вещественным числам и получить вещественное значение.

Операции сравнения: больше (>), меньше (<), равно (==), не равно (!=) сравнивают значения чисел и выдают логическое значение: истина (true) или ложь (false).

3.2 Операция присваивания

Присваивание – это тоже операция, она является частью выражения. Значение правого операнда присваивается левому операнду.

```
x = 2;       // переменной x присвоить значе-
cond = x < 2; // ние 2, переменной cond
          // присвоить значение true,
          // если x меньше 2, в противном
          // случае присвоить значение
3 = 5;       // false ошибка, число 3
          // неспособно изменять свое
          // значение
```

Последний пример иллюстрирует требование к левому операнду операции присваивания. Он должен быть способен хранить и изменять свое значение. Переменные, объявленные в программе, обладают подобным свойством. В следующем фрагменте программы

```
int x = 0;
x = 3;
x = 4;
x = x + 1;
```

вначале объявляется переменная x с начальным значением 0. После этого значение x изменяется на 3, 4 и затем 5. Опять-таки, обратим внимание на последнюю строчку. При вычислении операции присваивания сначала вычисляется левый операнд, а затем правый. Когда вычисляется выражение x + 1, значение переменной x равно 4. Поэтому значение выражения x + 1 равно 5. После вычисления операции присваивания (или, проще говоря, после присваивания) значение переменной x становится равным 5.

У операции присваивания тоже есть результат. Он равен значению

левого операнда. Таким образом, операция присваивания может участвовать в более сложном выражении:

z = (x = y + 3);

В приведенном примере переменным x и z присваивается значение y + 3.

Очень часто в программе приходится значение переменной увеличивать или уменьшать на единицу. Для того чтобы сделать эти действия наиболее эффективными и удобными для использования, применяются предусмотренные в Си++ специальные знаки операций: ++ (увеличить на единицу) и -- (уменьшить на единицу). Существует две формы этих операций: префиксная и постфиксная. Рассмотрим их на примерах.

```
int x = 0;
++x;
```

Значение x увеличивается на единицу и становится равным 1.

```
--x;
```

Значение x уменьшается на единицу и становится равным 0.

```
int y = ++x;
```

Значение x опять увеличивается на единицу. Результат операции ++ – новое значение x, т.е. переменной y присваивается значение 1.

```
int z = x++;
```

Здесь используется постфиксная запись операции увеличения на единицу. Значение переменной x до выполнения операции равно 1. Сама операция та же – значение x увеличивается на единицу и становится равным 2. Однако результат постфиксной операции – значение аргумента до увеличения. Таким образом, переменной z присваивается значение 1. Аналогично, результатом постфиксной операции уменьшения на единицу является начальное значение операнда, а префиксной – его конечное значение.

Подобными мотивами оптимизации и сокращения записи руководствовались создатели языка Си (а затем и Си++), когда вводили новые знаки операций типа "выполнить операцию и присвоить". Довольно часто

одна и та же переменная используется в левой и правой части операции присваивания, например:

```
x = x + 5;
y = y * 3;
z = z – (x + y);
```

В Си++ эти выражения можно записать короче:

```
x += 5;
y *= 3;
z -= x + y;
```

Т.е. запись oper= означает, что левый операнд вначале используется как левый операнд операции oper, а затем как левый операнд операции присваивания результата операции oper. Кроме краткости выражения, такая запись облегчает оптимизацию программы компилятором.

3.3 Все операции языка Си++

Наряду с общепринятыми арифметическими и логическими операциями, в языке Си++ имеется набор операций для работы с битами – поразрядные И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и НЕ, а также сдвиги.

Особняком стоит операция sizeof. Эта операция позволяет определить, сколько памяти занимает то или иное значение. Например:

```
sizeof(long);
// сколько байтов занимает тип long

sizeof b;
// сколько байтов занимает переменная b
```

Операция sizeof в качестве аргумента берет имя типа или выражение. Аргумент заключается в скобки (если аргумент – выражение, скобки не обязательны). Результат операции – целое число, равное количеству байтов, которое необходимо для хранения в памяти заданной величины.

Ниже приводятся все операции языка Си++.

3.4 Арифметические операции

+ сложение
- вычитание

*** умножение**
/ деление

Операции сложения, вычитания, умножения и деления целых и вещественных чисел. Результат операции – число, по типу соответствующее большему по разрядности операнду. Например, сложение чисел типа short и long в результате дает число типа long.

% остаток

Операция нахождения остатка от деления одного целого числа на другое. Тип результата – целое число.

- минус
+ плюс

Операция "минус" – это унарная операция, при которой знак числа изменяется на противоположный. Она применима к любым числам со знаком. Операция "плюс" существует для симметрии. Она ничего не делает, т.е. примененная к целому числу, его же и выдает.

++ увеличить на единицу, префиксная и постфиксная формы
-- уменьшить на единицу, префиксная и постфиксная формы

Эти операции иногда называют "автоувеличением" и "автоуменьшением". Они увеличивают (или, соответственно, уменьшают) операнд на единицу. Разница между постфиксной (знак операции записывается после операнда, например x++) и префиксной (знак операции записывается перед операндом, например --y) операциями заключается в том, что в первом случае результатом является значение операнда до изменения на единицу, а во втором случае – после изменения на единицу.

3.5 Операции сравнения

== равно
!= не равно
< меньше
> больше
<= меньше или равно
>= больше или равно

Операции сравнения. Сравнивать можно операнды любого типа, но либо они должны быть оба одного и того же встроенного типа (сравнение на равенство и неравенство работает для двух величин любого типа), либо между ними должна быть определена соответствующая операция сравнения. Результат – логическое значение true или false.

3.6 Логические операции

&& логическое И
|| логическое ИЛИ
! логическое НЕ

Логические операции конъюнкции, дизъюнкции и отрицания. В качестве операндов выступают логические значения, результат – тоже логическое значение **true** или **false**.

3.7 Битовые операции

& битовое И
| битовое ИЛИ
^ битовое ИСКЛЮЧАЮЩЕЕ ИЛИ
~ битовое НЕ

Побитовые операции над целыми числами. Соответствующая операция выполняется над каждым битом операндов. Результатом является целое число.

<< сдвиг влево
>> сдвиг вправо

Побитовый сдвиг левого операнда на количество разрядов, соответствующее значению правого операнда. Результатом является целое число.

3.8 Условная операция

? : условное выражение

Трехарная операция; если значение первого операнда – истина, то результат – второй операнд; если ложь – результат – третий операнд. Первый операнд должен быть логическим значением, второй и третий операнды могут быть любого, но одного и того же, типа, а результат будет того же типа,

что и третий операнд.

3.9 Последовательность

, последовательность

Выполнить выражение до запятой, затем выражение после запятой. Два произвольных выражения можно поставить рядом, разделив их запятой. Они будут выполняться **последовательно**, и результатом всего выражения будет результат последнего выражения.

3.10 Операции присваивания

= присваивание

Присвоить значение правого операнда левому. Результат операции присваивания – это значение правого операнда.

+=, -=, *=, /=, %=, |=, &=, ^=, <<=, >>=
выполнить операцию и присвоить

Выполнить соответствующую операцию с левым операндом и правым операндом и присвоить результат левому операнду. Типы операндов должны быть такими, что, во-первых, для них должна быть определена соответствующая арифметическая операция, а во-вторых, результат может быть присвоен левому операнду.

3.11 Порядок вычисления выражений

У каждой операции имеется приоритет. Если в выражении несколько операций, то первой будет выполнена операция с более высоким приоритетом. Если же операции одного и того же приоритета, они выполняются слева направо.

Например, в выражении

2 + 3 * 6

сначала будет выполнено умножение, а затем сложение;соответственно, значение этого выражения — число 20.

В выражении

2 * 3 + 4 * 5

сначала будет выполнено умножение, а затем сложение. В каком порядке будет производиться умножение – сначала 2 * 3, а затем 4 * 5 или наоборот, не определено. Т.е. для операции сложения порядок вычисления ее операндов не задан.

В выражении

x = y + 3

вначале выполняется сложение, а затем присваивание, поскольку приоритет операции присваивания ниже, чем приоритет операции сложения.

Для данного правила существует исключение: если в выражении несколько операций присваивания, то они выполняются справа налево. Например, в выражении

x = y = 2

сначала будет выполнена операция присваивания значения 2 переменной y. Затем результат этой операции – значение 2 – присваивается переменной x.

Ниже приведен список всех операций в порядке понижения приоритета. Операции с одинаковым приоритетом выполняются слева направо (за исключением нескольких операций присваивания).

:: **(разрешение области видимости имен)**
. **(обращение к элементу класса),**
-> **(обращение к элементу класса по указателю),**
[] **(индексирование), вызов функции,**
++ **(постфиксное увеличение на единицу),**
-- **(постфиксное уменьшение на единицу),**
typeid **(нахождение типа),**
dynamic_cast static_cast reinterpret_cast const_cast **(преобразования типа)**
sizeof **(определение размера),**
++ **(префиксное увеличение на единицу),**
-- **(префиксное уменьшение на единицу),**
~ **(битовое НЕ),**
! **(логическое НЕ),**
– **(изменение знака),**
+ **(плюс),**

& **(взятие адреса),**
* **(обращение по адресу),**
new **(создание объекта),**
delete **(удаление объекта),**
(type) **(преобразование типа)**
.* ->* **(обращение по указателю на элемент класса)**
* **(умножение),**
/ **(деление),**
% **(остаток)**
+ **(сложение),**
– **(вычитание)**
<< , >> **(сдвиг)**
< <= > >= **(сравнения на больше или меньше)**
== != **(равно, неравно)**
& **(поразрядное И)**
^ **(поразрадное исключающее ИЛИ)**
| **(поразрядное ИЛИ)**
&& **(логическое И)**
|| **(логическое ИЛИ)**
= **(присваивание),**
*= /= %= += -= <<= >>= &= |= ^= **(выполнить операцию и присвоить)**
?: **(условная операция)**
throw
, **(последовательность)**

Для того чтобы изменить последовательность вычисления выражений, можно воспользоваться круглыми скобками. Часть выражения, заключенная в скобки, вычисляется в первую очередь. Значением

(2 + 3) * 6

будет 30.

Скобки могут быть вложенными, соответственно, самые внутренние выполняются первыми:

(2 + (3 * (4 + 5)) – 2)

4 Операторы

4.1 Что такое оператор

Запись действий, которые должен выполнить компьютер, состоит из операторов. При выполнении программы операторы выполняются один за другим, если только оператор не является оператором управления, который может изменить последовательное выполнение программы.

Различают операторы объявления имен, операторы управления и операторы-выражения.

4.2 Операторы-выражения

Выражения мы рассматривали в предыдущей лекции. Выражение, после которого стоит точка с запятой, – это оператор-выражение. Его смысл состоит в том, что компьютер должен выполнить все действия, записанные в данном выражении, иначе говоря, вычислить выражение. Чаще всего в операторе-выражении стоит операция присваивания или вызов функции. Операторы выполняются последовательно, и все изменения значений переменных, сделанные в предыдущем операторе, используются в последующих.

```
a = 1;
b = 3;
m = max(a, b);
```

Переменной a присваивается значение 1, переменной b – значение 3. Затем вызывается функция max с параметрами 1 и 3, и ее результат присваивается переменной m.

Как мы уже отмечали, присваивание – необязательная операция в операторе-выражении. Следующие операторы тоже вполне корректны:

```
x + y – 12;     // сложить значения x и y и
                // затем вычесть 12
func(d, 12, x)  // вызвать функцию func с
                // заданными параметрами
```

4.3 Объявления имен

Эти операторы объявляют имена, т.е. делают их известными программе. Все идентификаторы или имена, используемые в программе на языке Си++, должны быть объявлены.

Оператор объявления состоит из названия типа и объявляемого имени:

```
int  x;        // объявить целую переменную x
double f;      // объявить переменную f типа
               // double
const float pi = 3.1415;
   // объявить константу pi типа float
   // со значением 3.1415
```

Оператор объявления заканчивается точкой с запятой.

4.4 Операторы управления

Операторы управления определяют, в какой последовательности выполняется программа. Если бы их не было, операторы программы всегда выполнялись бы последовательно, в том порядке, в котором они записаны.

4.5 Условные операторы

Условные операторы позволяют выбрать один из вариантов выполнения действий в зависимости от каких-либо условий. Условие – это логическое выражение, т.е. выражение, результатом которого является логическое значение true (истина) или false (ложь).

Оператор if выбирает один из двух вариантов последовательности вычислений.

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

Если условие истинно, выполняется оператор1, если ложно, то выполняется оператор2.

```
if  (x > y)
   a = x;
```

```
else
    a = y;
```

В данном примере переменной a присваивается значение максимума из двух величин x и y.

Конструкция else необязательна. Можно записать:

```
if   (x < 0)
    x = -x;
abs = x;
```

В данном примере оператор x = -x; выполняется только в том случае, если значение переменной x было отрицательным. Присваивание переменной abs выполняется в любом случае. Таким образом, приведенный фрагмент программы изменит значение переменной x на его абсолютное значение и присвоит переменной abs новое значение x.

Если в случае истинности условия необходимо выполнить несколько операторов, их можно заключить в фигурные скобки:

```
if   (x < 0) {
    x = -x;
    cout << "Изменить значение x на
       противоположное по знаку";
}
abs = x;
```

Теперь если x отрицательно, то не только его значение изменится на противоположное, но и будет выведено соответствующее сообщение. Фактически, заключая несколько операторов в фигурные скобки, мы сделали из них один сложный оператор или блок. Прием заключения нескольких операторов в блок работает везде, где нужно поместить несколько операторов вместо одного.

Условный оператор можно расширить для проверки нескольких условий:

```
if   (x  < 0)
    cout  << "Отрицательная величина";
else if   (x > 0)
```

```
    cout  << "Положительная величина";
else
    cout  << "Ноль";
```

Конструкций else if может быть несколько.

Хотя любые комбинации условий можно выразить с помощью оператора if, довольно часто запись становится неудобной и запутанной. Оператор выбора switch используется, когда для каждого из нескольких возможных значений выражения нужно выполнить определенные действия. Например, предположим, что в переменной code хранится целое число от 0 до 2, и нам нужно выполнить различные действия в зависимости от ее значения:

```
switch (code) {
case 0:
    cout << "код ноль";
    x = x + 1;
    break;
case 1 :
    cout << "код один";
    y = y + 1;
    break;
case 2:
    cout << "код два";
    z = z + 1;
    break;
default:
    cout << "Необрабатываемое значение";
}
```

В зависимости от значения code управление передается на одну из меток case. Выполнение оператора заканчивается по достижении либо оператора break, либо конца оператора switch. Таким образом, если code равно 1, выводится "код один", а затем переменная y увеличивается на единицу. Если бы после этого не стоял оператор break, то управление "провалилось" бы дальше, была бы выведена фраза "код два", и переменная z тоже увеличилась бы на единицу.

Если значение переключателя не совпадает ни с одним из значений меток

case, то выполняются операторы, записанные после метки default. Метка default может быть опущена, что эквивалентно записи:

```
default:
  ;  // пустой оператор, не выполняющий
     // никаких действий
```

Очевидно, что приведенный пример можно переписать с помощью оператора if:

```
if   (code == 0) {
     cout << "код ноль";
     x = x + 1;
} else if (code == 1) {
     cout << "код один";
     y = y + 1;
} else if (code == 2) {
     cout << "код два";
     z = z + 1;
} else {
     cout << "Необрабатываемое значение";
}
```

Пожалуй, запись с помощью оператора переключения switch более наглядна. Особенно часто переключатель используется, когда значение выражения имеет тип набора. :

4.6 Операторы цикла

Предположим, нам нужно вычислить сумму всех целых чисел от 0 до 100. Для этого воспользуемся оператором цикла for:

```
int sum = 0;
int i;
for (i = 1; i <= 100; i = i + 1)
                    // заголовок цикла
    sum = sum + i;      // тело цикла
```

Оператор цикла состоит из заголовка цикла и тела цикла. Тело цикла – это оператор, который будет повторно выполняться (в данном случае – увеличение значения переменной sum на величину переменной i). Заголовок

– это ключевое слово for, после которого в круглых скобках записаны три выражения, разделенные точкой с запятой. Первое выражение вычисляется один раз до начала выполнения цикла. Второе – это условие цикла. Тело цикла будет повторяться до тех пор, пока условие цикла истинно. Третье выражение вычисляется после каждого повторения тела цикла.

Оператор for реализует фундаментальный принцип вычислений в программировании – итерацию. Тело цикла повторяется для разных, в данном случае последовательных, значений переменной i. Повторение иногда называется итерацией. Мы как бы проходим по последовательности значений переменной i, выполняя с текущим значением одно и то же действие, тем самым постепенно вычисляя нужное значение. С каждой итерацией мы подходим к нему все ближе и ближе. С другим принципом вычислений в программировании – рекурсией – мы познакомимся в разделе, описывающем функции.

Любое из трех выражений в заголовке цикла может быть опущено (в том числе и все три). То же самое можно записать следующим образом:

```
int sum = 0;
int i = 1;
for (; i <= 100; ) {
    sum = sum + i;
    i = i + 1;
}
```

Заметим, что вместо одного оператора цикла мы записали несколько операторов, заключенных в фигурные скобки – блок. Другой вариант:

```
int sum = 0;
int i = 1;
for (; ;) {
    if (i > 100)
        break;
    sum = sum + i;
    i = i + 1;
}
```

В последнем примере мы опять встречаем оператор break. Оператор break завершает выполнение цикла. Еще одним вспомогательным оператором при выполнении циклов служит оператор продолжения continue. Оператор continue заставляет пропустить остаток тела цикла и перейти к следующей итерации (повторению). Например, если мы хотим найти сумму всех целых чисел от 0 до 100, которые не делятся на 7, можно записать это так:

```
int sum = 0;
for (int i = 1; i <= 100; i = i+1) {
    if ( i % 7 == 0)
        continue;
    sum = sum + i;
}
```

Еще одно полезное свойство цикла for: в первом выражении заголовка цикла можно объявить переменную. Эта переменная будет действительна только в пределах цикла.

Другой формой оператора цикла является оператор while. Его форма следующая:

```
while (условие)
    оператор
```

Условие – как и в условном операторе if – это выражение, которое принимает логическое значение "истина" или "ложь". Выполнение оператора повторяется до тех пор, пока значением условия является true (истина). Условие вычисляется заново перед каждой итерацией. Подсчитать, сколько десятичных цифр нужно для записи целого положительного числа N, можно с помощью следующего фрагмента:

```
int digits = 0;
while (N > 0) {
    digits = digits + 1;
    N = N / 10;
```

```
}
```

Если число N меньше либо равно нулю, тело цикла не будет выполнено.

Третьей формой оператора цикла является цикл do while. Он имеет форму:

```
do { операторы } while ( условие);
```

Отличие от предыдущей формы цикла while заключается в том, что условие проверяется после выполнения тела цикла. Предположим, требуется прочитать символы с терминала до тех пор, пока не будет введен символ "звездочка".

```
char ch;
do {
  ch = getch();  // функция getch возвращает
                 // символ, введёный с
                 // клавиатуры
} while (ch != '*');
```

В операторах while и do также можно использовать операторы break и continue.

Как легко заметить, операторы цикла взаимозаменяемы. Оператор while соответствует операторатору for:

```
for  ( ; условие ; )
    оператор
```

Пример чтения символов с терминала можно переписать в виде:

```
char ch;
ch = getch();
while (ch != '*') {
    ch = getch();
}
```

Разные формы нужны для удобства и наглядности записи.

4.7 Оператор возврата

Оператор return завершает выполнение функции и возвращает управление в ту точку, откуда она была вызвана. Его форма:

return выражение;

Где выражение – это результат функции. Если функция не возвращает никакого значения, то оператор возврата имеет форму

return;

4.8 Оператор перехода

Последовательность выполнения операторов в программе можно изменить с помощью оператора перехода goto. Он имеет вид:

goto метка;

Метка ставится в программе, записывая ее имя и затем двоеточие. Например, вычислить абсолютную величину значения переменной x можно следующим способом:

```
if ( x >= 0)
  goto positiv;
x = -x;        // переменить знак x
positiv:       // объявление метки
abs = x;       // присвоить переменной abs
               // положительное значение
```

При выполнении goto вместо следующего оператора выполняется оператор, стоящий после метки positiv. Если значение x положительное, оператор x = - x выполняться не будет.

В настоящее время считается, что оператор goto очень легко запутывает программу.Без него, вообще говоря, можно обойтись, поэтому лучше его не использовать, ну разве что лишь в самом крайнем случае.

Пример:

```
int fact(int n)
{
    int k;
    if (n == 1) {
        k = 1;
    } else {
        k = n * fact(n – 1);
    }
    return k;
}
```

Это функция вычисления факториала. Первый оператор в ней – это объявление переменной k, в которой будет храниться результат вычисления. Затем выполняется условный оператор if. Если n равно единице, то вычисления факториала закончены, и выполняется оператор-выражение, который присваивает переменной значение 1. В противном случае выполняется другой оператор-выражение.

Последний оператор – это оператор возврата из функции.

Функции

5.1 Вызов функций

Функция вызывается при вычислении выражений. При вызове ей передаются определенные аргументы, функция выполняет необходимые действия и возвращает результат.

Программа на языке Си++ состоит, по крайней мере, из одной функции – функции main. С нее всегда начинается выполнение программы. Встретив имя функции в выражении, программа вызовет эту функцию, т.е. передаст управление на ее начало и начнет выполнять операторы. Достигнув конца функции или оператора return – выхода из функции, управление вернется в ту точку, откуда функция была вызвана, подставив вместо нее вычисленный результат.

Прежде всего, функцию необходимо объявить. Объявление функции, аналогично объявлению переменной, определяет имя функции и ее тип – типы и количество ее аргументов и тип возвращаемого значения.

```
// функция sqrt с одним аргументом –
// вещественным числом двойной точности,
//  возвращает результат типа double

double sqrt(double x);
// функция sum от трех целых аргументов
// возвращает целое число
int sum(int a, int b, int c);
```

Объявление функции называют иногда прототипом функции. После того, как функция объявлена, ее можно использовать в выражениях:

```
double x = sqrt(3) + 1;
sum(k, l, m) / 15;
```

Если функция не возвращает никакого результата, т.е. она объявлена как void, ее вызов не может быть использован как операнд более сложного выражения, а должен быть записан сам по себе:

```
func(a,b,c);
```

Определение функции описывает, как она работает, т.е. какие действия

надо выполнить, чтобы получить искомый результат. Для функции sum, объявленной выше, определение может выглядеть следующим образом:

```
int
sum(int a, int b, int c)
{
    int result;
    result = a + b + c;
    return result;
}
```

Первая строка – это заголовок функции, он совпадает с объявлением функции, за исключением того, что объявление заканчивается точкой с запятой. Далее в фигурных скобках заключено тело функции – действия, которые данная функция выполняет.

Аргументы a, b и c называются формальными параметрами. Это переменные, которые определены в теле функции (т.е. к ним можно обращаться только внутри фигурных скобок). При написании определения функции программа не знает их значения. При вызове функции вместо них подставляются фактические параметры – значения, с которыми функция вызывается. Выше, в примере вызова функции sum, фактическими параметрами (или фактическими аргументами) являлись значения переменных k, l и m.

Формальные параметры принимают значения фактических аргументов, заданных при вызове, и функция выполняется.

Первое, что мы делаем в теле функции — объявляем внутреннюю переменную result типа целое. Переменные, объявленные в теле функции, также называют локальными. Это связано с тем, что переменная result существует только во время выполнения тела функции sum. После завершения выполнения функции она уничтожается – ее имя становится неизвестным, и память, занимаемая этой переменной, освобождается.

Вторая строка определения тела функции – вычисление результата. Сумма всех аргументов присваивается переменной result. Отметим, что до присваивания значение result было неопределенным (то есть значение

переменной было неким произвольным числом, которое нельзя определить заранее).

Последняя строчка функции возвращает в качестве результата вычисленное значение. Оператор return завершает выполнение функции и возвращает выражение, записанное после ключевого слова return, в качестве выходного значения. В следующем фрагменте программы переменной s присваивается значение 10:

```
int k = 2;
int l = 3;
int m = 5;
int s = sum(k, l, m);
```

5.2 Имена функций

В языке Си++ допустимо иметь несколько функций с одним и тем же именем, потому что функции различаются не только по именам, но и по типам аргументов. Если в дополнение к определенной выше функции sum мы определим еще одну функцию с тем же именем

```
double
sum(double a, double b, double c)
{
    double result;
    result = a + b + c;
    return result;
}
```

это будет считаться новой функцией. Иногда говорят, что у этих функций разные подписи. В следующем фрагменте программы в первый раз будет вызвана первая функция, а во второй раз – вторая:

```
int x, y, z, ires;
double p,q,s, dres;
. . .
// вызвать первое определение функции sum
ires = sum(x,y,z);
// вызвать второе определение функции sum
dres = sum(p,q,s);
```

При первом вызове функции sum все фактические аргументы имеют

тип int. Поэтому вызывается первая функция. Во втором вызове все аргументы имеют тип double, соответственно, вызывается вторая функция.

Важен не только тип аргументов, но и их количество. Можно определить функцию sum, суммирующую четыре аргумента:

```
int
sum(int x1, int x2, int x3, int x4)
{
    return x1 + x2 + x3 + x4;
}
```

Отметим, что при определении функций имеют значение тип и количество аргументов, но не тип возвращаемого значения. Попытка определения двух функций с одним и тем же именем, одними и теми же аргументами, но разными возвращаемыми значениями, приведет к ошибке компиляции:

```
int foo(int x);
double foo(int x);
// ошибка – двукратное определение имени
```

5.3 Необязательные аргументы функций

При объявлении функций в языке Си++ имеется возможность задать значения аргументов по умолчанию. Первый случай применения этой возможности языка – сокращение записи. Если функция вызывается с одним и тем же значением аргумента в 99% случаев, и это значение достаточно очевидно, можно задать его по умолчанию. Предположим, функция expnt возводит число в произвольную целую положительную степень. Чаще всего она используется для возведения в квадрат. Ее объявление можно записать так:

```
double expnt (double x, unsigned int e = 2);
```

Определение функции:

```
double
expnt (double x, unsigned int e)
{
    double result = 1;
```

```
    for (int i = 0; i < e; i++)
        result *= x;
    return result;
}
int main()
{
    double y = expnt(3.14);
    double x = expnt(2.9, 5);
    return 1;
}
```

Использовать аргументы по умолчанию удобно при изменении функции. Если при изменении программы нужно добавить новый аргумент, то для того чтобы не изменять все вызовы этой функции, можно новый аргумент объявить со значением по умолчанию. В таком случае старые вызовы будут использовать значение по умолчанию, а новые – значения, указанные при вызове.

Необязательных аргументов может быть несколько. Если указан один необязательный аргумент, то либо он должен быть последним в прототипе, либо все аргументы после него должны также иметь значение по умолчанию.

Если для функции задан необязательный аргумент, то фактически задано несколько подписей этой функции. Например, попытка определения двух функций

```
double expnt (double x, unsigned int e = 2);
double expnt (double x);
```

приведет к ошибке компиляции – неоднозначности определения функции. Это происходит потому, что вызов

```
double x = expnt(4.1);
```

подходит как для первой, так и для второй функции.

5.4 Рекурсия

Определения функций не могут быть вложенными, т.е. нельзя внутри тела одной функции определить тело другой. Разумеется, можно вызвать одну функцию из другой. В том числе функция может вызвать сама себя.

Рассмотрим функцию вычисления факториала целого числа. Ее можно реализовать двумя способами. Первый способ использует итерацию:

```
int
fact(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
        result = result * i;
    return result;
}
```

Второй способ:

```
int
fact(int n)
{
  if (n == 1)     // факториал 1 равен 1
        return 1;
  else           // факториал числа n равен
                 // факториалу n-1
                 // умноженному на n

        return n * fact(n -1);
}
```

Функция fact вызывает сама себя с модифицированными аргументами. Такой способ вычислений называется рекурсией. Рекурсия – это очень мощный метод вычислений. Значительная часть математических функций определяется в рекурсивных терминах. В программировании алгоритмы обработки сложных структур данных также часто бывают рекурсивными. Рассмотрим, например, структуру двоичного дерева. Дерево состоит из узлов и направленных связей. С каждым узлом могут быть связаны один или два узла, называемые сыновьями этого узла. Соответственно, для "сыновей" узел, из которого к ним идут связи, называется "отцом". Узел, у которого нет "отца", называется корнем. У дерева есть только один корень. Узлы, у которых нет "сыновей", называются листьями. Пример дерева приведен на рис. 5.1.

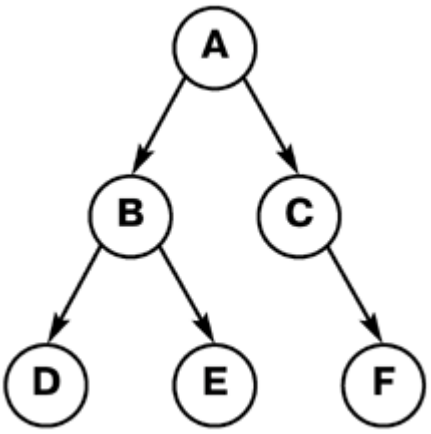

Рис. 5.1. Пример дерева.

В этом дереве узел A – корень дерева, узлы B и C – "сыновья" узла A, узлы D и E – "сыновья" узла B, узел F – "сын" узла C. Узлы D, E и F – листья. Узел B является корнем поддерева, состоящего из трех узлов B, D и E. Обход дерева (прохождение по всем его узлам) можно описать таким образом:

1. Посетить корень дерева.
2. Обойти поддеревья с корнями — "сыновьями" данного узла, если у узла есть "сыновья".
3. Если у узла нет "сыновей" — обход закончен.

Очевидно, что реализация такого алгоритма с помощью рекурсии не составляет труда.

Довольно часто рекурсия и итерация взаимозаменяемы (как в примере с факториалом). Выбор между ними может быть обусловлен разными факторами. Чаще рекурсия более наглядна и легче реализуется. Кроме того, в большинстве случаев итерация более эффективна.

6 Встроенные типы данных

6.1 Общая информация

Встроенные типы данных предопределены в языке. Это самые простые величины, из которых составляют все производные типы, в том числе и классы. Различные реализации и компиляторы могут определять различные диапазоны значений целых и вещественных чисел.

В таблице 6.1 перечислены простейшие типы данных, которые определяет язык Си++, и приведены наиболее типичные диапазоны их значений.

Таблица 6.1. Встроенные типы языка Си++.

Название	Обозначение	Диапазон значений
Байт	**char**	от -128 до +127
Байт без знака	**unsigned char**	от 0 до 255
Короткое целое число	**short**	от -32768 до +32767
Короткое целое число без знака	**unsigned short**	от 0 до 65535
Целое число	**int**	от – 2147483648 до + 2147483647
Целое число без знака	**unsigned int** (или просто **unsigned**)	от 0 до 4294967295
Длинное целое число	**long**	от – 2147483648 до + 2147483647

Длинное целое число	**unsigned long**	от 0 до 4294967295
Вещественное число одинарной точности	**float**	от ±3.4e-38 до ±3.4e+38 (7 значащих цифр)
Вещественное число двойной точности	**double**	от ±1.7e-308 до ±1.7e+308 (15 значащих цифр)
Вещественное число увеличенной точности	**long double**	от ±1.2e-4932 до ±1.2e+4932
Логическое значение	**bool**	значения true (истина) или false (ложь)

6.2 Целые числа

Для представления целых чисел в языке Си++ существует несколько типов – char, short int и long (полное название типов: short int, long int, unsigned long int и т.д.. Поскольку описатель int можно опустить, мы используем сокращенные названия). Они отличаются друг от друга диапазоном возможных значений. Каждый из этих типов может быть знаковым или беззнаковым. По умолчанию, тип целых величин – знаковый. Если перед определением типа стоит ключевое слово unsigned , то тип целого числа — беззнаковый. Для того чтобы определить переменную x типа короткого целого числа, нужно записать:

```
short x;
```

Число без знака принимает только положительные значения и значение ноль. Число со знаком принимает положительные значения, отрицательные значения и значение ноль.

Целое число может быть непосредственно записано в программе в виде

константы. Запись чисел соответствует общепринятой нотации. Примеры целых констант: 0, 125, -37. По умолчанию целые константы принадлежат к типу int. Если необходимо указать, что целое число — это константа типа long , можно добавить символ L или l после числа. Если константа беззнаковая, т.е. относится к типу unsigned long или unsigned int , после числа записывается символ U или u. Например: 34U, 700034L, 7654ul.

Кроме стандартной десятичной записи, числа можно записывать в восьмеричной или шестнадцатеричной системе счисления. Признаком восьмеричной системы счисления является цифра 0 в начале числа. Признаком шестнадцатеричной — 0x или 0X перед числом. Для шестнадцатеричных цифр используются латинские буквы от A до F (неважно, большие или маленькие).

Таким образом, фрагмент программы

```
const int x = 240;
int y = 0360;
const int z = 0xF0;
```

определяет три целые константы x, y и z с одинаковыми значениями.

Отрицательные числа предваряются знаком минус "-". Приведем еще несколько примеров:

```
  // ошибка в записи восьмеричного числа
const usigned long ll = 0678;
  // правильная запись
const short a = 0xa4;
  // ошибка в записи десятичного числа
const int x = 23F3;
```

Для целых чисел определены стандартные арифметические операции сложения (+), вычитания (-), умножения (*), деления (/); нахождение остатка от деления (%), изменение знака (-). Результатом этих операций также является целое число. При делении остаток отбрасывается. Примеры выражений с целыми величинами:

```
x + 4;
30 — x;
x * 2;
-x;
10 / x;
x % 3;
```

Кроме стандартных арифметических операций, для целых чисел определен набор битовых (или поразрядных) операций. В них целое число рассматривается как строка битов (нулей и единиц при записи числа в двоичной системе счисления или разрядов машинного представления).

К этим операциям относятся поразрядные операции И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ, поразрядное отрицание и сдвиги. Поразрядная операция ИЛИ, обозначаемая знаком |, выполняет операцию ИЛИ над каждым индивидуальным битом двух своих операндов. Например, 1 | 2 в результате дают 3, поскольку в двоичном виде 1 это 01, 2 – это 10, соответственно, операция ИЛИ дает 11 или 3 в десятичной системе (нули слева мы опустили).

Аналогично выполняются поразрядные операции И, ИСКЛЮЧАЮЩЕЕ ИЛИ и отрицание.

```
3 | 1       результат     3
4 & 7       результат     4
4 ^ 7       результат     3
0 & 0xF     результат     0
~0x00F0     результат     0xFF0F
```

Операция сдвига перемещает двоичное представление левого операнда на количество битов, соответствующее значению правого операнда. Например, двоичное представление короткого целого числа 3 – 0000000000000011. Результатом операции 3 << 2 (сдвиг влево) будет двоичное число 0000000000001100 или, в десятичной записи, 12. Аналогично, сдвинув число 9 (в двоичном виде 0000000000001001) на 2 разряда вправо (записывается 9 >> 2) получим 0000000000000010, т.е. 2.

При сдвиге влево число дополняется нулями справа. При сдвиге вправо

бит, которым дополняется число, зависит от того, знаковое оно или беззнаковое. Для беззнаковых чисел при сдвиге вправо они всегда дополняются нулевым битом. Если же число знаковое, то значение самого левого бита числа используется для дополнения. Это объясняется тем, что самый левый бит как раз и является знаком — 0 означает плюс и 1 означает минус. Таким образом, если

```
short x = 0xFF00;
unsigned short y = 0xFF00;
```

то результатом x >> 2 будет 0xFFC0 (двоичное представление 1111111111000000), а результатом y >> 2 будет 0x3FC0 (двоичное представление 0011111111000000).

Рассмотренные арифметические и поразрядные операции выполняются над целыми числами и в результате дают целое число. В отличие от них операции сравнения выполняются над целыми числами, но в результате дают логическое значение истина (true) или ложь (false).

Для целых чисел определены операции сравнения: равенства (==), неравенства (!=), больше (>), меньше (<), больше или равно (>=) и меньше или равно (<=).

Последний вопрос, который мы рассмотрим в отношении целых чисел, – это преобразование типов. В языке Си++ допустимо смешивать в выражении различные целые типы. Например, вполне допустимо записать x + y, где x типа short , а y – типа long . При выполнении операции сложения величина переменной x преобразуется к типу long . Такое преобразование можно произвести всегда, и оно безопасно, т.е. мы не теряем никаких значащих цифр. Общее правило преобразования целых типов состоит в том, что более короткий тип при вычислениях преобразуется в более длинный. Только при выполнении присваивания длинный тип может преобразовываться в более короткий. Например:

```
short x;
```

```
long y = 15;
. . .
x = y;
// преобразование длинного типа
// в более короткий
```

Такое преобразование не всегда безопасно, поскольку могут потеряться значащие цифры. Обычно компиляторы, встречая такое преобразование, выдают предупреждение или сообщение об ошибке.

6.3 Вещественные числа

Вещественные числа в Си++ могут быть одного из трех типов: с одинарной точностью — float , с двойной точностью – double , и с расширенной точностью – long double.

```
float x;
double e = 2.9;
long double s;
```

В большинстве реализаций языка представление и диапазоны значений соответствуют стандарту IEEE (Institute of Electrical and Electronics Engineers) для представления вещественных чисел. Точность представления чисел составляет 7 десятичных значащих цифр для типа float , 15 значащих цифр для double и 19 — для типа long double .

Вещественные числа записываются либо в виде десятичных дробей, например 1.3, 3.1415, 0.0005, либо в виде мантиссы и экспоненты: 1.2E0, 0.12e1. Отметим, что обе предыдущие записи изображают одно и то же число 1.2.

По умолчанию вещественная константа принадлежит к типу double . Чтобы обозначить, что константа на самом деле float , нужно добавить символ f или F после числа: 2.7f. Символ l или L означает, что записанное число относится к типу long double .

```
const float pi_f = 3.14f;
double pi_d = 3.1415;
long double pi_l = 3.1415L;
```

Для вещественных чисел определены все стандартные арифметические операции сложения (+), вычитания (-), умножения (*), деления (/) и изменения знака (-). В отличие от целых чисел, операция нахождения остатка от деления для вещественных чисел не определена. Аналогично, все битовые операции и сдвиги к вещественным числам неприменимы; они работают только с целыми числами. Примеры операций:

```
2 * pi;
(x – e) / 4.0
```

Вещественные числа можно сравнивать на равенство (==), неравенство (!=), больше (>), меньше (<), больше или равно (>=) и меньше или равно (<=). В результате операции сравнения получается логическое значение истина или ложь.

Если арифметическая операция применяется к двум вещественным числам разных типов, то менее точное число преобразуется в более точное, т.е. float преобразуется в double и double преобразуется в long double . Очевидно, что такое преобразование всегда можно выполнить без потери точности.

Если вторым операндом в операции с вещественным числом является целое число, то целое число преобразуется в вещественное представление.

Хотя любую целую величину можно представить в виде вещественного числа, при таком преобразовании возможна потеря точности (для больших чисел).

6.4 Логические величины

В языке Си++ существует специальный тип для представления логических значений bool . Для величин этого типа существует только два возможных значения: true (истина) и false (ложь). Объявление логической переменной выглядит следующим образом:

```
bool condition;
```

Соответственно, существуют только две логические константы – истина и ложь. Они обозначаются, соответственно, true и false .

Для типа bool определены стандартные логические операции: логическое И (&&), ИЛИ (||) и НЕ (!).

```
// истинно, если обе переменные,
// cond1 и cond2, истинны
cond1 && cond2
// истинно, если хотя бы одна из переменных
// истинна
cond1 || cond2
// результат противоположен значению cond1
!cond
```

Как мы уже отмечали ранее, логические значения получаются в результате операций сравнения. Кроме того, в языке Си++ принято следующее правило преобразования чисел в логические значения: ноль соответствует значению false , и любое отличное от нуля число преобразуется в значение true . Поэтому можно записать, например:

```
int k = 100;
while (k) {     // выполнить цикл 100 раз
    k--;
}
```

6.3 Символы и байты

Символьный или байтовый тип в языке Си++ относится к целым числам, однако мы выделили их в особый раздел, потому что запись знаков имеет свои отличия.

Итак, для записи знаков в языке Си++ служат типы char и unsigned char . Первый – это целое число со знаком, хранящееся в одном байте, второй – беззнаковое байтовое число. Эти типы чаще всего используются для манипулирования символами, поскольку коды символов как раз помещаются в байт.

Пояснение. Единственное, что может хранить компьютер, это числа. Поэтому для того чтобы можно было хранить символы и манипулировать

ими, символам присвоены коды – целые числа. Существует несколько стандартов, определяющих, какие коды каким символам соответствуют. Для английского алфавита и знаков препинания используется стандарт ASCII. Этот стандарт определяет коды от 0 до 127. Для представления русских букв используется стандарт КОИ-8 или CP-1251. В этих стандартах русские буквы кодируются числами от 128 до 255. Таким образом, все символы могут быть представлены в одном байте (максимальное число символов в одном байте – 255). Для работы с китайским, японским, корейским и рядом других алфавитов одного байта недостаточно, и используется кодировка с помощью двух байтов и, соответственно, тип wchar_t (подробнее см. ниже).

Чтобы объявить переменную байтового типа, нужно записать:

```
char c;
// байтовое число со знаком

unsigned char u;
// байтовое число без знака
```

Поскольку байты – это целые числа, то все операции с целыми числами применимы и к байтам. Стандартная запись целочисленных констант тоже применима к байтам, т.е. можно записать:

```
c = 45;
```

где c — байтовая переменная. Однако для байтов существует и другая запись констант. Знак алфавита (буква, цифра, знак препинания), заключенный в апострофы, представляет собой байтовую константу, например:

```
'S' '&' '8' 'ф'
```

Числовым значением такой константы является код данного символа, принятый в Вашей операционной системе.

В кодировке ASCII два следующих оператора эквивалентны:

```
char c = 68;
```

```
char c = 'D';
```

Первый из них присваивает байтовой переменной c значение числа 68. Второй присваивает этой переменной код латинской буквы D, который в кодировке ASCII равен 68.

Для обозначения ряда непечатных символов используются так называемые экранированные последовательности – знак обратной дробной черты, после которого стоит буква. Эти последовательности стандартны и заранее предопределены в языке:

\a звонок
\b возврат на один символ назад
\f перевод страницы
\n новая строка
\r перевод каретки
\t горизонтальная табуляция
\v вертикальная табуляция
\' апостроф
\" двойные кавычки
\\ обратная дробная черта
\? вопросительный знак

Для того чтобы записать произвольное байтовое значение, также используется экранированная последовательность: после обратной дробной черты записывается целое число от 0 до 255.

```
char zero = '\0';
const unsigned char bitmask = '\0xFF';
char tab = '\010';
```

Следующая программа выведет все печатные символы ASCII и их коды в порядке увеличения:

```
for (char c = 32; c < 127; c++)
    cout < c < " " < (int)c < " ";
```

Однако напомним еще раз, что байтовые величины – это, прежде всего, целые числа, поэтому вполне допустимы выражения вида

```
'F' + 1
'a' < 23
```

и тому подобные. Тип char был придуман для языка Си, от которого Си++

достались все базовые типы данных. Язык Си предназначался для программирования на достаточно "низком" уровне, приближенном к тому, как работает процессор ЭВМ, именно поэтому символ в нем – это лишь число.

В языке Си++ в большинстве случаев для работы с текстом используются специально разработанные классы строк, о которых мы будем говорить позже.

6.4 Кодировка, многобайтовые символы

Мы уже упоминали о наличии разных кодировок букв, цифр, знаков препинания и т.д. Алфавит большинства европейских языков может быть представлен однобайтовыми числами (т.е. кодами в диапазоне от 0 до 255). В большинстве кодировок принято, что первые 127 кодов отводятся для символов, входящих в набор ASCII: ряд специальных символов, латинские заглавные и строчные буквы, арабские цифры и знаки препинания. Вторая половина кодов – от 128 до 255 отводится под буквы того или иного языка. Фактически, вторая половина кодовой таблицы интерпретируется по-разному, в зависимости от того, какой язык считается "текущим". Один и тот же код может соответствовать разным символам в зависимости от того, какой язык считается "текущим".

Однако для таких языков, как китайский, японский и некоторые другие, одного байта недостаточно – алфавиты этих языков насчитывают более 255 символов.

Перечисленные выше проблемы привели к созданию многобайтовых кодировок символов. Двухбайтовые символы в языке Си++ представляются с помощью типа wchar_t :

```
wchar_t wch;
```

Тип wchar_t иногда называют расширенным типом символов, и детали его реализации могут варьироваться от компилятора к компилятору, в том числе может меняться и количество байт, которое отводится под один символ. Тем не менее, в большинстве случаев используется именно

двухбайтовое представление.

Константы типа wchar_t записываются в виде L'ab'.

6.5 Наборы перечисляемых значений

Достаточно часто в программе вводится тип, состоящий лишь из нескольких заранее известных значений. Например, в программе используется переменная, хранящая величину, отражающую время суток, и мы решили, что будем различать ночь, утро, день и вечер. Конечно, можно договориться обозначить время суток числами от 1 до 4. Но, во-первых, это не наглядно. Во-вторых, что даже более существенно, очень легко сделать ошибку и,например, использовать число 5, которое не соответствует никакому времени дня. Гораздо удобней и надежнее определить набор значений с помощью типа enum языка Си++:

```
enum DayTime { morning, day, evening, night };
```

Теперь можно определить переменную

```
DayTime current;
```

которая хранит текущее время дня, а затем присваивать ей одно из допустимых значений типа DayTime:

```
current = day;
```

Контроль, который осуществляет компилятор при использовании в программе этой переменной, гораздо более строгий, чем при использовании целого числа.

Для наборов определены операции сравнения на равенство (==) и неравенство (!=) с атрибутами этого же типа, т.е.

```
if (current != night)
    // выполнить работу
```

Вообще говоря, внутреннее представление значений набора – целые числа. По умолчанию элементам набора соответствуют последовательные целые числа, начиная с 0. Этим можно пользоваться в программе. Во-первых, можно задать, какое число какому атрибуту набора будет соответствовать:

```
enum { morning = 4, day = 3, evening = 2,
          night = 1 };
    // последовательные числа начиная с 1
enum { morning = 1, day, evening, night };
    // используются числа 0, 2, 3 и 4
enum { morning, day = 2, evening, night };
```

Во-вторых, атрибуты наборов можно использовать в выражениях вместо целых чисел. Преобразования из набора в целое и наоборот разрешены.

Однако мы не рекомендуем так делать. Для работы с целыми константами лучше применять символические обозначения констант, а наборы использовать по их прямому назначению.

7 Классы и объекты

7.1 Понятие класса

До сих пор мы говорили о встроенных типах, т.е. типах, определенных в самом языке. Классы - это типы, определенные в конкретной программе. Определение класса включает в себя описание, из каких составных частей или атрибутов он состоит и какие операции определены для класса.

Предположим, в программе необходимо оперировать комплексными числами. Комплексные числа состоят из вещественной и мнимой частей, и с ними можно выполнять арифметические операции.

```
class Complex {
public:
    int real;   // вещественная часть
    int imaginary;   // мнимая часть
    void Add(Complex x);
        // прибавить комплексное число
};
```

Приведенный выше пример - упрощенное определение класса Complex, представляющее комплексное число. Комплексное число состоит из вещественной части - целого числа real и мнимой части, которая представлена целым числом imaginary. real и imaginary - это атрибуты класса. Для класса Complex определена одна операция или метод - Add.

Определив класс, мы можем создать переменную типа Complex:

```
Complex number;
```

Переменная с именем number содержит значение типа Complex, то есть содержит объект класса Complex. Имея объект, мы можем установить значения атрибутов объекта:

```
number.real = 1;
number.imaginary = 2;
```

Операция "." обозначает обращение к атрибуту объекта. Создав еще один объект класса Complex, мы можем прибавить его к первому:

```
Complex num2;
number.Add(num2);
```

Как можно заметить, метод Add выполняется с объектом. Имя объекта (или переменной, содержащей объект, что, в сущности, одно и то же), в данном случае, number, записано первым. Через точку записано имя метода - Add с аргументом - значением другого объекта класса Complex, который прибавляется к number. Методы часто называются сообщениями. Но чтобы послать сообщение, необходим получатель. Таким образом, объекту number посылается сообщение Add с аргументом num2. Объект number принимает это сообщение и складывает свое значение со значением аргумента сообщения.

7.2 Определение методов класса

Данные рассуждения будут яснее, если мы определим, как выполняется операция сложения.

```
void
Complex::Add(Complex x)
{
  this->real = this->real + x.real;
  this->imaginary = this->imaginary +
                x.imaginary;
}
```

Первые две строки говорят о том, что это метод Add класса Complex. В фигурных скобках записано определение операции или метода Add. Это определение означает следующее: для того чтобы прибавить значение объекта класса Complex к данному объекту, надо сложить вещественные части и запомнить результат в атрибуте вещественной части текущего объекта. Точно так же следует сложить мнимые части двух комплексных чисел и запомнить результат в атрибуте текущего объекта, обозначающем мнимую часть.

Запись this-> говорит о том, что атрибут принадлежит к тому объекту, который выполняет метод Add (объекту, получившему сообщение Add). В большинстве случаев this-> можно опустить. В записи определения метода какого-либо класса упоминание атрибута класса без всякой

дополнительной информации означает, что речь идет об атрибуте текущего объекта.

Теперь приведем этот небольшой пример полностью:

```
// определение класса комплексных чисел
class Complex {
public:
    int real; // вещественная часть
    int imaginary; // мнимая часть
    void Add(Complex x);
      // прибавить комплексное число
};
// определение метода сложения
void
Complex::Add(Complex x)
{
    real = real + x.real;
    imaginary = imaginary + x.imaginary;
}
int
main()
{
Complex number;
    number.real = 1;
          // первый объект класса Complex
    number.imaginary = 3;
    Complex num2;
          // второй объект класса Complex
    num2.real = 2;
    num2.imaginary = 1;
    number.Add(num2);
          // прибавить значение второго
          // объекта к первому
    return 1;
}
```

7.4 Переопределение операций

В языке Си++ можно сделать так, что класс будет практически неотличим от предопределенных встроенных типов при использовании в выражениях. Для класса можно определить операции сложения, умножения и т.д. пользуясь стандартной записью таких операций, т.е. x + y. В языке Си++ считается, что подобная запись - это также вызов метода с именем

operator+ того класса, к которому принадлежит переменная x. Перепишем определение класса Complex:

```
// определение класса комплексных чисел
class Complex
{
public:
  int real; // вещественная часть
  int imaginary; // мнимая часть
  // прибавить комплексное число
  Complex operator+(const Complex x) const;
};
```

Вместо метода Add появился метод operator+. Изменилось и его определение. Во-первых, этот метод возвращает значение типа Complex (операция сложения в результате дает новое значение того же типа, что и типы операндов). Во-вторых, перед аргументом метода появилось ключевое слово const. Это слово обозначает, что при выполнении данного метода аргумент изменяться не будет. Также const появилось после объявление метода. Второе ключевое слово const означает, что объект, выполняющий метод, не будет изменен. При выполнении операции сложения x + y над двумя величинами x и y сами эти величины не изменяются. Теперь запишем определение операции сложения:

```
Complex
Complex::operator+(const Complex x) const
{
  Complex result;
  result.real = real + x.real;
  result.imaginary = imaginary + x.imaginary;
  return result;
}
```

7.5 Подписи методов и необязательные аргументы

Как и при объявлении функций, язык Си++ допускает определение в одном классе нескольких методов с одним и тем же именем, но разными типами и количеством аргументов. (Определение методов или атрибутов с одинаковыми именами в разных классах не вызывает проблем, поскольку

пространства имен разных классов не пересекаются).

```
// определение класса комплексных чисел
class Complex
{
public:
  int real; // вещественная часть
  int imaginary; // мнимая часть
  // прибавить комплексное число
  Complex operator+(const Complex x) const;
  // прибавить целое число
  Complex operator+(long x) const;
};
```

В следующем примере вначале складываются два комплексных числа, и вызывается первая операция +. Затем к комплексному числу прибавляется целое число, и тогда выполняется вторая операция сложения.

```
Complex c1;
Complex c2;
long x;
c1 + c2;
c2 + x;
```

Аналогично можно задавать значения аргументов методов по умолчанию. Более подробное описание можно найти в лекции 5.

7.6 Запись классов

Как уже отмечалось раньше, выбор имен - это не праздный вопрос. Существует множество систем именования классов. Опишем ту, которой мы придерживаемся в данной книге.

Имена классов, их методов и атрибутов составляются из английских слов, описывающих их смысл, при этом если слов несколько, они пишутся слитно. Имена классов начинаются с заглавной буквы; если название состоит из нескольких слов, каждое слово начинается с заглавной буквы, остальные буквы маленькие:

Complex, String, StudentLibrarian

Имена методов классов также начинаются с большой буквы:

Add, Concat

Имена атрибутов класса начинаются с маленькой буквы, однако если имя состоит из нескольких слов, последующие слова начинаются с большой:

real, classElement

При записи определения класса мы придерживаемся той же системы расположения, что и при записи функций. Ключевое слово class и имя класса записываются в первой строке, открывающаяся фигурная скобка - на следующей строке, методы и атрибуты класса - на последующих строках с отступом.

8 Производные типы данных

8.1 Массивы

Массив – это коллекция нескольких величин одного и того же типа. Простейшим примером массива может служить набор из двенадцати целых чисел, соответствующих числу дней в каждом календарном месяце:

```
int days[12];
days[0] = 31;    // январь
days[1] = 28;    // февраль
days[2] = 31;    // март
days[3] = 30;    // апрель
days[4] = 31;    // май
days[5] = 30;    // июнь
days[6] = 31;    // июль
days[7] = 31;    // август
days[8] = 30;    // сентябрь
days[9] = 31;    // октябрь
days[10] = 30;   // ноябрь
days[11] = 31;   // декабрь
```

В первой строчке мы объявили массив из 12 элементов типа int и дали ему имя days. Остальные строки примера – присваивания значений элементам массива. Для того, чтобы обратиться к определенному элементу массива, используют операцию индексации []. Как видно из примера, первый элемент массива имеет индекс 0, соответственно, последний – 11.

При объявлении массива его размер должен быть известен в момент компиляции, поэтому в качестве размера можно указывать только целую константу. При обращении же к элементу массива в роли значения индекса может выступать любая переменная или выражение, которое вычисляется во время выполнения программы и преобразуется к целому значению.

Предположим, мы хотим распечатать все элементы массива days. Для этого удобно воспользоваться циклом for.

```
for (int i = 0; i < 12; i++) {
    cout << days[i];
}
```

Следует отметить, что при выполнении программы границы массива не

контролируются. Если мы ошиблись и вместо 12 в приведенном выше цикле написали 13, то компилятор не выдаст ошибку. При выполнении программа попытается напечатать 13-е число. Что при этом случится, вообще говоря, не определено. Быть может, произойдет сбой программы. Более вероятно, что будет напечатано какое-то случайное 13-е число. Выход индексов за границы массива – довольно распространенная ошибка, которую иногда очень трудно обнаружить. В дальнейшем при изучении классов мы рассмотрим, как можно переопределить операцию [] и добавить контроль за индексами.

Отсутствие контроля индексов налагает на программиста большую ответственность. С другой стороны, индексация – настолько часто используемая операция, что наличие контроля, несомненно, повлияло бы на производительность программ.

Рассмотрим еще один пример. Предположим, что имеется массив из 100 целых чисел, и его необходимо отсортировать, т.е. расположить в порядке возрастания. Сортировка методом "пузырька" – наиболее простая и распространенная – будет выглядеть следующим образом:

```
int array[100];
. . .
for (int i = 0; i < 99; i++ ) {
    for (int j = i + 1; j < 100; j++) {
        if (array[j] < array[i] ) {
            int tmp = array[j];
            array[j] = array[i];
            array[i] = tmp;
        }
    }
}
```

В приведенных примерах у массивов имеется только один индекс. Такие одномерные массивы часто называются векторами. Имеется возможность определить массивы с несколькими индексами или **размерностями**. Например, объявление

```
int m[10][5];
```

представляет матрицу целых чисел размером 10 на 5. По-другому

интерпретировать приведенное выше объявление можно как массив из 10 элементов, каждый из которых – вектор целых чисел длиной 5. Общее количество целых чисел в массиве m равно 50.

Обращение к элементам многомерных массивов аналогично обращению к элементам векторов: m[1][2] обращается к третьему элементу второй строки матрицы m.

Количество размерностей в массиве может быть произвольным. Как и в случае с вектором, при объявлении многомерного массива все его размеры должны быть заданы константами.

При объявлении массива можно присвоить начальные значения его элементам (**инициализировать** массив). Для вектора это будет выглядеть следующим образом:

```
int days[12] = { 31, 28, 31, 30, 31, 31,
                 30, 31, 30, 31 };
```

При инициализации многомерных массивов каждая размерность должна быть заключена в фигурные скобки:

```
double temp[2][3] = {
    { 3.2, 3.3, 3.4 },
    { 4.1, 3.9, 3.9 } };
```

Интересной особенностью инициализации многомерных массивов является возможность не задавать размеры всех измерений массива, кроме самого последнего. Приведенный выше пример можно переписать так:

```
double temp[][3] = {
    { 3.2, 3.3, 3.4 },
    { 4.1, 3.9, 3.9 } };
// Вычислить размер пропущенной размерности
const int size_first = sizeof (temp) / sizeof
                       (double[3]);
```

8.2 Структуры

Структуры – это не что иное, как классы, у которых разрешен доступ ко всем их элементам (доступ к определенным атрибутам класса может быть ограничен, о чем мы узнаем в лекции 11). Пример структуры:

```
struct Record {
    int number;
    char name[20];
};
```

Так же, как и для классов, операция "." обозначает обращение к элементу структуры.

В отличие от классов, можно определить переменную-структуру без определения отдельного типа:

```
struct {
    double x;
    double y;
} coord;
```

Обратиться к атрибутам переменной coord можно coord.x и coord.y.

8.3 Битовые поля

В структуре можно определить размеры атрибута с точностью до бита. Традиционно структуры используются в системном программировании для описания регистров аппаратуры. В них каждый бит имеет свое значение. Не менее важной является возможность экономии памяти – ведь минимальный тип атрибута структуры это байт (char), который занимает 8 битов. До сих пор, несмотря на мегабайты и даже гигабайты оперативной памяти, используемые в современных компьютерах, существует немало задач, где каждый бит на счету.

Если после описания атрибута структуры поставить двоеточие и затем целое число, то это число задает количество битов, выделенных под данный атрибут структуры. Такие атрибуты называют битовыми полями. Следующая структура хранит в компактной форме дату и время дня с точностью до секунды.

```
struct TimeAndDate
{
unsigned hours    :5; // часы от 0 до 24
unsigned mins     :6; // минуты
unsigned secs     :6; // секунды от 0 до 60
unsigned weekDay  :3; // день недели
```

```
unsigned monthDay :6; // день месяца от 1 до 31
unsigned month    :5; // месяц от 1 до 12
unsigned year     :8; // год от 0 до 100
};
```

Одна структура TimeAndDate требует всего 39 битов, т.е. 5 байтов (один байт — 8 битов). Если бы мы использовали для каждого атрибута этой структуры тип char, нам бы потребовалось 7 байтов.

8.4 Объединения

Особым видом структур данных является объединение. Определение объединения напоминает определение структуры, только вместо ключевого слова struct используется union:

```
union number {
    short sx;
    long lx;
    double dx;
};
```

В отличие от структуры, все атрибуты объединения располагаются по одному адресу. Под объединение выделяется столько памяти, сколько нужно для хранения наибольшего атрибута объединения. Объединения применяются в тех случаях, когда в один момент времени используется только один атрибут объединения и, прежде всего, для экономии памяти. Предположим, нам нужно определить структуру, которая хранит "универсальное" число, т.е. число одного из предопределенных типов, и признак типа. Это можно сделать следующим образом:

```
struct Value {
  enum NumberType { ShortType, LongType,
                    DoubleType };
  NumberType type;
  short sx;      // если type равен ShortType
  long lx;       // если type равен LongType
  double dx;     // если type равен DoubleType
};
```

Атрибут type содержит тип хранимого числа, а соответствующий атрибут структуры – значение числа.

```
Value shortVal;
shortVal.type = Value::ShortType;
shortVal.sx = 15;
```

Хотя память выделяется под все три атрибута sx, ls и dx, реально используется только один из них. Сэкономить память можно, используя объединение:

```
struct Value {
    enum NumberType { ShortType, LongType,
                      DoubleType };
    NumberType type;
    union number {
        short sx;      // если type равен ShortType
        long lx;       // если type равен LongType
        double dx;     // если type равен DoubleType
    } val;
};
```

Теперь память выделена только для максимального из этих трех атрибутов (в данном случае dx). Однако и обращаться с объединением надо осторожно. Поскольку все три атрибута делят одну и ту же область памяти, изменение одного из них означает изменение всех остальных. На рисунке поясняется выделение памяти под объединение. В обоих случаях мы предполагаем, что структура расположена по адресу 1000. Объединение располагает все три своих атрибута по одному и тому же адресу.

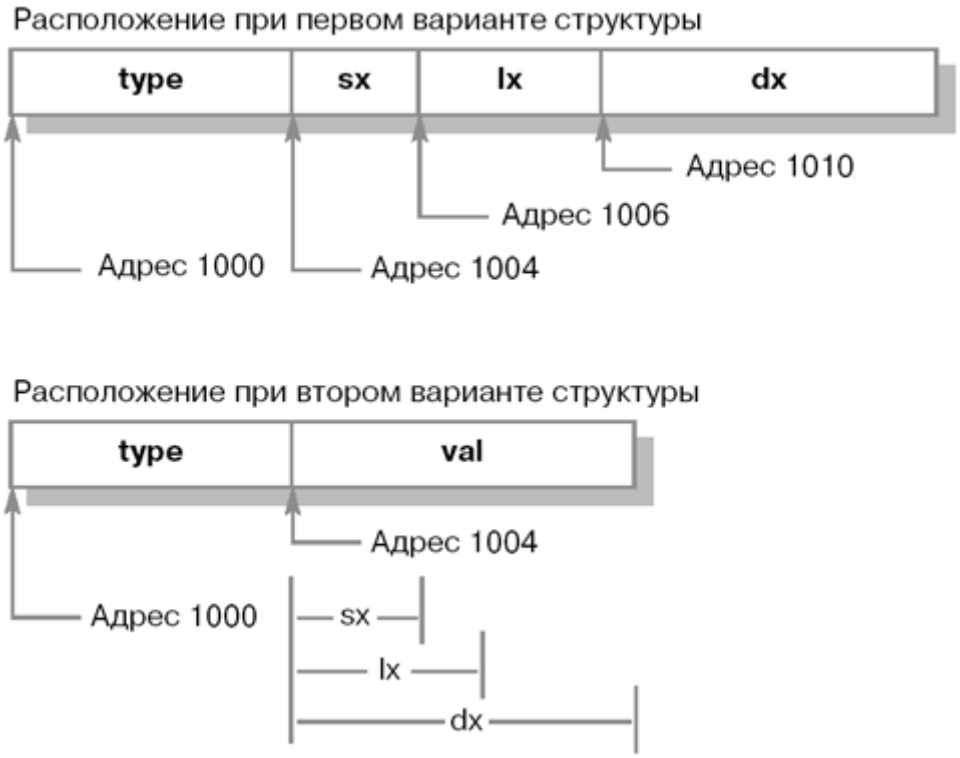

Рис. 8.1. Использование памяти в объединениях.

Замечание. Объединения существовали в языке Си, откуда без изменений и перешли в Си++. Использование наследования классов, описанное в следующей главе, позволяет во многих случаях добиться того же эффекта без использования объединений, причем программа будет более надежной.

8.5 Указатели

Указатель – это производный тип, который представляет собой адрес какого-либо значения. В языке Си++ используется понятие адреса переменных. Работа с адресами досталась Си++ в наследство от языка Си. Предположим, что в программе определена переменная типа int:

```
int x;
```

Можно определить переменную типа "указатель" на целое число:

```
int* xptr;
```

и присвоить переменной xptr адрес переменной x:

```
xptr = &x;
```

Операция &, примененная к переменной, – это операция взятия адреса. Операция *, примененная к адресу, – это операция обращения по адресу. Таким образом, два оператора эквивалентны:

```
int y = x;
// присвоить переменной y значение x
int y = *xptr;
// присвоить переменной y значение,
// находящееся по адресу xptr
```

С помощью операции обращения по адресу можно записывать значения:

```
*xptr = 10;
```

// записать число 10 по адресу xptr

После выполнения этого оператора значение переменной x станет равным 10, поскольку xptr указывает на переменную x.

Указатель – это не просто адрес, а адрес величины определенного типа. Указатель xptr – адрес целой величины. Определить адреса величин других типов можно следующим образом:

```
unsigned long* lPtr;
// указатель на целое число без знака

char* cp;
// указатель на байт

Complex* p;
// указатель на объект класса Complex
```

Если указатель ссылается на объект некоторого класса, то операция обращения к атрибуту класса вместо точки обозначается "->", например p->real. Если вспомнить один из предыдущих примеров:

```
void
Complex::Add(Complex x)
{
  this->real = this->real + x.real;
  this->imaginary = this->imaginary +
                    x.imaginary;
}
```

то this – это указатель на текущий объект, т.е. объект, который выполняет метод Add. Запись this-> означает обращение к атрибуту текущего объекта.

Можно определить указатель на любой тип, в том числе на функцию или метод класса. Если имеется несколько функций одного и того же типа:

```
int foo(long x);
int bar(long x);
```

можно определить переменную типа указатель на функцию и вызывать эти функции не напрямую, а косвенно, через указатель:

```
int (*functptr)(long x);
functptr = &foo;
(*functptr)(2);
functptr = &bar;
(*functptr)(4);
```

Для чего нужны указатели? Указатели появились, прежде всего, для нужд системного программирования. Поскольку язык Си предназначался для "низкоуровневого" программирования, на нем нужно было обращаться, например, к регистрам устройств. У этих регистров вполне определенные адреса, т.е. необходимо было прочитать или записать значение по определенному адресу. Благодаря механизму указателей, такие операции не требуют никаких дополнительных средств языка.

```
int* hardwareRegiste =0x80000;
*hardwareRegiste =12;
```

Однако использование указателей нуждами системного программирования не ограничивается. Указатели позволяют существенно упростить и ускорить ряд операций. Предположим, в программе имеется область памяти для хранения промежуточных результатов вычислений. Эту область памяти используют разные модули программы. Вместо того, чтобы каждый раз при обращении к модулю копировать эту область памяти, мы можем передавать указатель в качестве аргумента вызова функции, тем самым упрощая и ускоряя вычисления.

```
struct TempResults {
```

```
    double x1;
    double x2;
} tempArea;
  // Функция calc возвращает истину, если
  // вычисления были успешны, и ложь – при
  // наличии ошибки. Вычисленные результаты
  // записываются на место аргументов по
  // адресу, переданному в указателе trPtr
bool
calc(TempResults* trPtr)
{
    // вычисления
    if (noerrors) {
        trPtr->x1 = res1;
        trPtr->x2 = res2;
        return true;
    } else {
        return false;
    }
}
void
fun1(void)
{
    . . .
    TempResults tr;
    tr.x1 = 3.4;
    tr.x2 = 5.4;
    if (calc(&tr) == false) {
        // обработка ошибки
    }
    . . .
}
```

В приведенном примере проиллюстрированы сразу две возможности использования указателей: передача адреса общей памяти и возможность функции иметь более одного значения в качестве результата. Структура TempResults используется для хранения данных. Вместо того чтобы передавать эти данные по отдельности, в функцию calc передается указатель на структуру. Таким образом достигаются две цели: большая наглядность и большая эффективность (не надо копировать элементы структуры по одному). Функция calc возвращает булево значение – признак

успешного завершения вычислений. Сами же результаты вычислений записываются в структуру, указатель на которую передан в качестве аргумента.

Упомянутые примеры использования указателей никак не связаны с объектно-ориентированным программированием. Казалось бы, объектно-ориентированное программирование должно уменьшить зависимость от низкоуровневых конструкций типа указателей. На самом деле программирование с классами нисколько не уменьшило потребность в указателях, и даже наоборот, нашло им дополнительное применение, о чем мы будем рассказывать по ходу изложения.

8.6 Адресная арифметика

С указателями можно выполнять не только операции присваивания и обращения по адресу, но и ряд арифметических операций. Прежде всего, указатели одного и того же типа можно сравнивать с помощью стандартных операций сравнения. При этом сравниваются значения указателей, а не значения величин, на которые данные указатели ссылаются. Так, в приведенном ниже примере результат первой операции сравнения будет ложным:

```
int x = 10;
int y = 10;
int* xptr = &x;
int* yptr = &y;
// сравниваем указатели
if (xptr == yptr) {
   cout << "Указатели равны" << endl;
} else {
   cout << "Указатели неравны" << endl;
}
// сравниваем значения, на которые указывают
// указатели
if (*xptr == *yptr) {
    cout << "Значения равны" << endl;
} else {
    cout << "Значения неравны" << endl;
}
```

Однако результат второй операции сравнения будет истинным, поскольку переменные x и y имеют одно и то же значение.

Кроме того, над указателями можно выполнять ограниченный набор арифметических операций. К указателю можно прибавить целое число или вычесть из него целое число. Результатом прибавления к указателю единицы является адрес следующей величины типа, на который ссылается указатель, в памяти. Поясним это на рисунке. Пусть xPtr – указатель на целое число типа long, а cp – указатель на тип char. Начиная с адреса 1000, в памяти расположены два целых числа. Адрес второго — 1004 (в большинстве реализаций Си++ под тип long выделяется четыре байта). Начиная с адреса 2000, в памяти расположены объекты типа char.

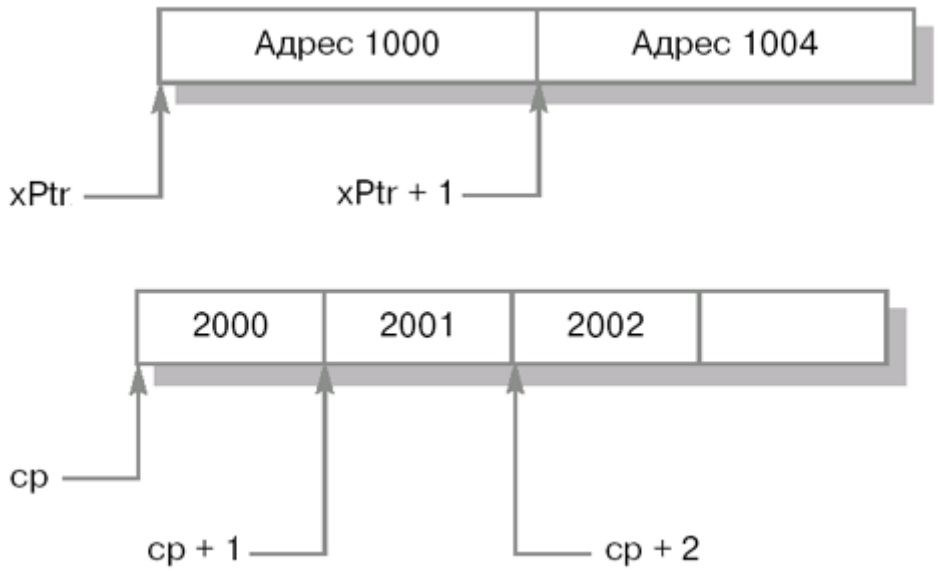

Рис. 8.2. Адресная арифметика.

Размер памяти, выделяемой для числа типа long и для char, различен. Поэтому адрес при увеличении xPtr и cp тоже изменяется по-разному. Однако и в том, и в другом случае увеличение указателя на единицу означает переход к следующей в памяти величине того же типа. Прибавление или вычитание любого целого числа работает по тому же принципу, что и увеличение на единицу. Указатель сдвигается вперед (при прибавлении положительного числа) или назад (при вычитании положительного числа) на

соответствующее количество объектов того типа, на который показывает указатель. Вообще говоря, неважно, объекты какого типа на самом деле находятся в памяти — адрес просто увеличивается или уменьшается на необходимую величину. На самом деле значение указателя ptr всегда изменяется на число, равное sizeof(*ptr).

Указатели одного и того же типа можно друг из друга вычитать. Разность указателей показывает, сколько объектов соответствующего типа может поместиться между указанными адресами.

8.7 Связь между массивами и указателями

Между указателями и массивами существует определенная связь. Предположим, имеется массив из 100 целых чисел. Запишем двумя способами программу суммирования элементов этого массива:

```
long array[100];
long sum = 0;
for (int i = 0; i < 100; i++)
    sum += array[i];
```

То же самое можно сделать с помощью указателей:

```
long array[100];
long sum = 0;
for (long* ptr = &array[0];
    ptr < &array[99] + 1; ptr++)
    sum += *ptr;
```

Элементы массива расположены в памяти последовательно, и увеличение указателя на единицу означает смещение к следующему элементу массива. Упоминание имени массива без индексов преобразуется в адрес его первого элемента:

```
for (long* ptr = array;
    ptr < <array[99] + 1; ptr++)
    sum += *ptr;
```

Хотя смешивать указатели и массивы можно, мы бы не стали рекомендовать такой стиль, особенно начинающим программистам.

При использовании многомерных массивов указатели позволяют

обращаться к срезам или подмассивам. Если мы объявим трехмерный массив exmpl:

long exmpl[5][6][7]

то выражение вида exmpl[1][1][2] – это целое число, exmpl[1][1] – вектор целых чисел (адрес первого элемента вектора, т.е. имеет тип *long), exmpl[1] – двухмерная матрица или указатель на вектор (тип (*long)[7]). Таким образом, задавая не все индексы массива, мы получаем указатели на массивы меньшей размерности.

8.8 Бестиповый указатель

Особым случаем указателей является бестиповый указатель. Ключевое слово void используется для того, чтобы показать, что указатель означает просто адрес памяти, независимо от типа величины, находящейся по этому адресу:

void* ptr;

Для указателя на тип void не определена операция ->, не определена операция обращения по адресу *, не определена адресная арифметика. Использование бестиповых указателей ограничено работой с памятью при использовании ряда системных функций, передачей адресов в функции, написанные на языках программирования более низкого уровня, например на ассемблере.

В программе на языке Си++ бестиповый указатель может применяться там, где адрес интерпретируется по-разному, в зависимости от каких-либо динамически вычисляемых условий. Например, приведенная ниже функция будет печатать целое число, содержащееся в одном, двух или четырех байтах, расположенных по передаваемому адресу:

```
void
printbytes(void* ptr, int nbytes)
{
 if (nbytes == 1) {
   char* cptr = (char*)ptr;
   cout << *cptr;
```

```
  } else if (nbytes == 2) {
    short* sptr = (short*)ptr;
    cout << *sptr;
  } else if (nbytes == 4) {
    long* lptr = (long*)ptr;
    cout << *lptr;
  } else {
    cout << "Неверное значение аргумента";
  }
}
```

В примере используется операция явного преобразования типа. Имя типа, заключенное в круглые скобки, стоящее перед выражением, преобразует значение этого выражения к указанному типу. Разумеется, эта операция может применяться к любым указателям.

8.9 Нулевой указатель

В программах на языке Си++ значение указателя, равное нулю, используется в качестве "неопределенного" значения. Например, если какая-то функция вычисляет значение указателя, то чаще всего нулевое значение возвращается в случае ошибки.

```
long* foo(void);
. . .
long* resPtr;
if ((resPtr = foo()) != 0) {
        // использовать результат
} else {
        // ошибка
}
```

В языке Си++ определена символическая константа NULL для обозначения нулевого значения указателя.

Такое использование нулевого указателя было основано на том, что по адресу 0 данные программы располагаться не могут, он зарезервирован операционной системой для своих нужд. Однако во многом нулевой указатель – просто удобное соглашение, которого все придерживаются.

8.10 Строки и литералы

Для того чтобы работать с текстом, в языке Си++ не существует особого

встроенного типа данных. Текст представляется в виде последовательности знаков (байтов), заканчивающихся нулевым байтом. Иногда такое представление называют Си-строки, поскольку оно появилось в языке Си. Кроме того, в Си++ можно создать классы для более удобной работы с текстами (готовые классы для представления строк имеются в стандартной библиотеке шаблонов).

Строки представляются в виде массива байтов:

```
char string[20];
string[0] = 'H';
string[1] = 'e';
string[2] = 'l';
string[3] = 'l';
string[4] = 'o';
string[5] = 0;
```

В массиве string записана строка "Hello". При этом мы использовали только 6 из 20 элементов массива.

Для записи строковых констант в программе используются литералы. Литерал – это последовательность знаков, заключенная в двойные кавычки:

```
"Это строка"
"0123456789"
"*"
```

Заметим, что символ, заключенный в двойные кавычки, отличается от символа, заключенного в апострофы. Литерал "*" обозначает два байта: первый байт содержит символ звездочки, второй байт содержит ноль. Константа '*' обозначает один байт, содержащий знак звездочки.

С помощью литералов можно инициализировать массивы:

```
char alldigits[] = "0123456789";
```

Размер массива явно не задан, он определяется исходя из размера инициализирующего его литерала, в данном случае 11 (10 символов плюс нулевой байт).

При работе со строками особенно часто используется связь между массивами и указателями. Значение литерала – это массив

неизменяемых байтов нужного размера. Строковый литерал может быть присвоен указателю на char:

```
const char* message = "Сообщение программы";
```

Значение литерала – это адрес его первого байта, указатель на начало строки. В следующем примере функция CopyString копирует первую строку во вторую:

```
void
CopyString(char* src, char* dst)
{
    while (*dst++ = *src++)
        ;
    *dst = 0;
}
int
main()
{
    char first[] = "Первая строка";
    char second[100];
    CopyString(first, second);
    return 1;
}
```

Указатель на байт (тип char*) указывает на начало строки. Предположим, нам нужно подсчитать количество цифр в строке, на которую показывает указатель str:

```
#include <ctype.h>
int count = 0;
while (*str != 0) {
   // признак конца строки – ноль

    if (isdigit(*str++))
   // проверить байт, на который

        count++;
   // указывает str, и сдвинуть
   // указатель на следующий байт
}
```

При выходе из цикла while переменная count содержит количество цифр в строке str, а сам указатель str указывает на конец строки – нулевой байт.

Чтобы проверить, является ли текущий символ цифрой, используется функция isdigit. Это одна из многих стандартных функций языка, предназначенных для работы с символами и строками.

С помощью функций стандартной библиотеки языка реализованы многие часто используемые операции над символьными строками. В большинстве своем в качестве строк они воспринимают указатели. Приведем ряд наиболее употребительных. Прежде чем использовать эти указатели в программе, нужно подключить их описания с помощью операторов #include <string.h> и #include <ctype.h>.

char* strcpy(char* target,
const char* source);

Копировать строку source по адресу target, включая завершающий нулевой байт. Функция предполагает, что памяти, выделенной по адресу target, достаточно для копируемой строки. В качестве результата функция возвращает адрес первой строки.

char* strcat(char* target,
const char* source);

Присоединить вторую строку с конца первой, включая завершающий нулевой байт. На место завершающего нулевого байта первой строки переписывается первый символ второй строки. В результате по адресу target получается строка, образованная слиянием первой со второй. В качестве результата функция возвращает адрес первой строки.

int strcmp(const char* string1,
const char* string2);

Сравнить две строки в лексикографическом порядке (по алфавиту). Если первая строка должна стоять по алфавиту раньше, чем вторая, то результат функции меньше нуля, если позже – больше нуля, и ноль, если две строки равны.

size_t strlen(const char* string);

Определить длину строки в байтах, не считая завершающего нулевого

байта.

В следующем примере, использующем приведенные функции, в массиве result будет образована строка "1 января 1998 года, 12 часов":

```
char result[100];
char* date = "1 января 1998 года";
char* time = "12 часов";
strcpy(result, date);
strcat(result, ", ");
strcat(result, time);
```

Как видно из этого примера, литералы можно непосредственно использовать в выражениях.

Определить массив строк можно с помощью следующего объявления:

```
char* StrArray[5] =
 { "one", "two", "three", "four", "five" };
```

9 Распределение памяти

9.1 Автоматические переменные

Самый простой метод – это объявление переменных внутри функций. Если переменная объявлена внутри функции, каждый раз, когда функция вызывается, под переменную автоматически отводится память. Когда функция завершается, память, занимаемая переменными, освобождается. Такие переменные называют автоматическими.

При создании автоматических переменных они никак не инициализируются, т.е. значение автоматической переменной сразу после ее создания не определено, и нельзя предсказать, каким будет это значение. Соответственно, перед использованием автоматических переменных необходимо либо явно инициализировать их, либо присвоить им какое-либо значение.

```
int
funct()
{
    double f;  // значение f не определено
    f = 1.2;
    // теперь значение f определено
    // явная инициализация автоматической
   // переменной
    bool result = true;
    . . .
}
```

Аналогично автоматическим переменным, объявленным внутри функции, автоматические переменные, объявленные внутри блока (последовательности операторов, заключенных в фигурные скобки) создаются при входе в блок и уничтожаются при выходе из блока.

Замечание. Распространенной ошибкой является использование адреса автоматической переменной после выхода из функции. Конструкция типа:

```
int*
func()
```

```
{
    int x;
    . . .
    return  &x;
}
```

дает непредсказуемый результат.

9.2 Статические переменные

Другой способ выделения памяти – статический

Если переменная определена вне функции, память для нее отводится статически, один раз в начале выполнения программы, и переменная уничтожается только тогда, когда выполнение программы завершается. Можно статически выделить память и под переменную, определенную внутри функции или блока. Для этого нужно использовать ключевое слово static в его определении:

```
double globalMax;
// переменная определена вне функции
void
func(int x)
{
    static bool visited = false;
    if (!visited) {
        . . .  // инициализация
        visited = true;
    }
    . . .
}
```

В данном примере переменная visited создается в начале выполнения программы. Ее начальное значение – false. При первом вызове функции func условие в операторе if будет истинным, выполнится инициализация, и переменной visited будет присвоено значение true. Поскольку статическая переменная создается только один раз, ее значения между вызовами функции сохраняются. При втором и последующих вызовах функции func инициализация производиться не будет.

Если бы переменная visited не была объявлена static, то инициализация

происходила бы при каждом вызове функции.

9.3 Динамическое выделение памяти

Третий способ выделения памяти в языке Си++ – динамический. Память для величины какого-либо типа можно выделить, выполнив операцию new. В качестве операнда выступает название типа, а результатом является адрес выделенной памяти.

```
long* lp;
lp = new long;
Complex* cp;
cp = new Complex;
 // создать новое целое число

 // создать новый объект типа Complex
```

Созданный таким образом объект существует до тех пор, пока память не будет явно освобождена с помощью операции delete. В качестве операнда delete должен быть задан адрес, возвращенный операцией new:

```
delete lp;
delete cp;
```

Динамическое распределение памяти используется, прежде всего, тогда, когда заранее неизвестно, сколько объектов понадобится в программе и понадобятся ли они вообще. С помощью динамического распределения памяти можно гибко управлять временем жизни объектов, например выделить память не в самом начале программы (как для глобальных переменных), но, тем не менее, сохранять нужные данные в этой памяти до конца программы.

Если необходимо динамически создать массив, то нужно использовать немного другую форму new:

```
new int[100];
```

В отличие от определения переменной типа массив, размер массива в операции new может быть произвольным, в том числе вычисляемым в ходе выполнения программы. (Напомним, что при объявлении переменной типа

массив размер массива должен быть константой.)

Освобождение памяти, выделенной под массив, должно быть выполнено с помощью следующей операции delete

```
delete [] address;
```

9.4 Выделение памяти под строки

В следующем фрагменте программы мы динамически выделяем память под строку переменной длины и копируем туда исходную строку

```
// стандартная функция strlen подсчитывает
// количество символов в строке
int length = strlen(src_str);
// выделить память и добавить один байт
// для завершающего нулевого байта
char* buffer = new char[length + 1];
strcpy(buffer, src_str);
// копирование строки
```

Операция new возвращает адрес выделенной памяти. Однако нет никаких гарантий, что new обязательно завершится успешно. Объем оперативной памяти ограничен, и может случиться так, что найти еще один участок свободной памяти будет невозможно. В таком случае new возвращает нулевой указатель (адрес 0). Результат new необходимо проверять:

```
char* newstr;
newstr = new char[length];
if (newstr == NULL) {  // проверить результат
        // обработка ошибок
}
// память выделена успешно
```

9.5 Рекомендации по использованию указателей и динамического распределения памяти

Указатели и динамическое распределение памяти – очень мощные средства языка. С их помощью можно разрабатывать гибкие и весьма эффективные программы. В частности, одна из областей применения Си++ – системное программирование – практически не могла бы существовать без возможности работы с указателями. Однако возможности, которые получает программист при работе с указателями, накладывают на него и

большую ответственность. Наибольшее количество ошибок в программу вносится именно при работе с указателями. Как правило, эти ошибки являются наиболее трудными для обнаружения и исправления.

Приведем несколько примеров.

Использование неверного адреса в операции delete. Результат такой операции непредсказуем. Вполне возможно, что сама операция пройдет успешно, однако внутренняя структура памяти будет испорчена, что приведет либо к ошибке в следующей операции new, либо к порче какой-нибудь информации.

Пропущенное освобождение памяти, т.е. программа многократно выделяет память под данные, но "забывает" ее освобождать. Такие ошибки называют утечками памяти. Во-первых, программа использует ненужную ей память, тем самым понижая производительность. Кроме того, вполне возможно, что в 99 случаях из 100 программа будет успешно выполнена. Однако если потеря памяти окажется слишком большой, программе не хватит памяти под какие-нибудь данные и, соответственно, произойдет сбой.

Запись по неверному адресу. Скорее всего, будут испорчены какие-либо данные. Как проявится такая ошибка – неверным результатом, сбоем программы или иным образом – предсказать трудно

Примеры ошибок можно приводить бесконечно. Общие их черты, обуславливающие сложность обнаружения, это, во-первых, непредсказуемость результата и, во-вторых, проявление не в момент совершения ошибки, а позже, быть может, в том месте программы, которое само по себе не содержит ошибки (неверная операция delete – сбой в последующей операции new, запись по неверному адресу – использование испорченных данных в другой части программы и т.п.).

Отнюдь не призывая отказаться от применения указателей (впрочем, в Си++ это практически невозможно), мы хотим подчеркнуть, что их использование требует внимания и дисциплины. Несколько общих рекомендаций.

1. Используйте указатели и динамическое распределение памяти только там, где это действительно необходимо. Проверьте, можно ли выделить память статически или использовать автоматическую переменную.
2. Старайтесь локализовать распределение памяти. Если какой-либо метод выделяет память (в особенности под временные данные), он же и должен ее освободить.
3. Там, где это возможно, вместо указателей используйте ссылки.
4. Проверяйте программы с помощью специальных средств контроля памяти (Purify компании Rational, Bounce Checker компании Nu-Mega и т.д.)

9.6 Ссылки

Ссылка – это еще одно имя переменной. Если имеется какая-либо переменная, например

Complex x;

то можно определить ссылку на переменную x как

Complex& y = x;

и тогда x и y обозначают одну и ту же величину. Если выполнены операторы

x.real = 1;
x.imaginary = 2;

то y.real равно 1 и y.imaginary равно 2. Фактически, ссылка – это адрес переменной (поэтому при определении ссылки используется символ & -- знак операции взятия адреса), и в этом смысле она сходна с указателем, однако у ссылок есть свои особенности.

Во-первых, определяя переменную типа ссылки, ее необходимо инициализировать, указав, на какую переменную она ссылается. Нельзя определить ссылку

int& xref;

можно только

int& xref = x;

Во-вторых, нельзя переопределить ссылку, т.е. изменить на какой объект она ссылается. Если после определения ссылки xref мы выполним

присваивание

```
xref = y;
```

то выполнится присваивание значения переменной y той переменной, на которую ссылается xref. Ссылка xref по-прежнему будет ссылаться на x. В результате выполнения следующего фрагмента программы:

```
int x = 10;
int y = 20;
int& xref = x;
xref = y;
x += 2;
cout << "x = " << x << endl;
cout << "y = " << y << endl;
cout << "xref = " << xref << endl;
```

будет выведено:

x = 22

y = 20

xref = 22

В-третьих, синтаксически обращение к ссылке аналогично обращению к переменной. Если для обращения к атрибуту объекта, на который ссылается указатель, применяется операция ->, то для подобной же операции со ссылкой применяется точка ".".

```
Complex a;
Complex* aptr = &a;
Complex& aref = a;
aptr->real = 1;
aref.imaginary = 2;
```

Как и указатель, ссылка сама по себе не имеет значения.Ссылка должна на что-то ссылаться, тогда как указатель должен на что-то указывать.

9.7 Распределение памяти при передаче аргументов функции

Рассказывая о функциях, мы отметили, что у функций (как и у методов классов) есть аргументы, фактические значения которых передаются при вызове функции.

Рассмотрим более подробно метод Add класса Complex. Изменим его немного, так, чтобы он вместо изменения состояния объекта возвращал результат операции сложения:

```
Complex
Complex::Add(Complex x)
{
 Complex result;
 result.real = real + x.real;
 result.imaginary = imaginary + x.imaginary;
 return result;
}
```

При вызове этого метода

```
Complex n1;
Complex n2;
. . .
Complex n3 = n1.Add(n2);
```

значение переменной n2 передается в качестве аргумента. Компилятор создает временную переменную типа Complex, копирует в нее значение n2 и передает эту переменную в метод Add. Такая передача аргумента называется передачей по значению. У передачи аргументов по значению имеется два свойства. Во-первых, эта операция не очень эффективна, особенно если объект сложный и требует большого объема памяти или же если создание объекта сопряжено с выполнением сложных действий (о конструкторах объектов будет рассказано в лекции 12). Во-вторых, изменения аргумента функции не сохраняются. Если бы метод Add был бы определен как

```
Complex
Complex::Add(Complex x)
{
 Complex result;
 x.imaginary = 0;
   // изменение аргумента метода
 result.real = real + x.real;
 result.imaginary = imaginary + x.imaginary;
 return result;
```

```
}
```

то при вызове n3 = n1.Add(n2) результат был бы, конечно, другой, но значение переменной n2 не изменилось бы. Хотя в данном примере изменяется значение аргумента метода Add, этот аргумент – лишь копия объекта n2, а не сам объект. По завершении выполнения метода Add его аргументы просто уничтожаются, и первоначальные значения фактических параметров сохраняются.

При возврате результата функции выполняются те же действия, т.е. создается временная переменная, в которую копируется результат, и уже затем значение временной переменной копируется в переменную n3. Временные переменные потому и называют временными, что компилятор сам создает их на время выполнения метода и сам их уничтожает.

Другим способом передачи аргументов является передача по ссылке. Если изменить описание метода Add на

```
Complex
Complex::Add(Complex& x)
{
 Complex result;
 result.real = real + x.real;
 result.imaginary = imaginary + x.imaginary;
 return result;
}
```

то при вызове n3 = n1.Add(n2) компилятор будет создавать ссылку на переменную n2 и передавать ее методу Add. В большинстве случаев это намного эффективнее, так как для ссылки требуется немного памяти и создать ее проще. Однако мы получим нежелательный в данном случае эффект. Метод

```
Complex
Complex::Add(Complex& x)
{
 Complex result;
 x.imaginary = 0;   // изменение значения
                    // по переданной ссылке
 result.real = real + x.real;
```

```
  result.imaginary = imaginary + x.imaginary;
   return result;
}
```

изменит значение переменной n2. Операция Add не предусматривает изменения собственных операндов. Чтобы избежать ошибок, лучше записать аргумент с описателем const, который определяет соответствующую переменную как неизменяемую.

```
Complex::Add(const Complex& x)
```

В таком случае попытка изменить значение аргумента будет обнаружена на этапе компиляции, и компилятор выдаст ошибку. Передачей аргумента по неконстантной ссылке можно воспользоваться в том случае, когда функция действительно должна изменить свой аргумент. Например, метод Coord класса Figure записывает координаты некой фигуры в свои аргументы:

```
void
Figure::Coord(int& x, int& y)
{
x = coordx;
y = coordy;
}
При вызове
int cx, cy;
Figure fig;
. . .
fig.Coord(cx, cy);
```

переменным cx и cy будет присвоено значение координат фигуры fig.

Вернемся к методу Add и попытаемся оптимизировать передачу вычисленного значения. Простое на первый взгляд решение возвращать ссылку на результат не работает:

```
Complex&
Complex::Add(const Complex& x)
{
  Complex result;
  result.real = real + x.real;
  result.imaginary = imaginary + x.imaginary;
```

```
  return result;
}
```

При выходе из метода автоматическая переменная result уничтожается, и память, выделенная для нее, освобождается. Поэтому результат Add – ссылка на несуществующую память. Результат подобных действий непредсказуем. Иногда программа будет работать как ни в чем не бывало, иногда может произойти сбой, иногда результат будет испорчен. Однако возвращение результата по ссылке возможно, если объект, на который эта ссылка ссылается, не уничтожается после выхода из функции или метода. Если метод Add прибавляет значение аргумента к текущему значению объекта и возвращает новое значение в качестве результата, то его можно записать:

```
Complex&
Complex::Add(const Complex& x)
{
  real += x.real;
  imaginary += x.imaginary;
  return *this;
    // передать ссылку на текущий объект
}
```

Как и в случае с аргументом, передача ссылки на текущий объект позволяет использовать метод Add слева от операции присваивания, например в следующем выражении:

```
x.Add(y) = z;
```

К значению объекта x прибавляется значение y, а затем результату присваивается значение z (фактически это эквивалентно x = z). Чтобы запретить подобные конструкции, достаточно добавить описатель const перед типом возвращаемого значения:

```
const Complex&
Complex::Add(const Complex& x)
. . .
```

Передача аргументов и результата по ссылке аналогична передаче указателя в качестве аргумента:

```
Complex*
Complex::Add(Complex* x)
{
  real += x->real;
  imaginary += x->imaginary;
  return this;
}
```

Если нет особых оснований использовать в качестве аргумента или результата именно указатель, передача по ссылке предпочтительней. Во-первых, проще запись операций, а во-вторых, обращения по ссылке легче контролировать.

9.8 Рекомендации по передаче аргументов

1. Встроенные типы лучше передавать по значению. С точки зрения эффективности разницы практически нет, поскольку встроенные типы занимают минимальную память, и создание временных переменных и копирование их значений выполняется быстро.
2. Если в функции или методе значение аргумента используется, но не изменяется, передавайте аргумент по неизменяемой ссылке.
3. Передачу изменяемой ссылки необходимо применять только тогда, когда функция должна изменить переменную, ссылка на которую передается.
4. Передача по указателю используется, только если функции нужен именно указатель, а не значение объекта.

10 Производные классы, наследование

10.1 Наследование

Важнейшим свойством объектно-ориентированного программирования является наследование. Для того, чтобы показать, что класс B наследует классу A (класс B выведен из класса A), в определении класса B после имени класса ставится двоеточие и затем перечисляются классы, из которых B наследует:

```
class A
{
public:
    A();
    ~A();
    MethodA();
};
class B : public A
{
public:
    B();
    . . .
};
```

Термин "наследование" означает, что класс B обладает всеми свойствами класса A, он их унаследовал. У объекта производного класса есть все атрибуты и методы базового класса. Разумеется, новый класс может добавить собственные атрибуты и методы.

```
B b;
b.MethodA();  // вызов метода базового класса
```

Часто выведенный класс называют подклассом, а базовый класс – суперклассом. Из одного базового класса можно вывести сколько угодно подклассов. В свою очередь, производный класс может служить базовым для других классов. Изображая отношения наследования, их часто рисуют в виде иерархии или дерева.

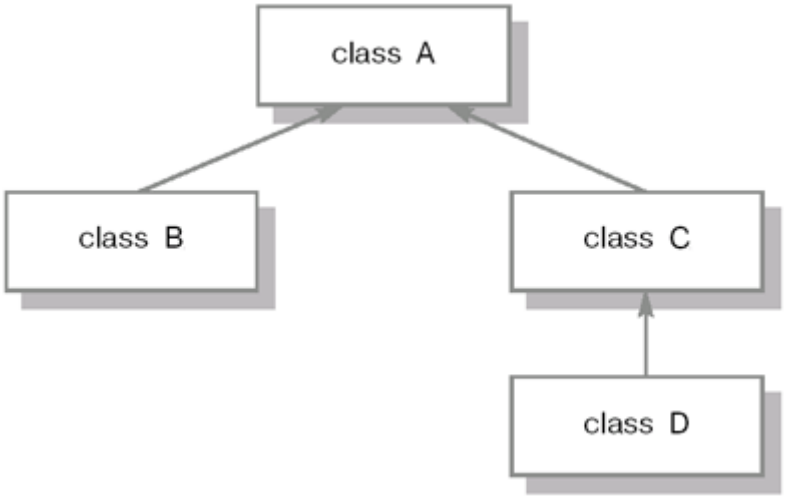

Рис. 10.1. Пример иерархии классов.

Иерархия классов может быть сколь угодно глубокой. Если нужно различить, о каком именно классе идет речь, класс C называют непосредственным или прямым базовым классом класса D, а класс A – косвенным базовым классом класса D.

Предположим, что для библиотечной системы, которую мы разрабатываем, необходимо создать классы, описывающие различные книги, журналы и т.п., которые хранятся в библиотеке. Книга, журнал, газета и микрофильм обладают как общими, так и различными свойствами. У книги имеется автор или авторы, название и год издания. У журнала есть название, номер и содержание – список статей. В то же время книги, журналы и т.д. имеют и общие свойства: все это – "единицы хранения" в библиотеке, у них есть инвентарный номер, они могут быть в читальном зале, у читателей или в фонде хранения. Их можно выдать и, соответственно, сдать в библиотеку. Эти общие свойства удобно объединить в одном базовом классе. Введем класс Item, который описывает единицу хранения в библиотеке:

```
class Item
{
public:
  Item();
  ~Item();
  // истина, если единица хранения на руках

  bool IsTaken() const;
```

```
    // истина, если этот предмет имеется
    // в библиотеке

    bool IsAvailable() const;
    long GetInvNumber() const;
    // инвентарный номер

    void Take();      // операция "взять"
    void Return();    // операция "вернуть"

  private:
      // инвентарный номер — целое число
      long invNumber;
      // хранит состояние объекта —
      // взят на руки
      bool taken;
  };
```

Когда мы разрабатываем часть системы, которая имеет дело с процессом выдачи и возврата книг, вполне достаточно того интерфейса, который представляет базовый класс. Например:

```
  // выдать на руки
  void
  TakeAnItem(Item& i)
  {
      . . .
      if (i.IsAvailable())
          i.Take();
  }
```

Конкретные свойства книги будут представлены классом Book.

```
  class Book : public Item
  {
  public:
      String Author(void) const;
      String Title(void) const;
      String Publisher(void) const;
      long YearOfPublishing(void) const;
      String Reference(void) const;

  private:
      String author;
      String title;
```

```
    String publisher;
    short year;
};    // автор
  // название
  // издательство
  // год выпуска
  // полная ссылка
  // на книгу
```

Для журнала класс Magazin предоставляет другие сведения:

```
class Magazin : public Item
{
public:
    String Volume(void) const;
    short Number(void) const;
    String Title(void) const;
    Date DateOfIssue() const;
private:
    String volume;
    short number;
    String title;
    Date date;
};
  // том
  // номер
  // название
  // дата выпуска
```

Ключевое слово public перед именем базового класса определяет, что внешний интерфейс базового класса становится внешним интерфейсом порожденного класса. Это наиболее употребляемый тип наследования. Описание защищенного и внутреннего наследования будет рассмотрено чуть позже.

У объекта класса Book имеются методы, непосредственно определенные в классе Book и методы, определенные в классе Item.

```
Book b;
long in = b.GetInvNumber();
String t = b.Reference();
```

Производный класс имеет доступ к методам и атрибутам базового класса, объявленным во внешней и защищенной части базового класса,

однако доступ к внутренней части базового класса не разрешен. Предположим, в качестве части полной ссылки на книгу решено использовать инвентарный номер. Метод Reference класса Book будет выглядеть следующим образом:

```
String
Book::Reference(void) const
{
    String result = author + "\n"
            + title + "\n"
            + String(GetInvNumber());
    return result;
```

(Предполагается, что у класса String есть конструктор, который преобразует целое число в строку.) Запись:

```
String result = author + "\n"
            + title + "\n"
            + String(invNumber);
```

не разрешена, поскольку invNumber – внутренний атрибут класса Item. Однако если бы мы поместили invNumber в защищенную часть класса:

```
class Item
{
. . .
protected:
    long invNumber;
};
```

то методы классов Book и Magazin могли бы непосредственно использовать этот атрибут.

Назначение защищенной (protected) части класса в том и состоит, чтобы, закрыв доступ "извне" к определенным атрибутам и методам, разрешить пользоваться ими производным классам.

Если одно и то же имя атрибута или метода встречается как в базовом классе, так и в производном, то производный класс перекрывает базовый.

```
class A
{
```

```
public:
    . . .
    int foo();
    . . .
};
class B : public A
{
public:
    int foo();
    void bar();
};
void
B::bar()
{
    x = foo();
   // вызывается метод foo класса B
}
```

Однако метод базового класса не исчезает. Просто при поиске имени foo сначала просматриваются атрибуты и методы самого класса. Если бы имя не было найдено, начался бы просмотр имен в базовом классе, затем просмотр внешних имен. В данном случае имя foo существует в самом классе, поэтому оно и используется.

С помощью записи A::foo() можно явно указать, что нас интересует имя, определенное в классе A, и тогда запись:

```
x = A::foo();
```

вызовет метод базового класса.

Вообще, запись класс::имя уже многократно нами использовалась. При поиске имени она означает, что имя относится к заданному классу.

10.2 Виртуальные методы

В обоих классах, выведенных из класса Item, имеется метод Title, выдающий в качестве результата заглавие книги или название журнала. Кроме этого метода, полезно было бы иметь метод, выдающий полное название любой единицы хранения. Реализация этого метода различна, поскольку название книги и журнала состоит из разных частей. Однако вид метода – возвращаемое значение и аргументы – и его общий смысл один и

тот же. Название – это общее свойство всех единиц хранения в библиотеке, и логично поместить метод, выдающий название, в базовый класс.

```
class Item
{
public:
    virtual String Name(void) const;
    . . .
};
class Book : public Item
{
public:
    virtual String Name(void) const;
    . . .
};
class Magazin : public Item
{
public:
    virtual String Name(void) const;
    . . .
};
```

Реализация метода Name для базового класса тривиальна: поскольку название известно только производному классу, мы будем возвращать пустую строку.

```
String
Item::Name(void) const
{
    return "";
}
```

Для книги название состоит из фамилии автора, названия книги, издательства и года издания:

```
String
Book::Name(void) const
{
    return author + title + publisher +
        String(year);
}
```

У журнала полное название состоит из названия журнала, года и номера:

```
String
```

```
Magazin::Name(void) const
{
    return title + String(year) +
        String(number);
}
```

Методы Name определены как виртуальные с помощью описателя virtual, стоящего перед определением метода. Виртуальные методы реализуют идею полиморфизма в языке Си++. Если в программе используется указатель на базовый класс Item и с его помощью вызывается метод Name:

```
Item* ptr;
. . .
String name = ptr->Name();
```

то по виду вызова метода невозможно определить, какая из трех приведенных выше реализаций Name будет выполнена. Все зависит от того, на какой конкретный объект указывает указатель ptr.

```
Item* ptr;
. . .
if (type == "Book")
    ptr = new Book;
else if (type == "Magazin")
    ptr = new Magazin;
. . .
String name = ptr->Name();
```

В данном фрагменте программы, если переменная type, обозначающая тип библиотечной единицы, была равна "Book", то будет вызван метод Name класса Book. Если же она была равна "Magazin", то будет вызван метод класса Magazin.

Виртуальные методы позволяют программировать действия, общие для всех производных классов, в терминах базового класса. Динамически, во время выполнения программы, будет вызываться метод нужного класса.

Приведем еще один пример виртуального метода. Предположим, в графическом редакторе при нажатии определенной клавиши нужно

перерисовать текущую форму на экране. Форма может быть квадратом, кругом, эллипсом и т.д. Мы введем базовый класс для всех форм Shape. Конкретные фигуры, с которыми работает редактор, будут представлены классами Square (квадрат), Circle (круг), Ellipse (эллипс), производными от класса Shape. Класс Shape определяет виртуальный метод Draw для отображения формы на экране.

```
class Shape
{
public:
    Shape();
    virtual void Draw(void);
};
//
// квадрат
//
class Square : public Shape
{
public:
    Square();
    virtual void Draw(void);
private:
    double length;   // длина стороны
};
//
// круг
//
class Circle : public Shape
{
public:
    Circle();
    virtual void Draw(void);
private:
    short radius;
};
. . .
```

Конкретные классы реализуют данный метод, и, разумеется, делают это по-разному. Однако в функции перерисовки текущей формы, если у нас имеется указатель на базовый класс, достаточно лишь записать вызов виртуального метода, и динамически будет вызван нужный алгоритм

рисования конкретной формы в зависимости от того, к какому из классов (Square, Circle и т.д.) принадлежит объект, на который указывает указатель shape:

```
Repaint(Shape* shape)
{
   shape->Draw();
}
```

10.3 Виртуальные методы и переопределение методов

Что бы изменилось, если бы метод Name не был описан как виртуальный? В таком случае решение о том, какой именно метод будет выполняться, принимается статически, во время компиляции программы. В примере с методом Name, поскольку мы работаем с указателем на базовый класс, был бы вызван метод Name класса Item. При определении метода как virtual решение о том, какой именно метод будет выполняться, принимается во время выполнения.

Свойство виртуальности проявляется только тогда, когда обращение к методу идет через указатель или ссылку на объект. Указатель или ссылка могут указывать как на объект базового класса, так и на объект производного класса. Если же в программе имеется сам объект, то уже во время компиляции известно, какого он типа и, соответственно, виртуальность не используется.

```
func(Item item)
{
   item.Name();
}
func1(Item& item)
{
   item.Name();


}

 // вызывается метод Item::Name()
```

```
// вызывается метод в соответствии
// с типом того объекта, на который
// ссылается item
```

10.4 Преобразование базового и производного классов

Объект базового класса является частью объекта производного класса. Если в программе используется указатель на производный класс, то его всегда можно без потери информации преобразовать в указатель на базовый класс. Поэтому во многих случаях компилятор может выполнить такое преобразование автоматически.

```
Circle* pC;
. . .
Shape* pShape = pC;
```

Обратное не всегда верно. Преобразование из базового класса в производный не всегда можно выполнить. Поэтому говорят, что преобразование

```
Item* iPtr;
. . .
Book* bPtr = (Book*)iPtr;
```

небезопасно. Такое преобразование можно выполнять только тогда, когда точно известно, что iPtr указывает на объект класса Book.

10.5 Внутреннее и защищенное наследование

До сих пор мы использовали только внешнее наследование. Однако в языке Си++ имеется также внутреннее и защищенное наследование. Если перед именем базового класса ставится ключевое слово private, то наследование называется внутренним.

```
class B : private A
{
. . .
};
```

В случае внутреннего наследования внешняя и защищенная части базового класса становятся внутренней частью производного класса. Внутренняя часть базового класса остается для производного класса

недоступной.

Если перед именем базового класса поставить ключевое слово protected, то будет использоваться защищенное наследование. При нем внешняя и защищенная части базового класса становятся защищенной частью производного класса. Внутренняя часть базового класса остается недоступной для производного класса.

Фактически, при защищенном и внутреннем наследовании производный класс исключает из своего интерфейса интерфейс базового класса, но сам может им пользоваться. Разницу между защищенным и внутренним наследованием почувствует только класс, выведенный из производного.

Если в классе A был определен какой-то метод:

```
class A
{
public:
    int foo();
};
```

то запись

```
B b;
b.foo();
```

недопустима, так же, как и

```
class C
{
    int m() {
        foo();
    }
};
```

если класс B внутренне наследует A. Если же класс B использовал защищенное наследование, то первая запись b.foo() также была бы неправильной, но зато вторая была бы верна.

10.6 Абстрактные классы

Вернемся к примеру наследования, который мы рассматривали раньше.

Мы ввели базовый класс Item, который представляет общие свойства всех единиц хранения в библиотеке. Но существуют ли объекты класса Item? То есть существует ли в действительности "единица хранения" сама по себе? Конечно, каждая книга (класс Book), журнал (класс Magazin) и т.д. принадлежат и к классу Item, поскольку они выведены из него, однако объект самого базового класса вряд ли имеет смысл. Базовый класс – это некое абстрактное понятие, описывающее общие свойства других, конкретных объектов.

Тот факт, что в данном случае объекты базового класса не могут существовать сами по себе, обусловлен еще одним обстоятельством. Некоторые методы базового класса не могут быть реализованы в нем, а должны быть реализованы в порожденных классах. Возьмем, например, тот же метод Name. Его реализация в базовом классе довольно условна, она не имеет особого смысла. Было бы логичнее вообще не реализовывать этот метод в базовом классе, а возложить ответственность за его реализацию на производные классы.

С другой стороны, нам важен факт наличия метода Name во всех производных классах и то, что этот метод виртуален. Именно поэтому мы можем работать с указателями (или ссылками) на объекты базового класса, не зная точно, на какой именно из производных классов этот указатель указывает. Виртуальный механизм во время выполнения программы сам разберется и вызовет нужную реализацию метода Name.

Такая ситуация складывается довольно часто в объектно-ориентированном программировании. (Вспомните пример с различными формами в графическом редакторе: рисование некой обобщенной формы невозможно.) В подобных случаях используется механизм абстрактных классов. Запишем базовый класс Item немного по-другому:

```
class Item
{
public:
```

```
    . . .
    virtual String Name() const = 0;
};
```

Теперь мы определили метод Name как чисто виртуальный. Класс, у которого есть хотя бы один чисто виртуальный метод, называется абстрактным.

Если метод объявлен чисто виртуальным, значит, он должен быть определен во всех классах, производных от Item. Наличие чисто виртуального метода запрещает создание объекта типа Item. В программе можно использовать указатели или ссылки на тип Item. Записи

```
Item it;
Item* itptr = new Item;
```

не разрешены, и компилятор сообщит об ошибке. Однако можно записать:

```
Book b;
Item* itptr = &b;
Item& itref = b;
```

Отметим, что, определив чисто виртуальный метод в классе Book, в следующем уровне наследования его уже не обязательно переопределять (в классах, производных из Book).

Если по каким-либо причинам в производном классе чисто виртуальный метод не определен, то этот класс тоже будет абстрактным, и любые попытки создать объект данного класса будут вызывать ошибку. Таким образом, забыть определить чисто виртуальный метод просто невозможно. Абстрактный базовый класс навязывает определенный интерфейс всем производным из него классам. Собственно, в этом и состоит главное назначение абстрактных классов – в определении интерфейса для всей иерархии классов. Разумеется, это не означает, что в абстрактном классе не может быть определенных методов или атрибутов.

Вообще говоря, класс можно сделать абстрактным, даже если все его методы определены. Иногда это необходимо сделать для того, чтобы быть

уверенным в том, что объект данного класса никогда не будет создан. Можно задать один из методов как чисто виртуальный, но, тем не менее, определить его реализацию. Обычно для этих целей выбирается деструктор:

```
class A
{
public:
    virtual ~A() = 0;
};
A::~A()
{
. . .
}
```

Класс A – абстрактный, и объект типа A создать невозможно. Однако деструктор его определен и будет вызван при уничтожении объектов производных классов (о порядке выполнения конструкторов и деструкторов см. ниже).

10.7 Множественное наследование

В языке Си++ имеется возможность в качестве базовых задать несколько классов. В таком случае производный класс наследует методы и атрибуты всех его родителей. Пример иерархии классов в случае множественного наследования приведен на следующем рисунке.

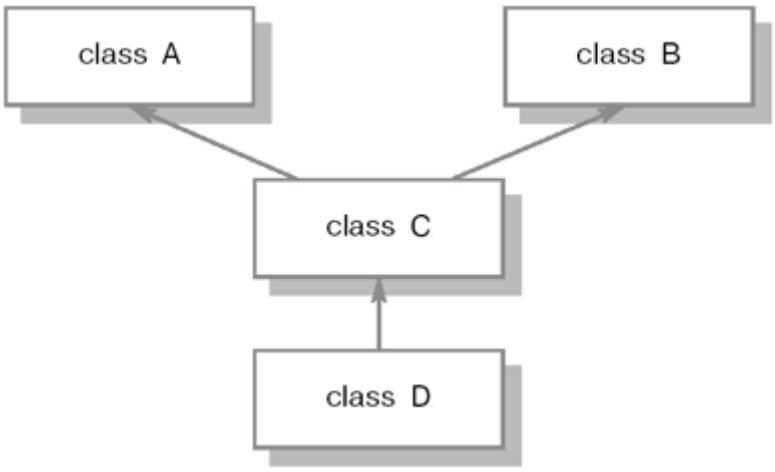

Рис. 10.2. Иерархия классов при множественном наследовании.

В данном случае класс C наследует двум классам, A и B.

Множественное наследование – мощное средство языка. Приведем

некоторые примеры использования множественного наследования.

Предположим, имеющуюся библиотечную систему решено установить в университете и интегрировать с другой системой учета преподавателей и студентов. В библиотечной системе имеются классы, описывающие читателей и работников библиотеки. В системе учета кадров существуют классы, хранящие информацию о преподавателях и студентах. Используя множественное наследование, можно создать классы студентов-читателей, преподавателей-читателей и студентов, подрабатывающих библиотекарями.

В графическом редакторе для некоторых фигур может быть предусмотрен пояснительный текст. При этом все алгоритмы форматирования и печати пояснений работают с классом Annotation. Тогда те фигуры, которые могут содержать пояснение, будут представлены классами, производными от двух базовых классов:

```
class Annotation
{
public:
    String GetText(void);
private:
    String annotation;
};
class Shape
{
public:
    virtual void Draw(void);
};
class AnnotatedSquare : public Shape,
                        public Annotation
{
public:
    virtual void Draw();
};
```

У объекта класса AnnotatedSquare имеется метод GetText, унаследованный от класса Annotation, он определяет виртуальный метод Draw, унаследованный от класса Shape.

При применении множественного наследования возникает ряд проблем. Первая из них – возможный конфликт имен методов или атрибутов нескольких базовых классов.

```
class A
{
public:
    void fun();
    int a;
};
class B
{
public:
    int fun();
    int a;
};
class C : public A, public B
{
};
```

При записи

```
C* cp = new C;
cp->fun();
```

невозможно определить, к какому из двух методов fun происходит обращение. Ситуация называется неоднозначной, и компилятор выдаст ошибку. Заметим, что ошибка выдается не при определении класса C, в котором заложена возможность возникновения неоднозначной ситуации, а лишь при попытке вызова метода fun.

Неоднозначность можно разрешить, явно указав, к которому из базовых классов происходит обращение:

```
cp->A::fun();
```

Вторая проблема заключается в возможности многократного включения базового класса. В упомянутом выше примере интеграции библиотечной системы и системы кадров вполне вероятна ситуация, при которой классы для работников библиотеки и для студентов были выведены из одного и того же базового класса Person:

```
class Person
{
public:
    String name();
};
class Student : public Person
{
. . .
};
class Librarian : public Person
{
. . .
};
```

Если теперь создать класс для представления студентов, подрабатывающих в библиотеке

```
class StudentLibrarian : public Student,
                         public Librarian
{
};
```

то объект данного класса будет содержать объект базового класса Person дважды (см. рисунок 10.3).

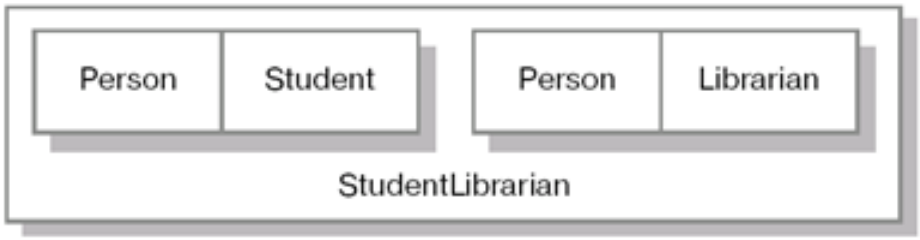

Рис. 10.3. Структура объекта StudentLibrarian.

Кроме того, что подобная ситуация отражает нерациональное использование памяти, никаких неудобств в данном случае она не вызывает. Возможную неоднозначность можно разрешить, явного указав класс:

```
StudentLibrarian* sp;
// ошибка – неоднозначное обращение,
// непонятно, к какому именно экземпляру
// типа Person обращаться
sp->Person::name();
// правильное обращение
sp->Student::Person::name();
```

Тем не менее, иногда необходимо, чтобы объект базового класса содержался в производном один раз. Для этих целей применяется виртуальное наследование, речь о котором впереди.

10.8 Виртуальное наследование

Базовый класс можно объявить виртуальным базовым классом, используя запись:

```
class Student : virtual Person
{
};
class Librarian : virtual Person
{
};
```

Гарантировано, что объект виртуального базового класса будет содержаться в объекте выведенного класса (см. рисунок 10.4) один раз. Платой за виртуальность базового класса являются дополнительные накладные расходы при обращениях к его атрибутам и методам наследование.

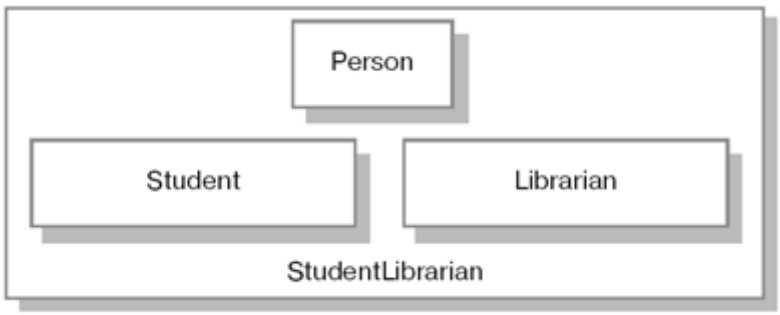

Рис. 10.4. Структура объекта StudentLibrarian при виртуальном множественном наследовании.

11 Контроль доступа к объекту

11.1 Интерфейс и состояние объекта

Основной характеристикой класса с точки зрения его использования является интерфейс, т.е. перечень методов, с помощью которых можно обратиться к объекту данного класса. Кроме интерфейса, объект обладает текущим значением или состоянием, которое он хранит в атрибутах класса. В Си++ имеются богатые возможности, позволяющие следить за тем, к каким частям класса можно обращаться извне, т.е. при использовании объектов, и какие части являются "внутренними", необходимыми лишь для реализации интерфейса.

Определение класса можно поделить на три части – внешнюю, внутреннюю и защищенную. Внешняя часть предваряется ключевым словом public , после которого ставится двоеточие. Внешняя часть – это определение интерфейса. Методы и атрибуты, определенные во внешней части класса, доступны как объектам данного класса, так и любым функциям и объектам других классов. Определением внешней части мы контролируем способ обращения к объекту. Предположим, мы хотим определить класс для работы со строками текста. Прежде всего, нам надо соединять строки, заменять заглавные буквы на строчные и знать длину строк. Соответственно, эти операции мы поместим во внешнюю часть класса:

```
class String
{
public:
  // добавить строку в конец текущей строки
  void Concat(const String& str);
  // заменить заглавные буквы на строчные
  void ToLower(void);
  int GetLength(void) const;
  // сообщить длину строки
    . . .
};
```

Внутренняя и защищенная части класса доступны только при реализации

методов этого класса. Внутренняя часть предваряется ключевым словом private, защищенная – ключевым словом protected.

```
class String
{
public:
  // добавить строку в конец текущей строки
  void Concat(const String& str);
  // заменить заглавные буквы на строчные
  void ToLower(void);
  int GetLength(void) const;
  // сообщить длину строки
private:
    char* str;
    int length;
};
```

В большинстве случаев атрибуты во внешнюю часть класса не помещаются, поскольку они представляют состояние объекта, и возможности их использования и изменения должны быть ограничены. Представьте себе, что произойдет, если в классе String будет изменен указатель на строку без изменения длины строки, которая хранится в атрибуте length.

Объявляя атрибуты str и length как private, мы говорим, что непосредственно к ним обращаться можно только при реализации методов класса, как бы изнутри класса (private по-английски – частный, личный). Например:

```
int
String::GetLength(void) const
{
   return length;
}
```

Внутри определения методов класса можно обращаться не только к внутренним атрибутам текущего объекта, но и к внутренним атрибутам любых других известных данному методу объектов того же класса. Реализация метода Concat будет выглядеть следующим образом:

```
void
String::Concat(const String& x)
```

```
{
    length += x.length;
    char* tmp = new char[length + 1];
    ::strcpy(tmp, str);
    ::strcat(tmp, x.str);
    delete [] str;
    str = tmp;
}
```

Однако если в программе будет предпринята попытка обратиться к внутреннему атрибуту или методу класса вне определения метода, компилятор выдаст ошибку, например:

```
main()
{
    String s;
    if (s.length > 0)   // ошибка
        . . .
}
```

Разница между защищенными (protected) и внутренними атрибутами была описана в предыдущей лекции, где рассматривалось создание иерархий классов.

При записи классов мы помещаем первой внешнюю часть, затем защищенную часть и последней – внутреннюю часть. Дело в том, что внешняя часть определяет интерфейс, использование объектов данного класса. Соответственно, при чтении программы эта часть нужна прежде всего. Защищенная часть необходима при разработке зависимых от данного класса новых классов. И внутреннюю часть требуется изучать реже всего – при разработке самого класса.

11.2 Объявление friend

Предположим, мы хотим в дополнение к интерфейсу класса String создать функцию, которая формирует новую строку, являющуюся результатом слияния двух строк, но не изменяет сами аргументы. (Особенно часто подобный интерфейс необходимо создавать при определении операций – см. ниже). Для того чтобы эта функция работала быстро, желательно, чтобы она

имела доступ к внутренним атрибутам класса String. Доступ можно разрешить, объявив функцию "другом" класса String с помощью ключевого слова friend:

```
class String
{
  . . .
  friend String concat(const String& s1,
          const String& s2);
};
```

Тогда функция concat может быть реализована следующим образом:

```
String
concat(const String& s1, const String& s2)
{
  String result;
  result.length = s1.length + s2.length;
  result.str = new char[result.length + 1];
  if (result.str == 0) {
     // обработка ошибки
  }
  strcpy(result.str, s1.str);
  strcat(result.str, s2.str);
  return result;
}
```

С помощью механизма friend можно разрешить обращение к внутренним элементам класса как отдельной функции, отдельному методу другого класса или всем методам другого класса:

```
class String
{
  // все методы класса StringParser обладают
  // правом доступа ко всем атрибутам класса
  // String
  friend class StringParser;
  // из класса Lexer только метод CharCounter
  // может обращаться к внутренним атрибутам
  // String
  friend int Lexer::CharCounter(const
                String& s, char c);
};
```

Конечно, злоупотреблять механизмом friend не следует. Каждое решение

по использованию friend должно быть продумано. Если только одному методу какого-либо класса действительно необходим доступ, не следует объявлять весь класс как friend.

11.3 Использование описателя const

Во многих примерах мы уже использовали ключевое слово const для обозначения того, что та или иная величина не изменяется. В данном параграфе приводятся подробные правила употребления описателя const.

Если в начале описания переменной стоит описатель const, то описываемый объект во время выполнения программы не изменяется:

```
const double pi = 3.1415;
const Complex one(1,1);
```

Если const стоит перед определением указателя или ссылки, то это означает, что не изменяется объект, на который данный указатель или ссылка указывает:

```
// указатель на неизменяемую строку
const char* ptr = &string;
char x = *ptr;
ptr++;
*ptr = '0';
  // обращение по указателю — допустимо
  // изменение указателя — допустимо
  // попытка изменения объекта, на
  // который указатель указывает –
  // ошибка
```

Если нужно объявить указатель, значение которого не изменяется, то такое объявление выглядит следующим образом:

```
char* const ptr = &string;
// неизменяемый указатель
char x = *ptr;
ptr++;
*ptr = '0';
  // обращение по указателю – допустимо
  // изменение указателя – ошибка
  // изменение объекта, на который
  // указатель указывает – допустимо
```

11.4 Доступ к объекту по чтению и записи

Кроме контроля доступа к атрибутам класса с помощью разделения класса на внутреннюю, защищенную и внешнюю части, нужно следить за тем, с помощью каких методов можно изменить текущее значение объекта, а с помощью каких – нельзя.

При описании метода класса как const выполнение метода не может изменять значение объекта, который этот метод выполняет.

```
class A
{
public:
    int GetValue (void) const;
    int AddValue (int x) const;
private:
    int value;
}
int
A::GetValue(void) const
{
    return value; }
    // объект не изменяется
int
A::AddValue(int x) const
{
    value += x;
   // попытка изменить атрибут объекта
   // приводит к ошибке компиляции

    return value;
}
```

Таким образом, использование описателя const позволяет программисту контролировать возможность изменения информации в программе, тем самым предупреждая ошибки.

В описании класса String один из методов – GetLength – представлен как неизменяемый (в конце описания метода стоит слово const). Это означает, что вызов данного метода не изменяет текущее значение объекта. Остальные методы изменяют его значение. Контроль использования тех или иных методов ведется на стадии компиляции. Например, если аргументом какой-

либо функции объявлена ссылка на неизменяемый объект, то, соответственно, эта функция может вызывать только методы, объявленные как const:

```
int
Lexer::CharCounter(const String& s, char c)
{     int n = s.GetLength();   // допустимо
    s.Concat("ab");
     // ошибка – Concat изменяет значение s
}
```

Общим правилом является объявление всех методов как неизменяемых, за исключением тех, которые действительно изменяют значение объекта. Иными словами, объявляйте как можно больше методов как const. Такое правило соответствует правилу объявления аргументов как const. Объявление константных аргументов запрещает изменение объектов во время выполнения функции и тем самым предотвращает случайные ошибки.

12 Классы – конструкторы и деструкторы

При определении класса имеется возможность задать для объекта начальное значение. Специальный метод класса, называемый конструктором, выполняется каждый раз, когда создается новый объект этого класса. Конструктор – это метод, имя которого совпадает с именем класса. Конструктор не возвращает никакого значения.

Для класса String имеет смысл в качестве начального значения использовать пустую строку:

```
class String
{
public:
    String();   // объявление конструктора
};
// определение конструктора
String::String()
{
    str = 0;
    length = 0;
}
```

Определив такой конструктор, мы гарантируем, что даже при создании автоматической переменной объект будет соответствующим образом инициализирован (в отличие от переменных встроенных типов).

Конструктор без аргументов называется стандартным конструктором или конструктором по умолчанию. Можно определить несколько конструкторов с различными наборами аргументов. Возможности инициализации объектов в таком случае расширяются. Для нашего класса строк было бы логично инициализировать переменную с помощью указателя на строку.

```
class String
{
public:
    String(); // стандартный конструктор
    String(const char* p);
        // дополнительный конструктор
```

```
};
// определение второго конструктора
String::String(const char* p)
{
    length = strlen(p);
    str = new char[length + 1];
    if (str == 0) {
        // обработка ошибок
    }
    strcpy(str, p);   // копирование строки
}
```

Теперь можно, создавая переменные типа String, инициализировать их тем или иным образом:

```
char* cp;
// выполняется стандартный конструктор
String s1;
// выполняется второй конструктор
String s2("Начальное значение");
// выполняется стандартный конструктор
String* sptr = new String;
// выполняется второй конструктор
String* ssptr = new String(cp);
```

12.1 Копирующий конструктор

Остановимся чуть подробнее на одном из видов конструктора с аргументом, в котором в качестве аргумента выступает объект того же самого класса. Такой конструктор часто называют копирующим, поскольку предполагается, что при его выполнении создается объект-копия другого объекта. Для класса String он может выглядеть следующим образом:

```
class String
{
public:
    String(const String& s);
};
String::String(const String& s)
{
    length = s.length;
```

```
    str = new char[length + 1];
    strcpy(str, s.str);
}
```

Очевидно, что новый объект будет копией своего аргумента. При этом новый объект независим от первоначального в том смысле, что изменение значения одного не изменяет значения другого.

```
// первый объект с начальным значением
// "Astring"
String a("Astring");
// новый объект – копия первого,
// т.е. со значением "Astring"
String b(a);
// изменение значения b на "AstringAstring",
// значение объекта a не изменяется
b.Concat(a);
```

Столь логичное поведение объектов класса String на самом деле обусловлено наличием копирующего конструктора. Если бы его не было, компилятор создал бы его по умолчанию, и такой конструктор просто копировал бы все атрибуты класса, т.е. был бы эквивалентен:

```
String::String(const String& s)
{
    length = s.length;
    str = s.str;
}
```

При вызове метода Concat для объекта b произошло бы следующее: объект b перераспределил бы память под строку str, выделив новый участок памяти и удалив предыдущий (см. определение метода выше). Однако указатель str объекта a по-прежнему указывает на первоначальный участок памяти, только что new для данного объекта. Соответственно, для класса можно определить только одну операцию delete. Напомним, что операция delete ответственна только за освобожденный объектом b. Соответственно, значение объекта a испорчено.

Для класса Complex, который мы рассматривали ранее, кроме стандартного конструктора можно задать конструктор, строящий комплексное число из целых чисел:

```
class Complex
{
public:
    Complex();
    Complex(int rl, int im = 0);
    Complex(const Complex& c);
    // прибавить комплексное число
    Complex operator+(const Complex x) const;
private:
    int real;       // вещественная часть
    int imaginary;  // мнимая часть

};
//
// Стандартный конструктор создает число (0,0)
//
Complex::Complex() : real(0), imaginary(0)
{}
//
// Создать комплексное число из действительной
// и мнимой частей. У второго аргумента есть
// значениепо умолчанию — мнимая часть равна
// нулю
Complex::Complex(int rl, int im) :
    real(rl), imaginary(im)
{}
//
// Скопировать значение комплексного числа
//
Complex::Complex(const Complex& c) :
    real(c.real), imaginary(c.imaginary)
{}
```

Теперь при создании комплексных чисел происходит их инициализация:

```
Complex x1;    // начальное значение – ноль
Complex x2(3);
 // мнимая часть по умолчанию равна 0
```

```
 // создается действительное число 3
Complex x3(0, 1);  // мнимая единица
Complex y(x3);  // мнимая единица
```

Конструкторы, особенно копирующие, довольно часто выполняются неявно. Предположим, мы бы описали метод Concat несколько иначе:

```
Concat(String s);
```

вместо

```
Concat(const String& s);
```

т.е. использовали бы передачу аргумента по значению вместо передачи по ссылке. Конечный результат не изменился бы, однако при вызове метода

```
b.Concat(a)
```

компилятор создал бы временную переменную типа String – копию объекта a, и передал бы ее в качестве аргумента. При выходе из метода String эта переменная была бы уничтожена. Представляете, насколько снизилось бы быстродействие метода!

Второй пример вызова конструктора – неявное преобразование типа. Допустима запись вида:

```
b.Concat("LITERAL");
```

хотя сам метод определен только для аргумента – объекта типа String. Поскольку в классе String есть конструктор с аргументом – указателем на байт (а литерал – как раз константа такого типа), компилятор произведет автоматическое преобразование. Будет создана автоматическая переменная типа String с начальным значением "LITERAL", ссылка на нее будет передана в качестве аргумента метода String, а по завершении Concat временная переменная будет уничтожена.

Чтобы избежать подобного неэффективного преобразования, можно определить отдельный метод для работы с указателями:

```
class String
{
public:
    void Concat(const String& s);
    void Concat(const char* s);
};
void
String::Concat(const char* s)
{
    length += strlen(s);
    char* tmp = new char[length + 1];
    if (tmp == 0) {
        // обработка ошибки
    }
    strcpy(tmp, str);
    strcat(tmp. s);
    delete [] str;
    str = tmp;
}
```

12.2 Деструкторы

Аналогично тому, что при создании объекта выполняется конструктор, при уничтожении объекта выполняется специальный метод класса, называемый деструктором. Обычно деструктор освобождает ресурсы, использованные данным объектом.

У класса может быть только один деструктор. Его имя – это имя класса, перед которым добавлен знак "тильда" ‘~’. Для объектов класса String деструктор должен освободить память, используемую для хранения строки:

```
class String
{
    ~String();
};
String::~String()
{
    if (str)
        delete str;
}
```

Если деструктор в определении класса не объявлен, то при уничтожении объекта никаких действий не производится.

Деструктор всегда вызывается перед тем, как освобождается память, выделенная под объект. Если объект типа String был создан с помощью операции new, то при вызове

```
delete sptr;
```

выполняется деструктор ~String(), а затем освобождается память, занимаемая этим объектом. Предположим, в некой функции объявлена автоматическая переменная типа String:

```
int funct(void)
{
   String str;
   . . .
   return 0;
}
```

При выходе из функции funct по оператору return переменная str будет уничтожена: выполнится деструктор и затем освободится память, занимаемая этой переменной.

В особых случаях деструктор можно вызвать явно:

```
sptr->~String();
```

Такие вызовы встречаются довольно редко; соответствующие примеры будут рассматриваться позже, при описании переопределения операций new и delete.

12.3 Инициализация объектов

Рассмотрим более подробно, как создаются объекты. Предположим, формируется объект типа Book.

Во-первых, под объект выделяется необходимое количество памяти: либо динамически, если объект создается с помощью операции new, либо автоматически – при создании автоматической переменной, либо статически

– при создании статической переменной.

Класс Book – производный от класса Item, поэтому вначале вызывается конструктор Item.

У объекта класса Book имеются атрибуты – объекты других классов, в частности, String. После завершения конструктора базового класса будут созданы все атрибуты, т.е. вызваны их конструкторы. По умолчанию используются стандартные конструкторы, как для базового класса, так и для атрибутов.

И только теперь очередь дошла до вызова конструктора класса Book.

В самом конце, после завершения конструктора Book, создаются структуры, необходимые для работы виртуального механизма (отсюда следует, что в конструкторе нельзя использовать виртуальный механизм).

Вызов конструкторов базового класса и конструкторов для атрибутов класса можно задать явно. Особенно это важно, если есть необходимость либо использовать нестандартные конструкторы, либо присвоить начальные значения атрибутам класса. Вызов конструкторов записывается после имени конструктора класса после двоеточия. Друг от друга вызовы отделяются запятой. Такой список называется списком инициализации или просто инициализацией:

```
Item::Item() : taken(false), invNumber(0)
{}
```

В данном случае атрибутам объекта присваиваются начальные значения. Для класса Book конструктор может выглядеть следующим образом:

```
Book::Book() : Item(), title("<None>"),
    author("<None>"), publisher("<None>"),
    year(-1)
{}
```

Вначале выполняется стандартный конструктор класса Item, а затем создаются атрибуты объекта с некими начальными значениями. Теперь

предположим, что у классов Item и Book есть не только стандартные конструкторы, но и конструкторы, которые задают начальные значения атрибутов. Для класса Item конструктор задает инвентарный номер единицы хранения.

```
class Item
{
public:
    Item(long in) { invNumber = in; };
    . . .
};
class Book
{
public:
    Book(long in, const String& a,
        const String& t);
    . . .
};
```

Тогда конструктор класса Book имеет смысл записать так:

```
Book::Book(long in, const String& a,
         const String& t) :
    Item(in), author(a), title(t)
{}
```

Такого же результата можно добиться и при другой записи:

```
Book::Book(long in, const String& a,
         const String& t) :
        Item(in)
{
    author = a;
    title = t;
}
```

Однако предыдущий вариант лучше. Во втором случае вначале для атрибутов author и title объекта типа Book вызываются стандартные конструкторы. Затем программа выполнит операции присваивания новых значений. В первом же случае для каждого атрибута будет выполнен лишь

один копирующий конструктор. Посмотрев на реализацию класса String, вы можете убедиться, насколько эффективнее первый вариант конструктора класса Book.

Встречается еще один случай, когда без инициализации обойтись невозможно. В качестве атрибута класса можно определить ссылку. Однако при создании ссылки ее необходимо инициализировать, поэтому в конструкторе подобного класса нужно применять инициализацию.

```
class A
{
public:
    A(const String& x);
private:
    String& str_ref;
};
A::A(const String& x) : str_ref(x)
{}
```

Создавая объект класса A, мы задаем строку, на которую он будет ссылаться. Ссылка инициализируется во время конструирования объекта. Поскольку ссылку нельзя переопределить, все время жизни объект класса A будет ссылаться на одну и ту же строку. Выбор ссылки в качестве атрибута класса обычно как раз и определяется тем, что ссылка инициализируется при создании объекта и никогда не изменяется. Тем самым дается гарантия использования ссылки на одну и ту же переменную. Значение переменной может изменяться, но сама ссылка – никогда.

Рассмотрим еще один пример использования ссылки в качестве атрибута класса. Предположим, что в нашей библиотечной системе книги, журналы, альбомы и т.д. могут храниться в разных хранилищах. Хранилище описывается объектом класса Repository. У каждого элемента хранения есть атрибут, указывающий на его хранилище. Здесь может быть два варианта. Первый вариант – элемент хранения хранится всегда в одном и том же месте, переместить книгу из одного хранилища в другое нельзя. В данном случае использование ссылки полностью оправдано:

```
class Repository
{
. . .
};
class Item
{
public:
    Item(Repository& rep) :
            myRepository(rep) {};
    . . .
private:
    Repository& myRepository;
};
```

При создании объекта необходимо указать, где он хранится. Изменить хранилище нельзя, пока данный объект не уничтожен. Атрибут myRepository всегда ссылается на один и тот же объект.

Второй вариант заключается в том, что книги можно перемещать из одного хранилища в другое. Тогда в качестве атрибута класса Item лучше использовать указатель на Repository:

```
class Item
{
public:
    Item() : myRepository(0) {};
    Item(Repository* rep) :
            myRepository(rep) {};
    void MoveItem(Repository* newRep);
    . . .
private:
    Repository* myRepository;
};
```

Создавая объект Item, можно указать, где он хранится, а можно и не указывать. Впоследствии можно изменить хранилище, например с помощью метода MoveItem.

При уничтожении объекта вызов деструкторов происходит в обратном порядке. Вначале вызывается деструктор самого класса, затем деструкторы атрибутов этого класса и, наконец, деструктор базового

класса.

В создании и уничтожении объектов имеется одно существенное отличие. Создавая объект, мы всегда точно знаем, какому классу он принадлежит. При уничтожении это не всегда известно.

```
Item* itptr;
if (type == "book")
    itptr = new Book();
else
    itptr = new Magazin();
. . .
delete itptr;
```

Во время компиляции неизвестно, каким будет значение переменной type и, соответственно, объект какого класса удаляется операцией delete. Поэтому компилятор может вставить вызов только деструктора базового класса.

Для того чтобы все необходимые деструкторы были вызваны, нужно воспользоваться виртуальным механизмом – объявить деструктор как в базовом классе, так и в производном, как virtual.

```
class Item
{
    virtual ~Item();
};
class Book
{
public:
    virtual ~Book();
};
```

Возникает вопрос – почему бы всегда не объявлять деструкторы виртуальными? Единственная плата за это – небольшое увеличение памяти для реализации виртуального механизма. Таким образом, не объявлять деструктор виртуальным имеет смысл только в том случае, если во всей иерархии классов нет виртуальных функций, и удаление объекта никогда не происходит через указатель на базовый класс.

12.4 Операции new и delete

Выделение памяти под объекты некоего класса производится либо при создании переменных типа этого класса, либо с помощью операции new. Эти операции, как и другие операции класса, можно переопределить.

Прежде всего, рассмотрим модификацию операции new, которая уже определена в самом языке. (Точнее, она определена в стандартной библиотеке языка Си++.) Эта операция не выделяет память, а лишь создает объект на заранее выделенном участке памяти. Форма операции следующая:

```
new (адрес) имя_класса
        (аргументы_конструктора)
```

Перед именем класса в круглых скобках указывается адрес, по которому должен располагаться создаваемый объект. Фактически, такая операция new не выделяет памяти, а лишь создает объект по указанному адресу, выполняя его конструктор. Соответственно, можно не выполнять операцию delete для этого объекта, а лишь вызвать его деструктор перед тем, как поместить новый объект на то же место памяти.

```
char memory_chunk[4096];
Book* bp = new (memory_chunk) Book;
. . .
bp->~Book();
Magazin* mp = new (memory_chunk) Magazin;
. . .
mp->~Magazin();
```

В этом примере никакой потери памяти не происходит. Память выделена один раз, объявлением массива memory_chunk. Операции new создают объекты в начале этого массива (разумеется, мы предполагаем, что 4096 байтов для объектов достаточно). Когда объект становится ненужным, явно вызывается его деструктор и на том же месте создается новый объект.

Любой класс может использовать два вида операций new и delete – глобальную и определенную для класса. Если класс и ни один из его базовых

классов, как прямых, так и косвенных, не определяет операцию new, то используется глобальная операция new. Глобальная операция new всегда используется для выделения памяти под встроенные типы и под массивы (независимо от того, объекты какого класса составляют массив).

Если класс определит операцию new, то для всех экземпляров этого класса и любых классов, производных от него, глобальная операция будет переопределена, и будет использоваться new данного класса. Если нужно использовать именно глобальную операцию, можно перед new поставить два двоеточия ::new.

Вид стандартной операции new следующий:

```
class A
{
   void* operator new(size_t size);
};
```

Аргумент size задает размер необходимой памяти в байтах. size_t – это тип целого, подходящий для установления размера объектов в данной реализации языка, определенный через typedef. Чаще всего это тип long. Аргумент операции new явно при ее вызове не задается. Компилятор сам его подставляет, исходя из размера создаваемого объекта.

Реализация операции new, которая совпадает со стандартной, выглядит просто:

```
void*
A::operator new(size_t size)
{
   return ::new char[size];
}
```

В классе может быть определено несколько операций new с различными дополнительными аргументами. При вызове new эти аргументы указываются сразу после ключевого слова new в скобках до имени типа. Компилятор добавляет от себя еще один аргумент – размер памяти, и затем вызывает

соответствующую операцию. Описанная выше модификация new, помещающая объект по определенному адресу, имеет вид:

```
void* operator new(void* addr, size_t size);
```

Предположим, мы хотим определить такую операцию, которая будет инициализировать каждый байт выделенной памяти каким-либо числом.

```
class A
{
  void* operator new(char init, size_t size);
};
void*
A::operator new(char init, size_t size)
{
    char* result = ::new char[size];
    if (result) {
        for (size_t i = 0; i < size; i++)
        result[i] = init;
    }
    return result;
}
```

Вызов такой операции имеет вид:

```
A* aptr = new (32) A;
```

Память под объект класса A будет инициализирована числом 32 (что, кстати, является кодом пробела).

Отметим, что если класс определяет хотя бы одну форму операции new, глобальная операция будет переопределена. Например, если бы в классе A была определена только операция new с инициализацией, то вызов

```
A* ptr = new A;
```

привел бы к ошибке компиляции, поскольку подобная форма new в классе не определена. Поэтому, если вы определяете new, определяйте все ее формы, включая стандартную (быть может, просто вызывая глобальную операцию).

В отличие от операции new, для которой можно определить разные

модификации в зависимости от числа и типов аргументов, операция delete существует только в единственном варианте:

```
void operator delete (void* addr);
```

В качестве аргумента ей передается адрес, который в свое время возвратила операция new для данного объекта. Соответственно, для класса можно определить только одну операцию delete. Напомним, что операция delete ответственна только за освобождение занимаемой памяти. Деструктор объекта вызывается отдельно. Операция delete, которая будет вызывать стандартную форму, выглядит следующим образом:

```
void
A::operator delete(void* addr)
{
    ::delete [] (char*)addr;
}
```

13 Дополнительные возможности классов

13.1 Переопределение операций

Язык Си++ позволяет определять в классах особого вида методы – операции. Они называются операциями потому, что их запись имеет тот же вид, что и запись операции сложения, умножения и т.п. со встроенными типами языка Си++.

Определим две операции в классе String – сравнение на меньше и сложение:

```
class String
{
public:
   . . .
 String operator+(const String& s) const;
 bool operator<(const String& s) const;
};
```

Признаком того, что переопределяется операция, служит ключевое слово operator, после которого стоит знак операции. В остальном операция мало чем отличается от обычного метода класса. Теперь в программе можно записать:

```
String s1, s2;
. . .
s1 + s2
```

Объект s1 выполнит метод operator с объектом s2 в качестве аргумента.

Результатом операции сложения является объект типа String. Никакой из аргументов операции не изменяется. Описатель const при описании аргумента говорит о том, что s2 не может измениться при выполнении сложения, а описатель const в конце определения операции говорит то же самое об объекте, выполняющем сложение.

Реализация может выглядеть следующим образом:

```
String
String::operator+(const String& s) const
{
 String result;
```

```
  result.length = length + s.length;
  result.str = new char[result.length + 1];
  strcpy(result.str, str);
  strcat(result.str, s.str);
  return result;
}
```

При сравнении на меньше мы будем сравнивать строки в лексикографической последовательности. Проще говоря, меньше та строка, которая должна стоять раньше по алфавиту:

```
bool
String::operator<(const String& s) const
{
  char* cp1 = str;
  char* cp2 = s.str;
  while (true) {
    if (*cp1 < *cp2)
      return true;
    else if (*cp1 > *cp2)
      return false;
    else {
      cp1++;
      cp2++;
      if (*cp2 == 0)       // конец строки
        return false;
      else if (*cp1 == 0)   // конец строки
        return true;
    }
  }
}
```

13.2 Как определять операции

Если для класса определяют операции, то обычно определяют достаточно полный их набор, так, чтобы объекты этого класса могли участвовать в полноценных выражениях.

Прежде всего, определим операцию присваивания. Операция присваивания в качестве аргумента использует объект того же класса и копирует значение этого объекта. Однако, в отличие от копирующего

конструктора, у объекта уже имеется какое-то свое значение, и его нужно аккуратно уничтожить.

```
class String
{
public:
// объявление операции присваивания
String& operator=(const String& s);
};
// Реализация присваивания
String&
String::operator=(const String& s)
{
    if (this == &s)
        return *this;
    if (str != 0) {
        delete [] str;
    }
    length = s.length;
    str = new char[length + 1];
    if (str == 0) {
        // обработка ошибок
    }
    strcpy(str, s.str);
    return *this;
}
```

Обратим внимание на несколько важных особенностей операции присваивания. Во-первых, в качестве результата операции присваивания объект возвращает ссылку на самого себя. Это дает возможность использовать строки в выражениях типа:

```
s1 = s2 = s3;
```

Во-вторых, в начале операции проверяется, не равен ли аргумент самому объекту. Таким образом, присваивание s1 = s1 выполняется правильно и быстро.

В-третьих, перед тем как скопировать новое значение, операция присваивания освобождает память, занимаемую старым значением.

Аналогично операции присваивания можно определить операцию +=.

Набор операций, позволяющий задействовать класс String в различных

выражениях, представлен ниже:

```
class String
{
public:
  String();
  String(const String& s);
  String(const char*);
  String& operator=(const String& s);
  String& operator+=(const String& s);
  bool operator==(const String& s) const;
  bool operator!=(const String& s) const;
  bool operator<(const String& s) const;
  bool operator>(const String& s) const;
  bool operator<=(const String& s) const;
  bool operator>=(const String& s) const;
  String operator+(const String& s) const;
};
```

13.3 Преобразования типов

Определяя класс, программист задает методы и операции, которые применимы к объектам этого класса. Например, при определении класса комплексных чисел была определена операция сложения двух комплексных чисел. При определении класса строк мы определили операцию конкатенации двух строк. Что же происходит, если в выражении мы попытаемся использовать ту же операцию сложения с типами, для которых она явно не задана? Компилятор пытается преобразовать величины, участвующие в выражении, к типам, для которых операция задана. Это преобразование, называемое преобразованием типов, выполняется в два этапа.

Первый этап – попытка воспользоваться стандартными преобразованиями типов, определенными в языке Си++ для встроенных типов. Если это не помогает, тогда компилятор пытается применить преобразования, определенные пользователем. "Помочь" компилятору правильно преобразовать типы величин можно, явно задав преобразования типов.

13.4 Явные преобразования типов

Если перед выражением указать имя в круглых скобках, то значение выражения будет преобразовано к указанному типу:

```
double x = (double)1;
void* addr;
Complex* cptr = (Complex*) addr;
```

Такие преобразования типов использовались в языке Си. Их основным недостатком является полное отсутствие контроля. Явные преобразования типов традиционно использовались в программах на языке Си и, к сожалению, продолжали использоваться в Си++, что приводит и к ошибкам, и к путанице в программах. В большинстве своем ошибок в Си++ можно избежать. Тем не менее, иногда явные преобразования типов необходимы.

Для того чтобы преобразовывать типы, хотя бы с минимальным контролем, можно записать

```
static_cast < тип > (выражение)
```

Операция static_cast позволяет преобразовывать типы, основываясь лишь на сведениях о типах выражений, известных во время компиляции. Иными словами, static_cast не проверяет типы выражений во время выполнения. С одной стороны, это возлагает на программиста большую ответственность, а с другой — ускоряет выполнение программ. С помощью static_cast можно выполнять как стандартные преобразования, так и нестандартные. Операция static_cast позволяет преобразовывать типы, связанные отношением наследования, указатель к указателю, один числовой тип к другому, перечислимое значение к целому. В частности, с помощью операции static_cast можно преобразовывать не только указатель на производный класс к базовому классу, но и указатель на базовый класс к производному, что в общем случае небезопасно.

Однако попытка преобразовать целое число к указателю приведет к ошибке компиляции. Если все же необходимо преобразовать совершенно не

связанные между собой типы, можно вместо statistatic_cast записать reinterpret_cast:

```
void* addr;
int* intPtr = static_cast < int* > (addr);
Complex* cPtr = reinterpret_cast <
                    Complex* > (2000);
```

Если необходимо ограниченное преобразование типа, которое только преобразует неизменяемый тип к изменяемому (убирает описатель const), можно воспользоваться операцией const_cast:

```
const char* addr1;
char* addr2 = const_cast < char* > addr1;
```

Использование static_cast, const_cast и reinterpret_cast вместо явного преобразования в форме (тип) имеет существенные преимущества. Во-первых, можно всегда применить "минимальное" преобразование, т. е. преобразование, которое меньше всего изменяет тип. Во-вторых, все преобразования можно легко обнаружить в программе. В-третьих, легче распознать намерения программиста, что важно при модификации программы. Сразу можно будет отличить неконтролируемое преобразование от преобразования неизменяемого указателя к изменяемому.

13.5 Стандартные преобразования типов

К стандартным преобразованиям относятся преобразования целых типов и преобразования указателей. Они выполняются компилятором автоматически. Часть правил преобразования мы уже рассмотрели ранее. Преобразования целых величин, при которых не теряется точность, сводятся к следующим:

- Величины типа char, unsigned char, short или unsigned short преобразуются к типу int, если точность типа int достаточна, в противном случае они преобразуются к типу unsigned int.
- Величины типа wchar_t и константы перечисленных типов преобразуются к первому из типов int, unsigned int, long и unsigned long, точность которого достаточна для представления данной

величины.

- Битовые поля преобразуются к типу int, если точность типа int достаточна, или к unsigned int, если точность unsigned int достаточна. В противном случае преобразование не производится.
- Логические значения преобразуются к типу int, false становится 0 и true становится 1.

Эти четыре типа преобразований мы будем называть безопасными преобразованиями.

Язык Си (от которого Си++ унаследовал большинство стандартных преобразований) часто критиковали за излишне сложные правила преобразования типов и за их автоматическое применение без ведома пользователя. Основная рекомендация — избегать неявных преобразований типов, в особенности тех, при которых возможна потеря точности или знака.

Правила стандартных преобразований при выполнении арифметических операций следующие:

- вначале, если в выражении один из операндов имеет тип long double, то другой преобразуется также к long double;
  - в противном случае, если один из операндов имеет тип double, то другой преобразуется также к double;
  - в противном случае, если один из операндов имеет тип float, то другой преобразуется также к float;
  - в противном случае производится безопасное преобразование.
- затем, если в выражении один из операндов имеет тип unsigned long, то другой также преобразуется к unsigned long;
  - в противном случае, если один из операндов имеет тип long, а другой – unsigned int, и тип long может представить все значения unsigned int, то unsigned int преобразуется к long, иначе оба операнда преобразуются к unsigned long;
  - в противном случае, если один из операндов имеет тип long, то другой преобразуется также к long;
  - в противном случае, если один из операндов имеет тип unsigned, то другой преобразуется также к unsigned;
  - в противном случае оба операнда будут типа int.

(1L + 2.3) результат типа double
(8u + 4) результат типа unsigned long

Все приведенные преобразования типов производятся компилятором автоматически, и обычно при компиляции даже не выдается никакого предупреждения, поскольку не теряются значащие цифры или точность результата.

Как мы уже отмечали ранее, при выполнении операции присваивания со стандартными типами может происходить потеря точности. Большинство компиляторов при попытке такого присваивания выдают предупреждение или даже ошибку. Например, при попытке присваивания

```
long x;
char c;
c = x;
```

если значение x равно 20, то и c будет равно 20. Но если x равно 500, значение c будет равно -12 (при условии выполнения на персональном компьютере), поскольку старшие биты, не помещающиеся в char, будут обрезаны. Именно поэтому большинство компиляторов выдаст ошибку и не будет транслировать подобные конструкции.

13.6 Преобразования указателей и ссылок

При работе с указателями и ссылками компилятор автоматически выполняет только два вида преобразований.

Если имеется указатель или ссылка на производный тип, а требуется, соответственно, указатель или ссылка на базовый тип.

Если имеется указатель или ссылка на изменяемый объект, а требуется указатель или ссылка на неизменяемый объект того же типа.

```
size_t strlen(const char* s);
        // прототип функции
class A { };
class B : public A { };
char* cp;
strlen(cp);
        // автоматическое преобразование из
        // char* в const char*

B* bObj = new B;
```

```
        // преобразование из указателя на
A* aObj = bObj;
        // производный класс к указателю на
        // базовый класс
```

Если требуются какие-то другие преобразования, их необходимо указывать явно, но в этом случае вся ответственность за правильность преобразования лежит на программисте.

13.7 Преобразования типов, определенных в программе

В языке Си++ можно определить гораздо больше типов, чем в Си. Казалось бы, и правила преобразования новых типов должны стать намного сложнее. К счастью, этого не произошло. Все дело в том, что при определении классов программист может контролировать, какие преобразования допустимы и как они выполняются при преобразовании в данный тип или из данного типа в другой.

Прежде всего, выполнение тех или иных операций с аргументами разных типов можно регулировать с помощью методов и функций с разными аргументами. Для того чтобы определить операцию сложения комплексного числа с целым, нужно определить две функции в классе Complex:

```
class Complex {
    . . .
 friend Compex operator+(const Complex& x,
                         int y);
 friend Compex operator+(int y,
                         const Complex& x);
};
```

При наличии таких функций никаких преобразований типа не производится в следующем фрагменте программы:

```
int x;
Complex y;
. . .
Complex z = x + y;
```

Тем не менее, в других ситуациях преобразования типа производятся. Прежде всего, компилятор старается обойтись стандартными

преобразованиями типа. Если их не хватает, то выполняются преобразования либо с помощью конструкторов, либо с помощью определенных программистом операций преобразования.

Задав конструктор класса, имеющий в качестве аргумента величину другого типа, программист тем самым определяет правило преобразования:

```
class Complex
{
public:
    // неявное правило преобразования
    // из целого типа в тип Complex
    Complex(int x);
};
```

Операции преобразования имеют вид:

```
operator имя_типа ();
```

Например, преобразование из комплексного числа в целое можно записать так:

```
class Complex
{
public:
    // операция преобразования из типа
    // Complex в целый тип
    operator int();
};
```

При записи:

```
Complex cmpl;
int x = cmpl;
```

будет вызвана функция operator int().

14 Компоновка программ, препроцессор

14.1 Компоновка нескольких файлов в одну программу

Программа – это, прежде всего, текст на языке Си++. С помощью компилятора текст преобразуется в исполняемый файл – форму, позволяющую компьютеру выполнять программу.

Если мы рассмотрим этот процесс чуть более подробно, то выяснится, что обработка исходных файлов происходит в три этапа. Сначала файл обрабатывается препроцессором, который выполняет операторы #include, #define и еще несколько других. После этого программа все еще представлена в виде текстового файла, хотя и измененного по сравнению с первоначальным. Затем, на втором этапе, компилятор создает так называемый объектный файл. Программа уже переведена в машинные инструкции, однако еще не полностью готова к выполнению. В объектном файле имеются ссылки на различные системные функции и на стандартные функции языка Си++. Например, выполнение операции new заключается в вызове определенной системной функции. Даже если в программе явно не упомянута ни одна функция, необходим, по крайней мере, один вызов системной функции – завершение программы и освобождение всех принадлежащих ей ресурсов.

На третьем этапе компиляции к объектному файлу подсоединяются все функции, на которые он ссылается. Функции тоже должны быть скомпилированы, т.е. переведены на машинный язык в форму объектных файлов. Этот процесс называется компоновкой, и как раз его результат и есть исполняемый файл.

Системные функции и стандартные функции языка Си++ заранее откомпилированы и хранятся в виде библиотек. Библиотека – это некий архив объектных модулей, с которым удобно компоновать программу.

Основная цель многоэтапной компиляции программ – возможность компоновать программу из многих файлов. Каждый файл представляет собой законченный фрагмент программы, который может ссылаться на

функции, переменные или классы, определенные в других файлах. Компоновка объединяет фрагменты в одну "самодостаточную" программу, которая содержит все необходимое для выполнения.

14.2 Проблема использования общих функций и имен

В языке Си++ существует строгое правило, в соответствии с которым прежде чем использовать в программе имя или идентификатор, его необходимо определить. Рассмотрим для начала функции. Для того чтобы имя функции стало известно программе, его нужно либо объявить, либо определить.

Объявление функции состоит лишь из ее прототипа, т.е. имени, типа результата и списка аргументов. Объявление функции задает ее формат, но не определяет, как она выполняется. Примеры объявлений функций:

```
double sqrt(double x);// функция sqrt
long fact(long x);       // функция fact

// функция PrintBookAnnotation
void PrintBookAnnotation(const Book& book);
```

Определение функции – это определение того, как функция выполняется. Оно включает в себя тело функции, программу ее выполнения.

```
// функция вычисления факториала
// целого положительного числа
long fact(long x)
{
    if (x == 1)
        return 1;
    else
        return x * fact(x - 1);
}
```

Определение функции играет роль объявления ее имени, т.е. если в начале файла определена функция fact, в последующем тексте функций и классов ею можно пользоваться. Однако если в программе функция fact используется в нескольких файлах, такое построение программы уже не

подходит. В программе должно быть только одно Определение функции.

Удобно было бы поместить Определение функции в отдельный файл, а в других файлах в начале помещать лишь объявление, прототип функции.

```
// начало файла main.cpp
long fact(long);// прототип функции
int main()
{
    . . .
    int x10 = fact(10);   // вызов функции
    . . .
}
// конец файла main.cpp
// начало файла fact.cpp
// определение функции
// вычисления факториала целого
// положительного числа
//
long fact(long x)
{
    if (x == 1)
        return 1;
    else
        return x * fact(x - 1);
}
// конец файла fact. cpp
```

Компоновщик объединит оба файла в одну программу.

Аналогичная ситуация существует и для классов. Любой класс в языке Си++ состоит из двух частей: объявления и определения. В объявлении класса говорится, каков интерфейс класса, какие методы и атрибуты составляют объекты этого класса. Объявление класса состоит из ключевого слова class, за которым следует имя класса, список наследования и затем в фигурных скобках - методы и атрибуты класса. Заканчивается объявление класса точкой с запятой.

```
class Book : public Item
{
public:
    Book();
```

```
    ~Book();
    String Title();
    String Author();
private:
    String title;
    String author;
};
```

Определение класса – это определение всех его методов.

```
// определение метода Title
String
Book::String()
{
    return title;
}
```

Определение класса должно быть только одно, и если класс используется во многих файлах, его удобно поместить в отдельный файл. В остальных файлах для того, чтобы использовать класс Book, например определить переменную класса Book, в начале файла необходимо поместить объявление класса.

Таким образом, в начале каждого файла будут сосредоточены прототипы всех используемых функций и объявления всех применяемых классов.

Программа работать будет, однако писать ее не очень удобно.

В начале каждого файла нам придется повторять довольно большие одинаковые куски текста. Помимо того, что это утомительно, очень легко допустить ошибку. Если по каким-то причинам потребуется изменить объявление класса, придется изменять все файлы, в которых он используется. Хотя возможно, что изменение никак не затрагивает интерфейс класса. Например, добавление нового внутреннего атрибута непосредственно не влияет на использование внешних методов и атрибутов этого класса.

14.3 Использование включаемых файлов

В языке Си++ реализовано удобное решение. Можно поместить объявления классов и функций в отдельный файл и включать этот файл в

начало других файлов с помощью оператора #include.

```
#include "Book.h"
. . .
Book b;
```

Фактически оператор #include подставляет содержимое файла Book.h в текущий файл перед тем, как начать его компиляцию. Эта подстановка осуществляется во время первого прохода компилятора по программе – препроцессора. Файл Book.h называется файлом заголовков.

В такой же файл заголовков можно поместить прототипы функций и включать его в другие файлы, там, где функции используются.

Таким образом, текст программы на языке Си++ помещается в файлы двух типов – файлы заголовков и файлы программ. В большинстве случаев имеет смысл каждый класс помещать в отдельный файл, вернее, два файла – файл заголовков для объявления класса и файл программ для определения класса. Имя файла обычно состоит из имени класса. Для файла заголовков к нему добавляется окончание ".h" (иногда, особенно в системе Unix, ".hh" или ".H"). Имя файла программы – опять-таки имя класса с окончанием ".cpp" (иногда ".cc" или ".C").

Объединять несколько классов в один файл стоит лишь в том случае, если они очень тесно связаны и один без другого не используются.

Включение файлов может быть вложенным, т.е. файл заголовков может сам использовать оператор #include. Файл Book.h выглядит следующим образом:

```
#ifndef __BOOK_H__
#define __BOOK_H__

// включить файл с объявлением используемого
// здесь базового класса
#include "Item .h"
#include "String.h"
// объявление класса String
```

```
// объявление класса Book
class Book : public Item
{
public:
. . .
private:
        String title;
    . . .
}; #endif
```

Обратите внимание на первые две и последнюю строки этого файла. Оператор #ifndef начинает блок так называемой условной компиляции, который заканчивается оператором #endif. Блок условной компиляции – это кусок текста, который будет компилироваться, только если выполнено определенное условие. В данном случае условие заключается в том, что символ __BOOK_H__ не определен. Если этот символ определен, текст между #ifndef и #endif не будет включен в программу. Первым оператором в блоке условной компиляции стоит оператор #define, который определяет символ __BOOK_H__ как пустую строку.

Давайте посмотрим, что произойдет, если в какой-либо .cpp-файл будет дважды включен файл Book.h:

```
#include "Book.h"
. . .
#include "Book.h"
```

Перед началом компиляции текст файла Book.h **будет подставлен вместо оператора** #include**:**

```
#ifndef __BOOK_H__
#define __BOOK_H__
. . .
class Book
{
. . .
};
#endif
. . .
#ifndef __BOOK_H__
#define __BOOK_H__
```

```
. . .
class Book
{
. . .
};
#endif
```

В самом начале символ __BOOK_H__ не определен, и блок условной компиляции обрабатывается. В нем определяется символ __BOOK_H__ . Теперь условие для второго блока условной компиляции уже не выполняется, и он будет пропущен. Таким образом, объявление класса Book будет вставлено в файл только один раз. Разумеется, написание два раза подряд оператора #include с одинаковым аргументом легко поправить. Однако структура заголовков может быть очень сложной. Чтобы избежать необходимости отслеживать все вложенные заголовки и искать, почему какой-либо файл оказался вставленным дважды, можно применить изложенный выше прием и существенно упростить себе жизнь.

Еще одно замечание по составлению заголовков. Включайте в заголовок как можно меньше других заголовков. Например, в заголовок Book.h необходимо включить заголовки Item.h и String.h, поскольку класс Book использует их. Однако если используется лишь имя класса без упоминания его содержимого, можно обойтись и объявлением этого имени:

```
#include "Item.h"
#include "String.h"

class Annotation;
// Annotation – имя некого класса

class Book : public Item
{
public:
    Annotation* CreateAnnotation();
private:
    String title;
};
```

Объявление класса Item требуется знать целиком, для того, чтобы обработать объявление класса Book, т.е. компилятору надо знать все методы и атрибуты Item, чтобы включить их в класс Book. Объявление класса String также необходимо знать целиком, по крайней мере, для того, чтобы правильно вычислить размер экземпляра класса Book. Что же касается класса Annotation, то ни размер его объектов, ни его методы не важны для определения содержимого объекта класса Book. Единственное, что надо знать, это то, что Annotation есть имя некоего класса, который будет определен в другом месте.

Общее правило таково, что если объявление класса использует указатель или ссылку на другой класс и не задействует никаких методов или атрибутов этого класса, достаточно объявления имени класса. Разумеется, полное объявление класса Annotation понадобится в определении метода CreateAnnotation.

Компилятор поставляется с набором файлов заголовков, которые описывают все стандартные функции и классы. При включении стандартных файлов обычно используют немного другой синтаксис:

#include <string.h>

14.4 Препроцессор

В языке Си++ имеется несколько операторов, которые начинаются со знака #: #include, #define, #undef, #ifdef, #else, #if, #pragma. Все они обрабатываются так называемым препроцессором.

Иногда препроцессор называют макропроцессором, поскольку в нем определяются макросы. Директивы препроцессора начинаются со знака #, который должен быть первым символом в строке после пробелов.

14.5 Определение макросов

Форма директивы #define

#define имя определение

определяет макроимя. Везде, где в исходном файле встречается это имя, оно будет заменено его определением. Например, текст:

```
#define NAME "database"
Connect(NAME);
```

после препроцессора будет заменен на

```
Connect("database");
```

По умолчанию имя определяется как пустая строка, т.е. после директивы

```
#define XYZ
```

макроимя XYZ считается определенным со значением – пустой строкой.

Другая форма #define

```
#define имя ( список_имен ) определение
```

определяет макрос – текстовую подстановку с аргументами

```
#define max(X, Y) ((X > Y) ? X : Y)
```

Текст max(5, a) будет заменен на

```
((5 > a) ? 5 : a)
```

В большинстве случаев использование макросов (как с аргументами, так и без) в языке Си++ является признаком непродуманного дизайна. В языке Си макросы были действительно важны, и без них было сложно обойтись. В Си++ при наличии констант и шаблонов макросы не нужны. Макросы осуществляют текстовую подстановку, поэтому они в принципе не могут осуществлять никакого контроля использования типов. В отличие от них в шаблонах контроль типов полностью сохранен. Кроме того, возможности

текстовой подстановки существенно меньше, чем возможности генерации шаблонов.

Директива #undef отменяет определение имени, после нее имя перестает быть определенным.

У препроцессора есть несколько макроимен, которые он определяет сам, их называют предопределенными именами. У разных компиляторов набор этих имен различен, но два определены всегда: __FILE__ и __LINE__. Значением макроимени __FILE__ является имя текущего исходного файла, заключенное в кавычки. Значением __LINE__ – номер текущей строки в файле. Эти макроимена часто используют для печати отладочной информации.

14.6 Условная компиляция

Исходный файл можно компилировать не целиком, а частями, используя директивы условной компиляции:

```
#if LEVEL > 3
текст1
#elif LEVEL > 1
текст2
#else
текст3
#endif
```

Предполагается, что LEVEL – это макроимя, поэтому выражение в директивах #if и #elif можно вычислить во время обработки исходного текста препроцессором.

Итак, если LEVEL больше 3, то компилироваться будет текст1, если LEVEL больше 1, то компилироваться будет текст2, в противном случае компилируется текст3. Блок условной компиляции должен завершаться директивой #endif.

В каком-то смысле директива #if похожа на условный оператор if. Однако, в отличие от него, условие – это константа, которая вычисляется на стадии препроцессора, и куски текста, не удовлетворяющие условию, просто

игнорируются.

Директив #elif может быть несколько (либо вообще ни одной), директива #else также может быть опущена.

Директива #ifdef – модификация условия компиляции. Условие считается выполненным, если указанное после нее макроимя определено. Соответственно, для директивы #ifndef условие выполнено, если имя не определено.

14.7 Дополнительные директивы препроцессора

Директива #pragma используется для выдачи дополнительных указаний компилятору. Например, не выдавать предупреждений при компиляции, или вставить дополнительную информацию для отладчика. Конкретные возможности директивы #pragma у разных компиляторов различные.

Директива #error выдает сообщение и завершает компиляцию. Например, конструкция

```
#ifndef unix
#error "Программу можно компилировать
                только для Unix!"
#endif
```

выдаст сообщение и не даст откомпилировать исходный файл, если макроимя unix не определено.

Директива #line изменяет номер строки и имя файла, которые хранятся в предопределенных макроименах __LINE__ и __FILE__.

Кроме директив, у препроцессора есть одна операция ##, которая соединяет строки, например A ## B.

15 Определение, время жизни и области видимости переменных в больших программах

15.1 Файлы и переменные

Автоматические переменные определены внутри какой-либо функции или метода класса. Назначение автоматических переменных – хранение каких-либо данных во время выполнения функции или метода. По завершении выполнения этой функции автоматические переменные уничтожаются и данные теряются. С этой точки зрения автоматические переменные представляют собой временные переменные.

Иногда временное хранилище данных требуется на более короткое время, чем выполнение всей функции. Во- первых, поскольку в Си++ необязательно, чтобы все используемые переменные были определены в самом начале функции или метода, переменную можно определить непосредственно перед тем, как она будет использоваться. Во-вторых, переменную можно определить внутри блока – группы операторов, заключенных в фигурные скобки. При выходе из блока такая переменная уничтожается еще до окончания выполнения функции. Третьей возможностью временного использования переменной является определение переменной в заголовке цикла for только для итераций этого цикла:

```
funct(int N, Book[]& bookArray)
{
 int x;    // автоматическая переменная x
 for (int i = 0; i < N; i++) {
 // переменная i определена только на время
 // выполнения цикла for
   String s;
 // новая автоматическая переменная создается
 // при каждой итерации цикла заново
   s.Append(bookArray[i].Title());
   s.Append(bookArray[i].Author());
   cout << s;
 }
 cout << s;
}   // ошибка, переменная s не существует
```

Если переменную, определенную внутри функции или блока, описать как статическую, она не будет уничтожаться при выходе из этого блока и будет хранить свое значение между вызовами функции. Однако при выходе из соответствующего блока эта переменная станет недоступна, иными словами, невидима для программы. В следующем примере переменная allAuthors накапливает список авторов книг, переданных в качестве аргументов функции funct за все ее вызовы:

```
funct(int n, Book[]& bookArray)
{
  for (int i = 0; i < n; i++) {
    static String allAuthors;
    allAuthors.Append(bookArray[i].Author());
    cout << allAuthors;
    // авторы всех ранее обработанных книг, в
    // том числе в предыдущих вызовах функции
  }
  cout << allAuthors;
  // ошибка, переменная недоступна
}
```

15.2 Общие данные

Иногда необходимо, чтобы к одной переменной можно было обращаться из разных функций. Предположим, в нашей программе используется генератор случайных чисел. Мы хотим инициализировать его один раз, в начале выполнения программы, а затем обращаться к нему из разных частей программы. Рассмотрим несколько возможных реализаций.

Во-первых, определим класс RandomGenerator с двумя методами: Init, для инициализации генератора, и GetNumber — для получения следующего числа.

```
//
// файл RandomGenerator.h
//
class RandomGenerator
{
public:
  RandomGenerator();
```

```
  ~RandomGenerator();
  void Init(unsigned long start);
  unsigned long GetNumber();
private:
  unsigned long previousNumber;
};
//
// файл RandomGenerator.cpp
//
#include "RandomGenerator.h"
#include <time.h>
void
RandomGenerator::Init(unsigned long x)
{
  previousNumber = x;
}
unsigned long
RandomGenerator::GetNumber(void)
{
  unsigned long ltime;
  // получить текущее время в секундах,
  // прошедших с полуночи 1 января 1970 года
  time(&ltime);
  ltime <<= 16;
  ltime >>= 16;
  // взять младшие 16 битов
  previousNumber = previousNumber * ltime;
  return previousNumber;
}
```

Первый вариант состоит в создании объекта класса RandomGenerator в функции main и передаче ссылки на него во все функции и методы, где он потребуется.

```
// файл main.cpp
#include "RandomGenerator.h"
main()
{
    RandomGenerator rgen;
    rgen.Init(1000);
    fun1(rgen);
    . . .
    Class1 b(rgen);
    . . .
```

```
    fun2(rgen);
}
void
fun1(RandomGenerator& r)
{
    unsigned long x = r.GetNumber();
    . . .
}
// файл class.cpp
#include "RandomGenerator.h"
Class1::Class1(RandomGenerator& r)
{
    . . .
}
void
fun2(RandomGenerator& r)
{
    unsigned long x = r.GetNumber();
    . . .
}
```

Поскольку функция main завершает работу программы, все необходимые условия выполнены: генератор случайных чисел создается в самом начале программы, все объекты и функции обращаются к одному и тому же генератору, и генератор уничтожается по завершении программы. Такой стиль программирования допустимо использовать только в том случае, если передавать ссылку на используемый экземпляр объекта требуется нечасто. В противном случае этот способ крайне неудобен. Передавать ссылку на один и тот же объект утомительно, к тому же это загромождает интерфейс классов.

15.3 Глобальные переменные

Язык Си++ предоставляет возможность определения глобальной переменной. Если переменная определена вне функции, она создается в самом начале выполнения программы (еще до начала выполнения main). Эта переменная доступна во всех функциях того файла, где она определена. Аналогично прототипу функции, имя глобальной переменной можно объявить в других файлах и тем самым предоставить возможность обращаться к ней и в других файлах:

```
// файл main.cpp
#include "RandomGenerator.h"
// определение глобальной переменной
RandomGenerator rgen;
main()
{
    rgen.Init(1000);
}
void
fun1(void)
{
    unsigned long x = rgen.GetNumber();
    . . .
}
// файл class.cpp

#include "RandomGenerator.h"
// объявление глобальной переменной,
// внешней по отношению к данному файлу
extern RandomGenerator rgen;
Class1::Class1()
{
    . . .
}
void
fun2()
{
    unsigned long x = rgen.GetNumber();
    . . .
}
```

Объявление внешней переменной можно поместить в файл-заголовок. Тогда не нужно будет повторять объявление переменной с описателем extern в каждом файле, который ее использует.

Модификацией определения глобальной переменной является добавление описателя static. Для глобальной переменной описатель static означает то, что эта переменная доступна только в одном файле – в том, в котором она определена. (Правда, в данном примере такая модификация недопустима – нам-то как раз нужно, чтобы к глобальной переменной rgen можно было обращаться из разных файлов.)

15.4 Повышение надежности обращения к общим данным

Определять глобальную переменную намного удобнее, чем передавать ссылку на генератор случайных чисел в каждый метод и функцию в качестве аргумента. Достаточно описать внешнюю глобальную переменную (включив соответствующий файл заголовков с помощью оператора #include), и генератор становится доступен. Не нужно менять интерфейс, если вдруг понадобится обратиться к генератору. Не следует передавать один и тот же объект в разные функции.

Тем не менее, использование глобальных переменных может привести к ошибкам. В нашем случае с генератором при его использовании нужно твердо помнить, что глобальная переменная уже определена. Простая забывчивость может привести к тому, что будет определен второй объект – генератор случайных чисел, например с именем randomGen. Поскольку с точки зрения правил языка никаких ошибок допущено не было, компиляция пройдет нормально. Однако результат работы программы будет не тот, которого мы ожидаем. (Исходя из определения класса, ответьте, почему).

При составлении программ самым лучшим решением будет то, которое не позволит ошибиться, т.е. неправильная программа не будет компилироваться. Не всегда это возможно, но в данном случае, как и во многих других, соответствующие средства имеются в языке Си++.

Изменим описание класса RandomGenerator:

```
class RandomGenerator
{
public:
    static void Init(unsigned long start);
    static unsigned long GetNumber(void);
private:
    static unsigned long previousNumber;
};
```

Определения методов Init и GetNumber не изменятся. Единственное, что надо будет добавить в файл RandomGenerator.cpp, это определение переменной previousNumber:

```
//
// файл RandomGenerator.cpp
//
#include "RandomGenerator.h"
#include <time.h>
unsigned long RandomGenerator::previousNumber;
. . .
```

Методы и атрибуты класса, описанные static, существуют независимо от объектов этого класса. Вызов статического метода имеет вид имя_класса::имя_метода, например RandomGenerator::Init(x). У статического метода не существует указателя this, таким образом, он имеет доступ либо к статическим атрибутам класса, либо к атрибутам передаваемых ему в качестве аргументов объектов. Например:

```
class A
{
public:
    static void Method(const A& a);
private:
    static int a1;
    int a2;
};
void
A::Method1(const A& a)
{
    int x = a1;
    int y = a2;
    int z = a.a2;
}   // обращение к статическому атрибуту
  // ошибка, a2 не определен
  // правильно
```

Статический атрибут класса во многом подобен глобальной переменной, но доступ к нему контролируется классом. Один статический атрибут класса создается в начале программы для всех объектов данного класса (даже если ни одного объекта создано не было). Можно считать, что статический атрибут – это атрибут класса, а не объекта.

Теперь программа, использующая генератор случайных чисел, будет

выглядеть так:

```
// файл main.cpp
#include "RandomGenerator.h"
main()
{
  RandomGenerator::Init(1000);
}
void
fun1(void)
{
unsigned long x=RandomGenerator::GetNumber();
    . . .
}

// файл class.cpp
#include "RandomGenerator.h"
Class1::Class1()
{
    . . .
}
void
fun2()
{
unsigned long x=RandomGenerator::GetNumber();
    . . .
}
```

Такое построение программы и удобно, и надежно. В отличие от глобальной переменной, второй раз определить генератор невозможно – мы и первый-то раз определили его лишь фактом включения класса RandomGenerator в программу, а два раза определить один и тот же класс компилятор нам не позволит.

Разумеется, существуют и другие способы сделать так, чтобы существовал только один объект какого-либо класса.

Кратко суммируем результаты этого параграфа:

1. Автоматические переменные заново создаются каждый раз, когда управление передается в соответствующую функцию или блок.
2. Статические и глобальные переменные создаются один раз, в самом начале выполнения программы.
3. К глобальным переменным можно обращаться из всей

программы.

4. К статическим переменным, определенным вне функций, можно обращаться из всех функций данного файла.
5. Хотя использовать глобальные переменные иногда удобно, делать это следует с большой осторожностью, поскольку легко допустить ошибку (нет контроля доступа к ним, можно переопределить глобальную переменную).
6. Статические атрибуты класса существуют в единственном экземпляре и создаются в самом начале выполнения программы. Статические атрибуты применяют тогда, когда нужно иметь одну переменную, к которой могут обращаться все объекты данного класса. Доступ к статическим атрибутам контролируется теми же правилами, что и к обычным атрибутам.
7. Статические методы класса используются для функций, по сути являющихся глобальными, но логически относящихся к какому-либо классу.

15.5 Область видимости имен

Между именами переменных, функций, типов и т.п. при использовании одного и того же имени в разных частях программы могут возникать конфликты. Для того чтобы эти конфликты можно было разрешать, в языке существует такое понятие как область видимости имени.

Минимальной областью видимости имен является блок. Имена, определяемые в блоке, должны быть различны. При попытке объявить две переменные с одним и тем же именем произойдет ошибка. Имена, определенные в блоке, видимы (доступны) в этом блоке и во всех вложенных блоках. Аргументы функции, описанные в ее заголовке, рассматриваются как определенные в теле этой функции.

Имена, объявленные в классе, видимы внутри этого класса, т.е. во всех его методах. Для того чтобы обратиться к атрибуту класса, нужно использовать операции ".", "->" или "::".

Для имен, объявленных вне блоков, областью видимости является весь текст файла, следующий за объявлением.

Объявление может перекрывать такое же имя, объявленное во внешней области.

```
int x = 7;
class A
{
public:
    void foo(int y);
    int x;
};
int main()
{
    A a;
    a.foo(x);
  // используется глобальная переменная x
  // и передается значение 7
    cout << x;
    return 1;
}
void
A::foo(int y)
{
    x = y + 1;
    {
        double x = 3.14;

        cout << x;
    }
    cout << x;
}   // x – атрибут объекта типа A

  // новая переменная x перекрывает
  // атрибут класса x
```

В результате выполнения приведенной программы будет напечатано 3.14, 8 и 7.

Несмотря на то, что имя во внутренней области видимости перекрывает имя, объявленное во внешней области, перекрываемая переменная продолжает существовать. В некоторых случаях к ней можно обратиться, явно указав область видимости с помощью квалификатора "::". Обозначение ::имя говорит о том, что имя относится к глобальной области видимости. (Попробуйте поставить :: перед переменной x в приведенном примере.) Два двоеточия часто употребляют перед именами

стандартных функций библиотеки языка Си++, чтобы, во-первых, подчеркнуть, что это глобальные имена, и, во-вторых, избежать возможных конфликтов с именами методов класса, в котором они употребляются.

Если перед квалификатором поставить имя класса, то поиск имени будет производиться в указанном классе. Например, обозначение A::x показало бы, что речь идет об атрибуте класса A. Аналогично можно обращаться к атрибутам структур и объединений. Поскольку определения классов и структур могут быть вложенными, у имени может быть несколько квалификаторов:

```
class Example
{
public:
    enum Color { RED, WHITE, BLUE };
    struct Structure

        static int Flag;
        int x;
    };
    int y;
    void Method();
};
```

Следующие обращения допустимы извне класса:

```
Example::BLUE
Example::Structure::Flag
```

При реализации метода Method обращения к тем же именам могут быть проще:

```
void
Example::Method()
{
    Color x = BLUE;
    y = Structure::flag;
}
```

При попытке обратиться извне класса к атрибуту набора BLUE компилятор выдаст ошибку, поскольку имя BLUE определено только в контексте класса.

Отметим одну особенность типа enum. Его атрибуты как бы экспортируются во внешнюю область имен. Несмотря на наличие фигурных скобок, к атрибутам перечисленного типа Color не обязательно (хотя и не воспрещается) обращаться Color::BLUE.

15.6 Оператор определения контекста namespace

Несмотря на столь развитую систему областей видимости имен, иногда и ее недостаточно. В больших программах возможность возникновения конфликтов на глобальном уровне достаточно реальна. Имена всех классов верхнего уровня должны быть различны. Хорошо, если вся программа разрабатывается одним человеком. А если группой? Особенно при использовании готовых библиотек классов. Чтобы избежать конфликтов, обычно договариваются о системе имен классов. Договариваться о стиле имен всегда полезно, однако проблема остается, особенно в случае разработки классов, которыми будут пользоваться другие.

Одно из сравнительно поздних добавлений к языку Си++ – контексты, определяемые с помощью оператора namespace. Они позволяют заключить группу объявлений классов, переменных и функций в отдельный контекст со своим именем. Предположим, мы разработали набор классов для вычисления различных математических функций. Все эти классы, константы и функции можно заключить в контекст math для того, чтобы, разрабатывая программу, использующую наши классы, другой программист не должен был бы выбирать имена, обязательно отличные от тех, что мы использовали.

```
namespace math
{
    double const pi = 3.1415;
    double sqrt(double x);
    class Complex
    {
    public:
        . . .
    };
```

```
};
```

Теперь к константе pi следует обращаться math::pi.

Контекст может содержать как объявления, так и определения переменных, функций и классов. Если функция или метод определяется вне контекста, ее имя должно быть полностью квалифицировано

```
double math::sqrt(double x)
{
   . . .
}
```

Контексты могут быть вложенными, соответственно, имя должно быть квалифицировано несколько раз:

```
namespace first
{
   int i;
   namespace second    // первый контекст


 // второй контекст
   {
      int i;
      int whati() { return first::i; }
   // возвращается значение первого i
      int anotherwhat { return i; }
   // возвращается значение второго i

}
first::second::whati();   // вызов функции
```

Если в каком-либо участке программы интенсивно используется определенный контекст, и все имена уникальны по отношению к нему, можно сократить полные имена, объявив контекст текущим с помощью оператора using.

```
double x = pi;          // ошибка, надо использовать math::pi

using namespace math;  // использовать контекст math

double y = pi;          // теперь правильно
```

16 Обработка ошибок

16.1 Виды ошибок

Существенной частью любой программы является обработка ошибок. Прежде чем перейти к описанию средств языка Си++, предназначенных для обработки ошибок, остановимся немного на том,какие, собственно, ошибки мы будем рассматривать.

Ошибки компиляции пропустим:пока все они не исправлены, программа не готова, и запустить ее нельзя. Здесь мы будем рассматривать только ошибки, происходящие во время выполнения программы.

Первый вид ошибок, который всегда приходит в голову – это ошибки программирования. Сюда относятся ошибки в алгоритме, в логике программы и чисто программистские ошибки. Ряд возможных ошибок мы называли ранее (например, при работе с указателями), но гораздо больше вы узнаете на собственном горьком опыте.

Теоретически возможно написать программу без таких ошибок. Во многом язык Си++ помогает предотвратить ошибки во время выполнения программы,осуществляя строгий контроль на стадии компиляции. Вообще, чем строже контроль на стадии компиляции, тем меньше ошибок остается при выполнении программы.

Перечислим некоторые средства языка, которые помогут избежать ошибок:

1. Контроль типов. Случаи использования недопустимых операций и смешения несовместимых типов будут обнаружены компилятором.
2. Обязательное объявление имен до их использования. Невозможно вызвать функцию с неверным числом аргументов. При изменении определения переменной или функции легко обнаружить все места, где она используется.
3. Ограничение видимости имен, контексты имен. Уменьшается возможность конфликтов имен, неправильного переопределения имен.

Самым важным средством уменьшения вероятности ошибок является объектно-ориентированный подход к программированию,который

поддерживает язык Си++. Наряду с преимуществами объектного программирования, о которых мы говорили ранее, построение программы из классов позволяет отлаживать классы по отдельности и строить программы из надежных составных "кирпичиков", используя одни и те же классы многократно.

Несмотря на все эти положительные качества языка, остается "простор" для написания ошибочных программ. По мере рассмотрения свойств языка, мы стараемся давать рекомендации, какие возможности использовать, чтобы уменьшить вероятность ошибки.

Лучше исходить из того, что идеальных программ не существует, это помогает разрабатывать более надежные программы. Самое главное – обеспечить контроль данных, а для этого необходимо проверять в программе все, что может содержать ошибку. Если в программе предполагается какое-то условие, желательно проверить его, хотя бы в начальной версии программы, до того, как можно будет на опыте убедиться, что это условие действительно выполняется. Важно также проверять указатели, передаваемые в качестве аргументов, на равенство нулю; проверять, не выходят ли индексы за границы массива и т.п.

Ну и решающими качествами, позволяющими уменьшить количество ошибок, являются внимательность, аккуратность и опыт.

Второй вид ошибок – "предусмотренные", запланированные ошибки. Если разрабатывается программа диалога с пользователем, такая программа обязана адекватно реагировать и обрабатывать неправильные нажатия клавиш. Программа чтения текста должна учитывать возможные синтаксические ошибки. Программа передачи данных по телефонной линии должна обрабатывать помехи и возможные сбои при передаче. Такие ошибки – это, вообще говоря, не ошибки с точки зрения программы, а плановые ситуации, которые она обрабатывает.

Третий вид ошибок тоже в какой-то мере предусмотрен. Это исключительные ситуации, которые могут иметь место, даже если в

программе нет ошибок. Например, нехватка памяти для создания нового объекта. Или сбой диска при извлечении информации из базы данных.

Именно обработка двух последних видов ошибок и рассматривается в последующих разделах. Граница между ними довольно условна. Например, для большинства программ сбой диска – исключительная ситуация, но для операционной системы сбой диска должен быть предусмотрен и должен обрабатываться. Скорее два типа можно разграничить по тому, какая реакция программы должна быть предусмотрена. Если после плановых ошибок программа должна продолжать работать, то после исключительных ситуаций надо лишь сохранить уже вычисленные данные и завершить программу.

16.2 Возвращаемое значение как признак ошибки

Простейший способ сообщения об ошибках предполагает использование возвращаемого значения функции или метода. Функция сохранения объекта в базе данных может возвращать логическое значение: true в случае успешного сохранения, false – в случае ошибки.

```
class Database
{
public:
  bool SaveObject(const Object&obj);
};
Соответственно, вызов метода должен выглядеть так:
if (database.SaveObject(my_obj) == false ){
  //обработка ошибки
}
```

Обработка ошибки, разумеется, зависит от конкретной программы. Типична ситуация, когда при многократно вложенных вызовах функций обработка происходит на несколько уровней выше, чем уровень, где ошибка произошла. В таком случае результат, сигнализирующий об ошибке, придется передавать во всех вложенных вызовах.

```
int main()
{
```

```
  if (fun1()==false )  //обработка ошибки
  return 1;
}
bool
fun1()
{
  if (fun2()==false )
    return false ;
  return true ;
}
bool
fun2()
{
  if (database.SaveObject(obj)==false )
    return false ;
  return true ;
}
```

Если функция или метод должны возвращать какую-то величину в качестве результата, то особое, недопустимое, значение этой величины используется в качестве признака ошибки. Если метод возвращает указатель, выдача нулевого указателя применяется в качестве признака ошибки. Если функция вычисляет положительное число, возврат - 1 можно использовать в качестве признака ошибки.

Иногда невозможно вернуть признак ошибки в качестве возвращаемого значения. Примером является конструктор объекта, который не может вернуть значение. Как же сообщить о том, что во время инициализации объекта что-то было не так?

Распространенным решением является дополнительный атрибут объекта – флаг, отражающий состояние объекта. Предположим, конструктор класса Database должен соединиться с сервером базы данных.

```
class Database
{
public :
  Database(const char *serverName);
   ...
  bool Ok(void )const {return okFlag;};
private :
```

```
  bool okFlag;
};
Database::Database(const char*serverName)
{
  if (connect(serverName)==true )
    okFlag =true ;
  else
    okFlag =false ;
}
int main()
{
  Database database("db-server");
  if (!database.Ok()){
    cerr <<"Ошибка соединения с базой данных"<<endl;
  return 0;
  }
  return 1;
}
```

Лучше вместо метода Ok, возвращающего значение флага okFlag, переопределить операцию ! (отрицание).

```
class Database
{
public :
   bool operator !()const {return !okFlag;};
};
```

Тогда проверка успешности соединения с базой данных будет выглядеть так:

```
if (!database){
  cerr <<"Ошибка соединения с базой
                  данных"<<endl;
}
```

Следует отметить, что лучше избегать такого построения классов, при котором возможны ошибки в конструкторе. Из конструктора можно выделить соединение с сервером базы данных в отдельный метод Open :

```
class Database
{
public :
   Database();
   bool Open(const char*serverName);
```

```
}
```

и тогда отпадает необходимость в операции ! или методе Ok().

Использование возвращаемого значения в качестве признака ошибки – метод почти универсальный. Он применяется, прежде всего, для обработки запланированных ошибочных ситуаций. Этот метод имеет ряд недостатков. Во-первых, приходится передавать признак ошибки через вложенные вызовы функций. Во-вторых, возникают неудобства, если метод или функция уже возвращают значение, и приходится либо модифицировать интерфейс, либо придумывать специальное "ошибочное" значение. В-третьих, логика программы оказывается запутанной из-за сплошных условных операторов if с проверкой на ошибочное значение.

16.3 Исключительные ситуации

В языке Си++ реализован специальный механизм для сообщения об ошибках – механизм исключительных ситуаций. Название, конечно же, наводит на мысль, что данный механизм предназначен, прежде всего, для оповещения об исключительных ситуациях, о которых мы говорили чуть ранее. Однако механизм исключительных ситуаций может применяться и для обработки плановых ошибок.

Исключительная ситуация возникает при выполнении оператора throw . В качестве аргумента throw задается любое значение. Это может быть значение одного из встроенных типов (число, строка символов и т.п.) или объект любого определенного в программе класса.

При возникновении исключительной ситуации выполнение текущей функции или метода немедленно прекращается, созданные к этому моменту автоматические переменные уничтожаются, и управление передается в точку, откуда была вызвана текущая функция или метод. В точке возврата создается та же самая исключительная ситуация, прекращается выполнение текущей функции или метода, уничтожаются автоматические переменные, и управление передается в точку, откуда была вызвана эта функция или метод. Происходит своего рода откат всех вызовов до тех пор, пока не завершится

функция main и, соответственно, вся программа.

Предположим, из main была вызвана функция foo , которая вызвала метод Open , а он в свою очередь возбудил исключительную ситуацию:

```
class Database
{
public :
   void Open(const char*serverName);
};
void
Database::Open(const char*serverName)
{
   if (connect(serverName)==false )
      throw 2;
}
foo()
{
   Database database;
   database.Open("db-server");
   String y;
   ...
}
int main()
{
   String x;
   foo();
   return 1;
}
```

В этом случае управление вернется в функцию foo , будет вызван деструктор объекта database , управление вернется в main , где будет вызван деструктор объекта x , и выполнение программы завершится. Таким образом, исключительные ситуации позволяют аварийно завершать программы с некоторыми возможностями очистки переменных.

В таком виде оператор throw используется для действительно исключительных ситуаций, которые практически никак не обрабатываются. Гораздо чаще даже исключительные ситуации требуется обрабатывать.

16.4 Обработка исключительных ситуаций

В программе можно объявить блок, в котором мы будем отслеживать исключительные ситуации с помощью операторов try и catch :

```
try {
  ...
}catch (тип_исключительной_операции){
  ...
}
```

Если внутри блока try возникла исключительная ситуация, то она первым делом передается в оператор catch . Тип исключительной ситуации – это тип аргумента throw . Если тип исключительной ситуации совместим с типом аргумента catch , выполняется блок catch . Тип аргумента catch совместим, если он либо совпадает с типом ситуации, либо является одним из ее базовых типов. Если тип несовместим, то происходит описанный выше откат вызовов, до тех пор, пока либо не завершится программа, либо не встретится блок catch с подходящим типом аргумента.

В блоке catch происходит обработка исключительной ситуации.

```
foo()
{
 Database database;
 int attempCount =0;
again:
 try {
  database.Open("dbserver");
 } catch (int&ex){
  cerr <<"Ошибка соединения номер "
                    <<x <<endl;
  if (++attemptCount <5)
    goto again;
  throw ;
 }
 String y;
 ...
}
```

Ссылка на аргумент throw передается в блок catch . Этот блок гасит

исключительную ситуацию. Во время обработки в блоке catch можно создать либо ту же самую исключительную ситуацию с помощью оператора throw без аргументов, либо другую, или же не создавать никакой. В последнем случае исключительная ситуация считается погашенной, и выполнение программы продолжается после блока catch .

С одним блоком try может быть связано несколько блоков catch с разными аргументами. В этом случае исключительная ситуация последовательно "примеряется" к каждому catch до тех пор, пока аргумент не окажется совместимым. Этот блок и выполняется. Специальный вид catch

catch (...)

совместим с любым типом исключительной ситуации. Правда, в него нельзя передать аргумент.

16.5 Примеры обработки исключительных ситуаций

Механизм исключительных ситуаций предоставляет гибкие возможности для обработки ошибок, однако им надо уметь правильно пользоваться. В этом параграфе мы рассмотрим некоторые приемы обработки исключительных ситуаций.

Прежде всего, имеет смысл определить для них специальный класс. Простейшим вариантом является класс, который может хранить код ошибки:

```
class Exception
{
public :
  enum ErrorCode {
    NO_MEMORY,
    DATABASE_ERROR,
    INTERNAL_ERROR,
    ILLEGAL_VALUE
  };
  Exception(ErrorCode errorKind,
    const String&errMessage);
  ErrorCode GetErrorKind(void )const
```

```
        {return kind;};
  const String&GetErrorMessage(void )const
    {return msg;};
private :
  ErrorCode kind;
  String msg;
};
```

Создание исключительной ситуации будет выглядеть следующим образом:

```
if (connect(serverName)==false )
  throw Exception(Exception::DATABASE_ERROR,
            serverName);
```

А проверка на исключительную ситуацию так:

```
try {
  ...
}catch (Exception&e){
  cerr <<"Произошла ошибка "<<e.GetErrorKind()
    <<"Дополнительная информация:"
    <<e.GetErrorMessage();
}
```

Преимущества класса перед просто целым числом состоят, во-первых, в том, что передается дополнительная информация и, во-вторых, в операторах catch можно реагировать только на ошибки определенного вида. Если была создана исключительная ситуация другого типа, например

```
throw AnotherException;
```

то блок catch будет пропущен: он ожидает только исключительных ситуаций типа Exception . Это особенно существенно при сопряжении нескольких различных программ и библиотек – каждый набор классов отвечает только за собственные ошибки.

В данном случае код ошибки записывается в объекте типа Exception . Если в одном блоке catch ожидается несколько разных исключительных

ситуаций, и для них необходима разная обработка, то в программе придется анализировать код ошибки с помощью операторов if или switch .

```
try {
  ...
}catch (Exception&e){
  cerr <<"Произошла ошибка "<<e.GetErrorKind()
    <<"Дополнительная информация:"
    <<e.GetErrorMessage();
  if (e.GetErrorKind()==Exception::NO_MEMORY ||
      e.GetErrorKind()==
       Exception::INTERNAL_ERROR)
    throw ;
  else if (e.GetErrorKind()==
          Exception::DATABASE_ERROR)
    return TRY_AGAIN;
  else if (e.GetErrorKind()==
          Exception::ILLEGAL_VALUE)
    return NEXT_VALUE;
}
```

Другим методом разделения различных исключительных ситуаций является создание иерархии классов – по классу на каждый тип исключительной ситуации.

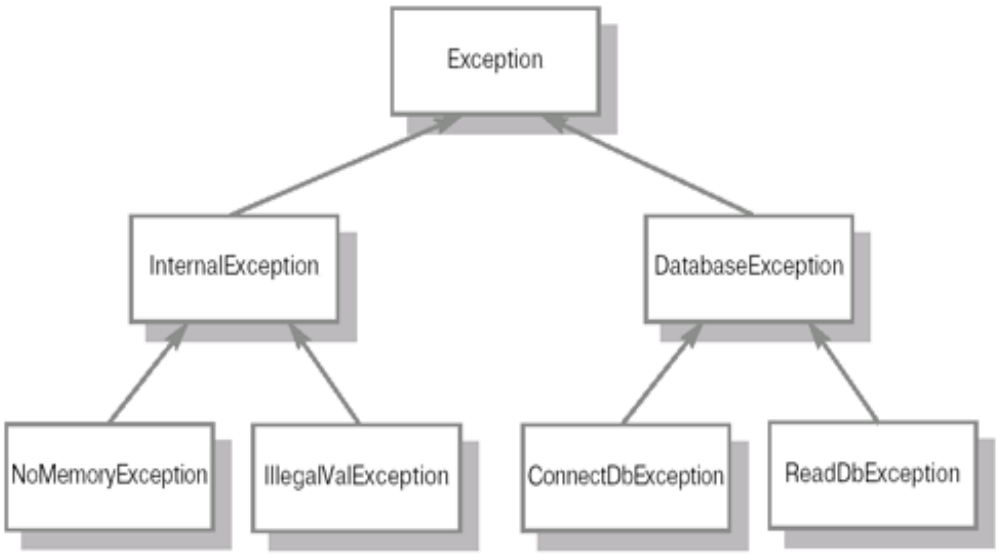

Рис. 16.1. Пример иерархии классов для представления исключительных ситуаций.

В приведенной на рисунке 16.1 структуре классов все исключительные

ситуации делятся на ситуации, связанные с работой базы данных (класс DatabaseException), и внутренние ошибки программы (класс InternalException). В свою очередь, ошибки базы данных бывают двух типов: ошибки соединения (представленные классом ConnectDbException) и ошибки чтения (ReadDbException). Внутренние исключительные ситуации и разделены на нехватку памяти (NoMemoryException)и недопустимые значения (IllegalValException).

Теперь блок catch может быть записан в следующем виде:

```
try {
}catch (ConnectDbException&e ){
  //обработка ошибки соединения с базой данных
}catch (ReadDbException&e){
  //обработка ошибок чтения из базы данных
}catch (DatabaseException&e){
  //обработка других ошибок базы данных
}catch (NoMemoryException&e){
  //обработка нехватки памяти
}catch (…){
  //обработка всех остальных исключительных
  //ситуаций
}
```

Напомним, что когда при проверке исключительной ситуации на соответствие аргументу оператора catch проверка идет последовательно до тех пор, пока не найдется подходящий тип. Поэтому, например, нельзя ставить catch для класса DatabaseException впереди catch для класса ConnectDbException – исключительная ситуация типа ConnectDbException совместима с классом DatabaseException (это ее базовый класс), и она будет обработана в catch для DatabaseException и не дойдет до блока с ConnectDbException .

Построение системы классов для разных исключительных ситуаций на стадии описания ошибок – процесс более трудоемкий, приходится создавать новый класс для каждого типа исключительной ситуации. Однако с точки зрения обработки он более гибкий и позволяет писать более

простые программы.

Чтобы облегчить обработку ошибок и сделать запись о них более наглядной, описания методов и функций можно дополнить информацией, какого типа исключительные ситуации они могут создавать:

```
class Database
{
public :
  Open(const char*serverName)
  throw ConnectDbException;
};
```

Такое описание говорит о том, что метод Open класса Database может создать исключительную ситуацию типа ConnectDbException . Соответственно, при использовании этого метода желательно предусмотреть обработку возможной исключительной ситуации.

В заключение приведем несколько рекомендаций по использованию исключительных ситуаций.

1. При возникновении исключительной ситуации остаток функции или метода не выполняется. Более того, при обработке ее не всегда известно, где именно возникла исключительная ситуация. Поэтому прежде чем выполнить оператор throw , освободите ресурсы, зарезервированные в текущей функции. Например, если какой-либо объект был создан с помощью new , необходимо явно вызвать для него delete .
2. Избегайте использования исключительных ситуаций в деструкторах. Деструктор может быть вызван в результате уже возникшей исключительной ситуации при откате вызовов функций и методов. Повторная исключительная ситуация не обрабатывается и завершает выполнение программы.

Если исключительная ситуация возникла в конструкторе объекта, считается, что объект сформирован не полностью, и деструктор для него вызван не будет.

17 Ввод-вывод

Обмен данными между программой и внешними устройствами осуществляется с помощью операций ввода-вывода. Типичным внешним устройством является терминал. На терминале можно напечатать информацию. Можно ввести информацию с терминала, напечатав ее на клавиатуре. Другим типичным устройством является жесткий или гибкий диск, на котором расположены файлы. Программа может создавать файлы, в которых хранится информация. Другая (или эта же) программа может читать информацию из файла.

В языке Си++ нет особых операторов для ввода или вывода данных. Вместо этого имеется набор классов, стандартно поставляемых вместе с компилятором, которые и реализуют основные операции ввода-вывода.

Причиной является как слишком большое разнообразие операций ввода и вывода в разных операционных системах, особенно графических, так и возможность определения новых типов данных в языке Си++. Вывод даже простой строки текста в MS DOS, MS Windows и в X Window настолько различен, что пытаться придумать общие для всех них операторы было бы слишком негибко и на самом деле затруднило бы работу. Что же говорить о классах, определенных программистом, у которых могут быть совершенно специфические требования к их вводу-выводу.

Библиотека классов для ввода-вывода решает две задачи. Во-первых, она обеспечивает эффективный ввод-вывод всех встроенных типов и простое, но тем не менее гибкое, определение операций ввода-вывода для новых типов, разрабатываемых программистом. Во-вторых, сама библиотека позволяет при необходимости развивать её и модифицировать.

В нашу задачу не входит описание программирования в графических системах типа MS Windows. Мы будем рассматривать операции ввода-вывода файлов и алфавитно-цифровой вывод на терминал, который будет работать на консольном окне MS Windows, MS DOS или Unix.

17.1 Потоки

Механизм для ввода-вывода в Си++ называется потоком. Название произошло от того,что информация вводится и выводится в виде потока байтов – символ за символом.

Класс istream реализует поток ввода, класс ostream – поток вывода. Эти классы определены в файле заголовков iostream.h. Библиотека потоков ввода-вывода определяет три глобальных объекта: cout,cin и cerr. cout называется стандартным выводом, cin – стандартным вводом, cerr – стандартным потоком сообщений об ошибках. cout и cerr выводят на терминал и принадлежат к классу ostream, cin имеет тип istream и вводит с терминала. Разница между cout и cerr существенна в Unix – они используют разные дескрипторы для вывода. В других системах они существуют больше для совместимости.

Вывод осуществляется с помощью операции >>, ввод с помощью операции <<. Выражение

```
cout << "Пример вывода: " << 34;
```

напечатает на терминале строку "Пример вывода", за которым будет выведено число 34. Выражение

```
int x;
cin >> x;
```

введет целое число с терминала в переменную x. (Разумеется, для того, чтобы ввод произошел, на терминале нужно напечатать какое-либо число и нажать клавишу возврат каретки.)

17.2 Операции << и >> для потоков

В классах iostream операции >> и << определены для всех встроенных типов языка Си++ и для строк (тип char*). Если мы хотим использовать такую же запись для ввода и вывода других классов, определенных в программе, для них нужно определить эти операции.

```
class String
```

```
{
public:
  friend  ostream& operator<<(ostream& os,
                        const String& s);
  friend  istream& operator>>(istream& is,
                        String& s);
private:
   char*  str;
   int  length;
};
ostream& operator<<(ostream& os,
                const String& s)
{
   os  <<  s.str;
   return  os;
}
istream& operator>>(istream& is,
                String& s)
{
// предполагается, что строк длиной более
// 1024 байтов не будет
   char  tmp[1024];
   is  >>  tmp;

   if (str  !=  0)  {
      delete  []  str;
   }
   length  =  strlen(tmp);
   str  =  new  char[length  +  1];
   if  (str  ==  0)  {
      // обработка ошибок
      length  =  0;
      return  is;
   }
   strcpy(str,  tmp);
   return  is;
}
```

Как показано в примере класса String, операция <<, во-первых, является не методом класса String, а отдельной функцией. Она и не может быть методом класса String, поскольку ее правый операнд – объект класса ostream. С точки зрения записи, она могла бы быть методом класса ostream, но тогда с добавлением нового класса приходилось бы модифицировать класс ostream,

что невозможно – каждый бы модифицировал стандартные классы, поставляемые вместе с компилятором. Когда же операция << реализована как отдельная функция, достаточно в каждом новом классе определить ее, и можно использовать запись:

```
String x;
   . . .
cout << "this is a string: " << x;
```

Во-вторых, операция << возвращает в качестве результата ссылку на поток вывода. Это позволяет использовать ее в выражениях типа приведенного выше, соединяющих несколько операций вывода в одно выражение.

Аналогично реализована операция ввода. Для класса istream она определена для всех встроенных типов языка Си++ и указателей на строку символов. Если необходимо, чтобы класс, определенный в программе, позволял ввод из потока, для него нужно определить операцию >> в качестве функции friend.

17.3 Манипуляторы и форматирование ввода-вывода

Часто бывает необходимо вывести строку или число в определенном формате. Для этого используются так называемые манипуляторы.

Манипуляторы – это объекты особых типов, которые управляют тем, как ostream или istream обрабатывают последующие аргументы. Некоторые манипуляторы могут также выводить или вводить специальные символы.

С одним манипулятором мы уже сталкивались, это endl. Он вызывает вывод символа новой строки. Другие манипуляторы позволяют задавать формат вывода чисел:

endl	при выводе перейти на новую строку;
ends	вывести нулевой байт (признак конца строки символов);

flush	немедленно вывести и опустошить все промежуточные буферы;
dec	выводить числа в десятичной системе (действует по умолчанию);
oct	выводить числа в восьмеричной системе;
hex	выводить числа в шестнадцатеричной системе счисления;
setw (int n)	установить ширину поля вывода в n символов (n – целое число);
setfill(int n)	установить символ-заполнитель; этим символом выводимое значение будет дополняться до необходимой ширины;
setprecision(int n)	установить количество цифр после запятой при выводе вещественных чисел;
setbase(int n)	установить систему счисления для вывода чисел; n может принимать значения 0, 2, 8, 10, 16, причем 0 означает систему счисления по умолчанию, т.е. 10.

Использовать манипуляторы просто – их надо вывести в выходной поток. Предположим, мы хотим вывести одно и то же число в разных системах счисления:

```
int x = 53;
cout << "Десятичный вид:       " << dec
   << x << endl
   << "Восьмеричный вид:      " << oct
   << x << endl
   << "Шестнадцатеричный вид: " << hex
```

```
    << x << endl
```

Аналогично используются манипуляторы с параметрами. Вывод числа с разным количеством цифр после запятой:

```
double x;
// вывести число в поле общей шириной
// 6 символов (3 цифры до запятой,
// десятичная точка и 2 цифры после запятой)
cout << setw(6) << setprecision(2)
    << x << endl;
```

Те же манипуляторы (за исключением endl и ends могут использоваться и при вводе. В этом случае они описывают представление вводимых чисел. Кроме того, имеется манипулятор, работающий только при вводе, это ws. Данный манипулятор переключает вводимый поток в такой режим, при котором все пробелы (включая табуляцию, переводы строки, переводы каретки и переводы страницы) будут вводиться. По умолчанию эти символы воспринимаются как разделители между атрибутами ввода.

```
int x;
// ввести шестнадцатеричное число
cin >> hex >> x;
```

17.4 Строковые потоки

Специальным случаем потоков являются строковые потоки, представленные классом strstream. Отличие этих потоков состоит в том, что все операции происходят в памяти. Фактически такие потоки формируют форматированную строку символов, заканчивающуюся нулевым байтом. Строковые потоки применяются, прежде всего, для того, чтобы облегчить форматирование данных в памяти.

Например, в приведенном в предыдущей главе классе Exception для исключительной ситуации можно добавить сообщение. Если мы хотим составить сообщение из нескольких частей, то может возникнуть необходимость форматирования этого сообщения:

```
// произошла ошибка
strstream ss;
```

```
ss << "Ошибка ввода-вывода, регистр: "
   << oct << reg1;
ss << "Системная ошибка номер: " << dec
   << errno << ends;
String msg(ss.str());
ss.rdbuf()->freeze(0);
Exception ex(Exception::INTERNAL_ERROR, msg);
throw ex;
```

Сначала создается объект типа strstream с именем ss. Затем в созданный строковый поток выводятся сформатированные нужным образом данные. Отметим, что в конце мы вывели манипулятор ends, который добавил необходимый для символьной строки байтов нулевой байт. Метод str() класса strstream предоставляет доступ к сформатированной строке (тип его возвращаемого значения – char*). Следующая строка освобождает память, занимаемую строковым потоком (подробнее об этом рассказано ниже). Последние две строки создают объект типа Exception с типом ошибки INTERNAL_ERROR и сформированным сообщением и вызывают исключительную ситуацию.

Важное свойство класса strstream состоит в том, что он автоматически выделяет нужное количество памяти для хранения строк. В следующем примере функция split_numbers выделяет числа из строки, состоящей из нескольких чисел, разделенных пробелом, и печатает их по одному на строке.

```
#include <strstream.h>
void
split_numbers(const char* s)
{
   strstream iostr;
   iostr << s << ends;
   int x;
   while (iostr >> x)
   cout << x<< endl;
}
int
main()
{
   split_numbers("123 34 56 932");
   return 1;
```

```
}
```

Замечание. В среде Visual C++ файл заголовков называется strstrea.h.

Как видно из этого примера, независимо от того, какова на самом деле длина входной строки, объект iostr автоматически выделяет память, и при выходе из функции split_numbers, когда объект уничтожается, память будет освобождена.

Однако из данного правила есть одно исключение. Если программа обращается непосредственно к хранимой в объекте строке с помощью метода str (), то объект перестает контролировать эту память, а это означает, что при уничтожении объекта память не будет освобождена. Для того чтобы память все-таки была освобождена, необходимо вызвать метод rdbuf()->freeze(0) (см. предыдущий пример).

17.5 Ввод-вывод файлов

Ввод-вывод файлов может выполняться как с помощью стандартных функций библиотеки Си, так и с помощью потоков ввода-вывода. Функции библиотеки Си являются функциями низкого уровня, без всякого контроля типов.

Прежде чем перейти к рассмотрению собственно классов, остановимся на том, как осуществляются операции ввода-вывода с файлами. Файл рассматривается как последовательность байтов. Чтение или запись выполняются последовательно. Например, при чтении мы начинаем с начала файла. Предположим, первая операция чтения ввела 4 байта, интерпретированные как целое число. Тогда следующая операция чтения начнет ввод с пятого байта, и так далее до конца файла.

Аналогично происходит запись в файл – по умолчанию первая запись производится в конец имеющегося файла, а все последующие операции записи последовательно пишут данные друг за другом. При операциях чтения-записи говорят, что существует текущая позиция, начиная с которой будет производиться следующая операция.

Большинство файлов обладают возможностью прямого доступа. Это

означает, что можно производить операции ввода-вывода не последовательно, а в произвольном порядке: после чтения первых 4-х байтов прочесть с 20 по 30, затем два последних и т.п. При написании программ на языке Си++ возможность прямого доступа обеспечивается тем, что текущую позицию чтения или записи можно установить явно.

В библиотеке Си++ для ввода-вывода файлов существуют классы ofstream (вывод) и ifstream (ввод). Оба они выведены из класса fstream. Сами операции ввода-вывода выполняются так же, как и для других потоков – операции >> и << определены для класса fstream как "ввести" и "вывести" соответствующее значение. Различия заключаются в том, как создаются объекты и как они привязываются к нужным файлам.

При выводе информации в файл первым делом нужно определить, в какой файл будет производиться вывод. Для этого можно использовать конструктор класса ofstream в виде:

```
ofstream(const char* szName,
        int nMode = ios::out,
        int  nProt  =  filebuf::openprot);
```

Первый аргумент – имя выходного файла, и это единственный обязательный аргумент. Второй аргумент задает режим, в котором открывается поток. Этот аргумент – битовое ИЛИ следующих величин:

ios::app	при записи данные добавляются в конец файла, даже если текущая позиция была перед этим перемещена;
ios::ate	при создании потока текущая позиция помещается в конец файла; однако, в отличие от режима app, запись ведется в текущую позицию;
ios::in	поток создается для ввода; если файл уже существует, он сохраняется;
ios::out	поток создается для вывода (режим по умолчанию);

ios::trunc	если файл уже существует, его прежнее содержимое уничтожается, и длина файла становится равной нулю; режим действует по умолчанию, если не заданы ios::ate, ios::app или ios::in;
ios::binary	ввод-вывод будет происходить в двоичном виде, по умолчанию используется текстовое представление данных.

Третий аргумент используется только в том случае, если создается новый файл; он определяет параметры создаваемого файла.

Можно создать поток вывода с помощью стандартного конструктора без аргументов, а позднее выполнить метод open с такими же аргументами, как у предыдущего конструктора:

```
void open(const char* szName,
        int nMode = ios::out,
        int nProt = filebuf::openprot);
```

Только после того, как поток создан и соединен с определенным файлом (либо с помощью конструктора с аргументами, либо с помощью метода open), можно выполнять вывод. Выводятся данные операцией <<. Кроме того, данные можно вывести с помощью методов write или put:

```
ostream& write(const char* pch,
          int nCount);
ostream& put(char ch);
```

Метод write выводит указанное количество байтов (nCount), расположенных в памяти, начиная с адреса pch. Метод put выводит один байт.

Для того чтобы переместить текущую позицию, используется метод seekp:

```
ostream& seekp(streamoff off,
          ios::seek_dir dir);
```

Первый аргумент – целое число, смещение позиции в байтах. Второй аргумент определяет, откуда отсчитывается смещение; он может принимать одно из трех значений:

ios::beg	смещение от начала файла
ios::cur	смещение от текущей позиции
ios::end	смещение от конца файла

Сместив текущую позицию, операции вывода продолжаются с нового места файла.

После завершения вывода можно выполнить метод close, который выводит внутренние буферы в файл и отсоединяет поток от файла. То же самое происходит и при уничтожении объекта.

Класс ifstream, осуществляющий ввод из файлов, работает аналогично. При создании объекта типа ifstream в качестве аргумента конструктора можно задать имя существующего файла:

```
ifstream(const char* szName, int nMode = ios::in,
      int nProt = filebuf::openprot);
```

Можно воспользоваться стандартным конструктором, а подсоединиться к файлу с помощью метода open.

Чтение из файла производится операцией >> или методами read или get:

```
istream& read(char* pch, int nCount);
istream& get(char& rch);
```

Метод read вводит указанное количество байтов (nCount) в память, начиная с адреса pch. Метод get вводит один байт.

Так же, как и для вывода, текущую позицию ввода можно изменить с помощью метода seekp, а по завершении выполнения операций закрыть файл с помощью close или просто уничтожить объект.

18 Шаблоны

18.1 Назначение шаблонов

Алгоритм выполнения какого-либо действия можно записывать независимо от того, какого типа данные обрабатываются. Простейшим примером служит определение минимума из двух величин.

```
if (a < b)
    x = a;
else
    x = b;
```

Независимо от того, к какому именно типу принадлежат переменные a, b и x, если это один и тот же тип, для которого определена операция "меньше", запись будет одна и та же. Было бы естественно определить функцию min, возвращающую минимум из двух своих аргументов. Возникает вопрос, как описать аргументы этой функции? Конечно, можно определить min для всех известных типов, однако, во-первых, пришлось бы повторять одну и ту же запись многократно, а во-вторых, с добавлением новых классов добавлять новые функции.

Аналогичная ситуация встречается и в случае со многими сложными структурами данных. В классе, реализующем связанный список целых чисел, алгоритмы добавления нового атрибута списка, поиска нужного атрибута и так далее не зависят от того, что атрибуты списка – целые числа. Точно такие же алгоритмы нужно будет реализовать для списка вещественных чисел или указателей на класс Book.

Механизм шаблонов в языке Си++ позволяет эффективно решать многие подобные задачи.

18.2 Функции-шаблоны

Запишем алгоритм поиска минимума двух величин, где в качестве параметра используется тип этих величин.

```
template <class T>
const T&  min(const T&  a, const T&  b)
{
```

```
    if (a < b)
        return a;
    else
        return b;
}
```

Данная запись еще не создала ни одной функции, это лишь шаблон для определенной функции. Только тогда, когда происходит обращение к функции с аргументами конкретного типа, будет выполнена генерация конкретной функции.

```
int x, y, z;
String s1, s2, s3;
. . .
// генерация функции min для класса String
s1 = min(s2, s3);
. . .
// генерация функции min для типа int
x = min(y, z);
```

Первое обращение к функции min генерирует функцию

```
const String& min(const String& a,
                  const String& b);
```

Второе обращение генерирует функцию

```
const int& min(const int& a,
               const int& b);
```

Объявление шаблона функции min говорит о том, что конкретная функция зависит от одного параметра – типа T. Первое обращение к max в программе использует аргументы типа String. В шаблон функции подставляется тип String вместо T. Получается функция:

```
const String& min(const String& a,
                  const String& b)
{
  if (a < b)
    return a;
  else
    return b;
}
```

Эта функция компилируется и используется в программе. Аналогичные

действия выполняются и при втором обращении, только теперь вместо параметра T подставляется тип int. Как видно из приведенных примеров, компилятор сам определяет, какую функцию надо использовать, и автоматически генерирует необходимое определение.

У функции-шаблона может быть несколько параметров. Так, например, функция find библиотеки STL (стандартной библиотеки шаблонов), которая ищет первый элемент, равный заданному, в интервале значений, имеет вид:

```
template <class InIterator, class T>
InIterator
find(InIterator first, InIterator last,
    const T& val);
```

Класс T **– это тип элементов интервала. Тип** InIterator **– тип указателя на его начало и конец.**

18.3 Шаблоны классов

Шаблон класса имеет вид:

```
template <список параметров>
class объявление_класса
```

Список параметров класса-шаблона аналогичен списку параметров функции-шаблона: список классов и переменных, которые подставляются в объявление класса при генерации конкретного класса.

Очень часто шаблоны используются для создания коллекций, т.е. классов, которые представляют собой набор объектов одного и того же типа. Простейшим примером коллекции может служить массив. Массив, несомненно, очень удобная структура данных, однако у него имеется ряд существенных недостатков, к которым, например, относятся необходимость задавать размер массива при его определении и отсутствие контроля использования значений индексов при обращении к атрибутам массива.

Попробуем при помощи шаблонов устранить два отмеченных недостатка у одномерного массива. При этом по возможности попытаемся сохранить синтаксис обращения к атрибутам массива. Назовем новую структуру данных вектор vector.

```
template <class T>
class vector
{
public:
  vector() : nItem(0), items(0) {};
  ~vector() { delete items; };
  void insert(const T& t)
    { T* tmp = items;
    items = new T[nItem + 1];
    memcpy(items, tmp, sizeof(T)* nItem);
    item[++nItem] = t;
    delete tmp; }
  void remove(void)
    { T* tmp = items;
    items = new T[--nItem];
    memcpy(items, tmp, sizeof(T) * nItem);
    delete tmp;
    }
  const T& operator[](int index) const
    {
    if ((index < 0) || (index >= nItem))
      throw IndexOutOfRange;
    return items[index];
    }
  T& operator[](int index)
    {
    if ((index < 0) || (index >= nItem))
      throw IndexOutOfRange;
    return items[index];
    }
private:
  T* items;
  int nItem;
};
```

Кроме конструктора и деструктора, у нашего вектора есть только три метода: метод insert добавляет в конец вектора новый элемент, увеличивая длину вектора на единицу, метод remove удаляет последний элемент вектора, уменьшая его длину на единицу, и операция [] обращается к n-ому элементу вектора.

```
vector<int> IntVector;
IntVector.insert(2);
```

```
IntVector.insert(3);
IntVector.insert(25);
// получили вектор из трех атрибутов:
// 2, 3 и 25
// переменная x получает значение 3
int x = IntVector[1];
// произойдет исключительная ситуация
int y = IntVector[4];
// изменить значение второго атрибута вектора.
IntVector[1] = 5;
```

Обратите внимание, что операция [] определена в двух вариантах – как константный метод и как неконстантный. Если операция [] используется справа от операции присваивания (в первых двух присваиваниях), то используется ее константный вариант, если слева (в последнем присваивании) – неконстантный. Использование операции индексирования [] слева от операции присваивания означает, что значение объекта изменяется, соответственно, нужна неконстантная операция.

Параметр шаблона vector – любой тип, у которого определены операция присваивания и стандартный конструктор. (Стандартный конструктор необходим при операции new для массива.)

Так же, как и с функциями-шаблонами, при задании первого объекта типа vector<int> автоматически происходит генерация конкретного класса из шаблона. Если далее в программе будет использоваться вектор вещественных чисел или строк, значит, будут сгенерированы конкретные классы и для них. Генерация конкретного класса означает, что генерируются все его методы, соответственно, размер исходного кода растет. Поэтому из небольшого шаблона может получиться большая программа. Ниже мы рассмотрим одну возможность сокращения размера программы, использующей почти однотипные шаблоны.

Сгенерировать конкретный класс из шаблона можно явно, записав:

```
template vector<int>;
```

Этот оператор не создаст никаких объектов типа vector<int>, но, тем не менее, вызовет генерацию класса со всеми его методами.

18.4 "Интеллигентный указатель"

Рассмотрим еще один пример использования класса-шаблона. С его помощью мы попытаемся " усовершенствовать" указатели языка Си++. Если указатель указывает на объект, выделенный с помощью операции new, необходимо явно вызывать операцию delete тогда, когда объект становится не нужен. Однако далеко не всегда просто определить, нужен объект или нет, особенно если на него могут ссылаться несколько разных указателей. Разработаем класс, который ведет себя очень похоже на указатель, но автоматически уничтожает объект, когда уничтожается последняя ссылка на него. Назовем этот класс "интеллигентный указатель" (Smart Pointer). Идея заключается в том, что настоящий указатель мы окружим специальной оболочкой. Вместе со значением указателя мы будем хранить счетчик – сколько других объектов на него ссылается. Как только значение этого счетчика станет равным нулю, объект, на который указатель указывает, пора уничтожать.

Структура Ref хранит исходный указатель и счетчик ссылок.

```
template <class T>
struct Ref
{
  T* realPtr;
  int counter;
};
```

Теперь определим интерфейс "интеллигентного указателя":

```
template <class T<
class SmartPtr
{
public:
  // конструктор из обычного указателя
  SmartPtr(T* ptr = 0);
  // копирующий конструктор
  SmartPtr(const SmartPtr& s);
  ~SmartPtr();
  SmartPtr& operator=(const SmartPtr& s);
  SmartPtr& operator=(T* ptr);
```

```
  T* operator->() const;
  T& operator*() const; private:
  Ref<T>* refPtr;
};
```

У класса **SmartPtr** определены операции обращения к элементу **->,** взятия по адресу **"*"** и операции присваивания. С объектом класса **SmartPtr** можно обращаться практически так же, как с обычным указателем.

```
struct A
{
    int x;
    int y;
};
SmartPtr<A> aPtr(new A);
int x1 = aPtr->x;
(*aPtr).y = 3;

 // создать новый указатель
 // обратиться к элементу A
 // обратиться по адресу
```

Рассмотрим реализацию методов класса **SmartPtr**. Конструктор инициализирует объект указателем. Если указатель равен нулю, то **refPtr** устанавливается в ноль. Если же конструктору передается ненулевой указатель, то создается структура **Ref**, счетчик обращений в которой устанавливается в **1**, а указатель – в переданный указатель:

```
template <class T>
SmartPtr<T>::SmartPtr(T* ptr)
{
 if (ptr == 0)
   refPtr = 0;
 else {
   refPtr = new Ref<T>;
   refPtr->realPtr = ptr;
   refPtr->counter = 1;
 }
}
```

Деструктор уменьшает количество ссылок на 1 и, если оно достигло 0, уничтожает объект

```
template <class T>
SmartPtr <T>::~SmartPtr()
{
  if (refPtr != 0) {
    refPtr->counter--;
    if (refPtr->counter <= 0) {
      delete refPtr->realPtr;
      delete refPtr;
    }
  }
}
```

Реализация операций **->** и ***** довольно проста:

```
template <class T>
T*
SmartPtr<T>::operator->() const
{
  if (refPtr != 0)
    return refPtr->realPtr;
  else
    return 0;
}
template <class T>
T&
SmartPtr<T>::operator*() const
{
  if (refPtr != 0)
    return *refPtr->realPtr;
  else
    throw bad_pointer;
}
```

Самые сложные для реализации – копирующий конструктор и операции присваивания. При создании объекта **SmartPtr** – копии имеющегося – мы не будем копировать сам исходный объект. Новый "интеллигентный указатель" будет ссылаться на тот же объект, мы лишь увеличим счетчик ссылок.

```
template <class T>
SmartPtr<T>::SmartPtr(const
          SmartPtr& s):refPtr(s.refPtr)
{
  if (refPtr != 0)
    refPtr->counter++;
```

```
}
```

При выполнении присваивания, прежде всего, нужно отсоединиться от имеющегося объекта, а затем присоединиться к новому, подобно тому, как это сделано в копирующем конструкторе.

```
template <class T>
SmartPtr&
SmartPtr<T>::operator=(const SmartPtr& s)
{
  // отсоединиться от имеющегося указателя
  if (refPtr != 0) {
    refPtr->counter--;
    if (refPtr->counter <= 0) {
      delete refPtr->realPtr;
      delete refPtr;
    }
  }
  // присоединиться к новому указателю
  refPtr = s.refPtr;
  if (refPtr != 0)
    refPtr->counter++;
}
```

В следующей функции при ее завершении объект класса **Complex** будет уничтожен:

```
void foo(void)
{
  SmartPtr<Complex> complex(new Complex);
  SmartPtr<Complex> ptr = complex;
  return;
}
```

18.5 Задание свойств класса

Одним из методов использования шаблонов является уточнение поведения с помощью дополнительных параметров шаблона. Предположим, мы пишем функцию сортировки вектора:

```
template <class T>
void sort_vector(vector<T>& vec)
{
  for (int i = 0; i < vec.size() -1; i++)
```

```
    for (int j = i; j < vec.size(); j++) {
      if (vec[i] < vec[j]) {
        T tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
      }
    }
}
```

Эта функция будет хорошо работать с числами, но если мы захотим использовать ее для массива указателей на строки (**char***), то результат будет несколько неожиданный. Сортировка будет выполняться не по значению строк, а по их адресам (операция "меньше" для двух указателей – это сравнение значений этих указателей, т.е. адресов величин, на которые они указывают, а не самих величин). Чтобы исправить данный недостаток, добавим к шаблону второй параметр:

```
template <class T, class Compare>
void sort_vector(vector<T>& vec)
{
  for (int i = 0; i < vec.size() -1; i++)
    for (int j = i; j < vec.size(); j++) {
      if (Compare::less(vec[i], vec[j])) {
        T tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
      }
    }
}
```

Класс **Compare** должен реализовывать статическую функцию **less**, сравнивающую два значения типа **T**. Для целых чисел этот класс может выглядеть следующим образом:

```
class CompareInt
{
  static bool less(int a, int b)
    { return a < b; };
};
```

Сортировка вектора будет выглядеть так:

```
vector<int> vec;
```

```
sort<int, CompareInt>(vec);
```

Для указателей на байт (строк) можно создать класс

```
class CompareCharStr
{
 static bool less(char* a, char* b)
   { return strcmp(a,b) >= 0; }
};
```

и, соответственно, сортировать с помощью вызова

```
vector<char*> svec;
sort<char*, CompareCharStr>(svec);
```

Как легко заметить, для всех типов, для которых операция "меньше" имеет нужный нам смысл, можно написать шаблон класса сравнения:

```
template<class T> Compare
{
 static bool less(T a, T b)
   { return a < b; };
};
```

и использовать его в сортировке (обратите внимание на пробел между закрывающимися угловыми скобками в параметрах шаблона; если его не поставить, компилятор спутает две скобки с операцией сдвига):

```
vector<double> dvec;
sort<double, Compare<double> >(dvec);
```

Чтобы не загромождать запись, воспользуемся возможностью задать значение параметра по умолчанию. Так же, как и для аргументов функций и методов, для параметров шаблона можно определить значения по умолчанию. Окончательный вид функции сортировки будет следующий:

```
template <class T, class C = Compare<T> >
void sort_vector(vector<T>& vec)
{
 for (int i = 0; i < vec.size() -1; i++)
   for (int j = i; j < vec.size(); j++) {
     if (C::less(vec[i], vec[j])) {
       T tmp = vec[i];
       vec[i] = vec[j];
```

```
        vec[j] = tmp;
      }
    }
  }
```

Второй параметр шаблона иногда называют параметром-штрих, поскольку он лишь модифицирует поведение класса, который манипулирует типом, определяемым первым параметром.

Практические занятия

1. ТИПЫ, ОПЕРАЦИИ, ВЫРАЖЕНИЯ

2.1. Верно ли записаны константы, представляющие целочисленные значения? Для верно записанных констант определить их значение, тип.

123	1E6	123456789LU-5	0XFUL	
‘0’	058	‘\x7’	0X-1AD	‘\122’
00123	0xffffffL	01A	-‘x’	″x″
‘a’U	0731UL	‘\n’	+0xaf	0X0

2.2. Верно ли записаны константы с плавающей точкой? Для верно записанных констант определить их значение, тип.

1.71	1E-6	0.314159E1F	.005	0051E-04
5.E+2	0e0	0x1A1.5	05.5	0
0X1E6	0F	1234.56789L	1.0E-10D	3.1415U
1e-2f	-12.3E-6	+10e6	123456L	E-6

2.3. Верно ли записаны выражения? Для верно записанных выражений вычислить их значения (операции + - * / % =):

int a, b, c, d, e;

a = 2; b = 13; c = 7; d = 19; e = -4;

b / a / c	d / a % c	c % d-e	-e % a + b / a ∗-5+5
b % e	7-d%+(3-a)	b % - e ∗ c	9 / c - - 20 / d

2.4. Верно ли решена задача: «значение целочисленной переменной c увеличить на 1; целочисленной переменной a присвоить значение, равное удвоенному значению переменной c ».

int a, c; c = 5;

a). c ++ ; b). a = 2 ∗ c++ ; c). c += 1; d). a = c++ + c;

a = 2 * c; a = c + c;

e). ++c; f). a = ++ c + c; g). a = c += 1 + c; h). a = (c+=1)+c;

a = c + c;

2.5. Верно ли решена задача: «значение целочисленной переменной с уменьшить на 1; целочисленной переменной а присвоить значение, равное частному от деления переменной с на 2».

int a, c; c = 5;

a). -- c ; b). a = -- c / 2; c). c -= 1; d). a = c -- / 2;

a = c / 2; a = c % 2;

e). a = c -= 1/2; f). a = (c = c - 1)/2; g). a = (c -= 1)/2; h). a=(c-= 1)/2.0;

2.6. Эквивалентны ли выражения?

a) E1 op= E2 и E1 = E1 op E2
b) E1 op= E2 и E1 = E1 op (E2)

Замечание: здесь E1, E2 - выражения допустимого в этом случае типа ; op - операция (одна из + - * / % >> << & ^ |).

2.7. Верно ли записаны выражения? Для верно записанных выражений вычислить их значения (операции + - * / ++ - - операции присваивания):

int a, b, c; a = 2; b = 6; c = 3;

- - - a -- - a b-- - a a += a ++ ++ b / a ++ * --c

a --- b - a-- -b a ++ = b a = a ++ b++ / ++a * c --

- --a a- --c a ++ = a ++ a = b a = (b + 1) ++

2.8. Верно ли записаны выражения? Для верно записанных выражений вычислить их значения, определить тип результата (операции + - * / % ++ операции отношения, операции присваивания):

int i, j, k, m; char c, d; i = 1; j = 2; k = -7; m = 0; c = ‘w’;

d = ’a’+1 < c	m = - i - 5 ∗ j >= k+1	i + j++ + k = = -2∗j
m = 3 < j < 5	m = 3 = = j < 5	m = = c = ’w’
m = c != 87	m = c = ! 87	m = ! c = 87
m = !c+87	! m = =c + 87	m ! = c + 87
k = = j - 9 = = i	k ∗= 3 + j	i + j = !k
i += ++ j + 3	k %= m = 1 + n / 2	1 + 3 ∗ n += 7 / 5
1 + 3 ∗ (n += 7) / 5	c + i < c - ‘x’+10	i - k = = ‘0’+9 < 10

2.9. В логике справедливы утверждения:

not (not x) = x

x and true = x

Верны ли соответствующие утверждения для операций ! и && в Си? Ответ обосновать.

2.10. При любом вещественном y > 0 x < x + y математически верно. Верно ли подобное утверждение для выражения на Си?

2.11. Написать эквивалентное выражение, не содержащее операции **!**

! (a>b)	! (2∗a == b+4)	! (a<b && c<d)
! (a<2 || a>5)	! (a<1 || b<2 && c<3)	

2.12. Пусть

char c; short s; int i; unsigned u; signed char sc;

float f; double d; long lng; unsigned short us; long double ld;

Определить тип выражений:

c - s / i	u ∗ 3 - 3.0 ∗ u - i	u - us ∗ i	(sc + d) ∗ ld

(5 ∗ lng - ‘a’) ∗ (s + u / 2) (f + 3) / (2.5f - s ∗ 3.14)

2.13. Допустимо ли в Си? Если "да" - опишите семантику этих действий; если "нет" - объясните почему.

a). . . .

```
int  i;
i = (1 || 2) % (1 | 2 );
printf ( ″ i = %d\n″, i);
```

b). . . .

```
int  a, b, m, n, z;
m = n = 5;
z = a = b = 0;
z--, ( a = b ) = z + ( m != n );
printf (″%d %d %d %d %d\n″,
        a, b, m, n, z);
```

c). . . .

```
int  i = 1;
i = i << i | i;
printf ( ″ i = %d\n″, i);
```

d). . . .

```
double  x = 1.9; int a;
double  b = 3.7;
a = b += (1 && 2 || 3) != (int)x;
printf (″%f %d %f\n″, x, a, b);
```

e). . . .

```
int x;
x = 5; ++ x =10;
printf ("%d\n", x);
```

f). . . .

```
int i, x, y;  x = 5; y = 10; i = 15;
x = ( y = 0, i = 1);
printf("%d %d %d\n", i, x ,y);
( x = y == 0) , i=1;
printf("%d %d %d\n", i, x, y);
```

g). . . .

```
int x, y;
x = 5; y = x && ++ x;
printf("%d %d\n", x, y);
```

h). . . .

```
int x = 2, y, z;
x *= 3+2; x *= y = z = 4;
printf ("%d %d %d\n", x, y, z);
x = y == z; x == ( y = z );
printf ("%d %d %d\n", x, y, z);
```

i). . . .

```
int x = 2, y = 1, z = 0;
```

j). . . .

```
int x = 03, y = 02, z = 01;
```

```
y = x && y || z;
x = x || !y && z;
z = x / ++x;
printf(" %d %d %d\n", x, y, z);
```

k). . . .

```
int x, y, z;  x = y = z = 1;
x += y += z;
printf("%d\n", x < y ? y++ : x++);
printf("%d\n", z+=x<y ? ++x : y--);
printf("%d %d %d\n", x, y, z);
printf("%d\n", z>=y && y>=x);
```

```
printf("%d\n", x | y & -z);
printf("%d\n", x ^ y & -z);
printf("%d\n", x & y && z);
printf("%d\n", x<<3);
```

l). . . .

```
int x, y, z, i;  x = y = z = 1;
i = ++x || ++y && ++z;
printf("%d%d%d%d\n", x,y,z,i);
i = x++ <= --y || ++z >= i;
printf("%d%d%d%d\n", x,y,z,i);
```

2.14. Что будет напечатано в результате выполнения следующего фрагмента программы?

```
. . .
double d; float f; long lng; int i; short s;
s = i = lng = f = d = 100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
d = f = lng = i = s =100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
s = i = lng = f = d = 1000000/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
d = f = lng = i = s =1000000/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
lng = s = f = i = d =100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
f = s = d = lng = i = (double)100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

```
s = i = lng = f = d = 100/(double)3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
f = s = d = lng = i = (double)100/3;
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
i = s = lng = d = f = (double)(100/3);
printf("s = %hd i = %d lng = %ld f = %f d = %f\n", s, i, lng, f, d);
```

2.15. Что будет напечатано в результате выполнения следующего фрагмента программы?

```
double d = 3.2, x; int i = 2, y;
x = ( y = d / i ) * 2; printf ("x = %f ;y = %d\n", x, y);
x = ( y = d / i ) * 2; printf ("x = %d ;y = %f\n", x, y);
y = ( x = d / i ) * 2; printf ("x = %f ;y = %d\n", x, y);
y = d * ( x = 2.5 / d); printf ("x = %f; y = %d\n", x, y);
x = d * ( y = ( (int)2.9 + 1.1) / d; printf ("x = %d y = %f\n", x, y);
```

2.16. Дано вещественное число x . Не пользуясь никакими операциями, кроме умножения, сложения и вычитания, вычислить

$$2x^4-3x^3+4x^2-5x+6.$$

Разрешается использовать не более четырех умножений и четырех сложений и вычитаний.

2.17. Целой переменной k присвоить значение, равное третьей от конца цифре в записи целого положительного числа x.

2.18. Целой переменной k присвоить значение, равное сумме цифр в записи целого положительного трехзначного числа x.

2.19. Целой переменной k присвоить значение, равное первой цифре дробной части в записи вещественного положительного числа x.

2.20. Определить число, полученное выписыванием в обратном порядке цифр заданного целого трехзначного числа.

2.21. Идет n-ая секунда суток. Определить, сколько полных часов и полных минут прошло к этому моменту.

2.22. Дано вещественное число x . Не пользуясь никакими операциями, кроме умножения, получить

a) x^{21} за шесть операций
b) x^3 и x^{10} за четыре операции
c) x^5 и x^{13} за пять операций
d) x^2, x^5 и x^{17} за шесть операций
e) x^4, x^{12} и x^{28} за шесть операций

2.23. Выражения, соединенные операциями && и || , по правилам Си вычисляются слева направо; вычисления прекращаются, как только становится известна истинность или ложность результата. В других языках программирования, например в Паскале, вычисляются все части выражения в любом случае. Приведите «за» и «против» каждого из этих решений.

2.24. Почему в Си не допускается, чтобы один и тот же литерал-перечислитель входил в два различных перечислимых типа? Могут ли совпадать имена литералов-перечислителей и имена обычных переменных в одной области видимости? Могут ли разные литералы-перечислители иметь одинаковые значения?

2.25. «Упаковать» четыре символа в беззнаковое целое. Длина беззнакового целого равна 4.

2.26. «Распаковать» беззнаковое целое число в четыре символа. Длина беззнакового целого равна 4.

2.27. Заменить в целочисленной переменной x n бит, начиная с позиции p, n старшими инвертированными битами целочисленной переменной y.

2.28. Циклически сдвинуть значение целочисленной величины на n позиций вправо.

2.29. Циклически сдвинуть значение целочисленной величины на n позиций влево.

2.30. Выясните некоторые свойства и особенности поведения доступного Вам транслятора Си:

a) выяснить, сколько байт отведено для хранения данных типа short, int, long, float, double и long double;

b) выяснить способ представления типа char (signed- или unsigned- вариант);

c) проконтролировать, все ли способы записи констант допустимы:

- целых (обычная форма записи, u/U, l/L, их комбинации; запись констант в восьмеричной и шестнадцатиричной системах счисления)
- вещественных (обычная форма записи, в экспоненциальном виде, f/F, l/L, e/E)
- символьных (обычная форма записи, с помощью эскейп-последовательности) и строковых (в частности, происходит ли конкатенация рядом расположенных строковых констант)

d) выяснить, как упорядочены коды символов '0' - '9', 'a' - 'z', 'A' - 'Z', пробел (между собой и относительно друг друга);

e) проконтролировать, происходит ли инициализация переменных по умолчанию;

f) проверить, реагирует ли транслятор на попытку изменить константу;

g) исследовать особенности выполнения операции % с отрицательными операндами;

h) проверьте, действительно ли операции отношения == и != имеют более низкий приоритет, чем все другие операции отношения;

i) проверьте, действительно ли выполняется правило "ленивых вычислений" выражений в Си, т.е. прекращается ли вычисление выражений с логическими операциями, если возможно "досрочно" установить значение результата;

j) проверьте, все ли виды операнда операции sizeof (X), определяемые стандартом для арифметических типов, допускаются компилятором; действительно ли выражение X не вычисляется.

2. УПРАВЛЕНИЕ

3.1 Синтаксис и семантика операторов языка Си

3.1. Перечислить все ситуации, когда в программах на Си используется составной оператор.

3.2. В Си точка с запятой используется в качестве признака конца оператора; в Паскале - в качестве разделителя операторов. Сравните эти решения, сформулируйте возможные «за» и «против».

3.3. Эквивалентны ли следующие фрагменты программы:

```
if (e1) if (e2) S1; else S2;

if (e1) { if (e2) S1; else S2; }

if (e1) { if (e2) S1; } else S2;

if (e1)  if (e2) S1; else ; else S2;

if (e1)  if (e2) S1; else S2; else ;
```

Замечание: здесь e1 и e2 - выражения допустимого в этом случае типа; S1 и S2 - произвольные операторы.

3.4. Описать в виде блок-схемы семантику каждого оператора цикла в Си (с учетом операторов break и continue, которые, возможно, содержатся в теле цикла).

3.5. Может ли быть определено число итераций цикла for до начала его выполнения?

a) в Паскале
b) в Си

3.6. Верно ли решена задача: «найти сумму первых 100 натуральных чисел»?

a)
```
i = 1; sum = 0;
for ( ; i <= 100; i++) sum += i;
```

b)
```
sum = 0;
for ( i = 1; i <= 100;) sum += i++;
```

c)
```
for ( i = 1, sum = 0; i <= 100; sum += i+,  i++);
```

d) for (i = 1, sum = 0; i <= 100; sum += i++);

e) for (i = 0, sum = 0; i++, i <= 100; sum += i);

3.7. Выразить семантику цикла for с помощью цикла while. Эквивалентны ли полученные фрагменты программ?

3.8. Эквивалентны ли следующие фрагменты программы:

a) for (; e2 ;) S; и while (e2) S;
b) for (; ;) S; и while (1) S;

Замечание: здесь e2 - выражение допустимого в этом случае типа; S - произвольный оператор.

3.9. Можно ли написать фрагмент программы на Си, эквивалентный данному, используя один оператор цикла for с пустым оператором в качестве тела цикла?

```
i = 0; c = getchar();

while (c != ’  ’ && c != ’\n’ && c !=’ \t’ && c != EOF)

   { i++; c = getchar(); }
```

3.10. Сравнить семантику операторов repeat в Паскале и do-while в Си.

3.11. Улучшить стиль (структуру) следующих фрагментов программы на Си:

a) while (E1)

```
    { if ( E2 ) continue; S; }
```

b) do { if (E1) continue; else S1; S2; }

```
   while ( E2 );
```

Замечание: здесь E1, E2 - выражения допустимого в этом случае типа; S, S1, S2 - произвольные операторы.

3.12. Что напечатает следующая программа?

```
# include <stdio.h>
```

```
main()
  { int x, y, z;
    x = y = 0;
    while ( y < 10 ) ++y; x += y;
    printf ("x = %d y = %d\n", x, y);
    x = y = 0;
    while ( y < 10 ) x += ++ y;
    printf (" x= %d y = %d\n", x, y);
    y = 1;
    while ( y < 10 ) { x = y ++; z = ++y;}
    printf ("x = %d y = %d z = %d\n", x, y, z);
    for ( y =1; y < 10; y++ ) x = y;
    printf (" x= %d y = %d\n", x, y);
    for ( y = 1; ( x = y ) < 10; y++ );
    printf ("x = %d y = %d\n", x, y);
    for ( x = 0, y = 1000; y > 1; x++, y /= 10 )
    printf ("x = %d y = %d\n", x, y);
  }
```

3.13. Сравнить семантику операторов case в Паскале и switch в Си.

3.14. Верны ли следующие утверждения:

a) «любое выражение в Си может быть преобразовано в оператор добавлением к нему точки с запятой (;) »

b) «пустой оператор в Си - это отсутствие каких-либо символов в том месте конструкции, где по синтаксису может находиться оператор»

c) «составной оператор (блок) в Си - это совокупность операторов, заключенная в фигурные скобки { }»

d) «оператор присваивания в Си - это выражение вида

переменная = выражение»

e) «тип выражения в условии в операторе if , в условии завершения цикла в операторах цикла может быть скалярным (т.е. любым целочисленным, любым вещественным либо указателем)»

f) «в блоке описания/объявления и операторы могут располагаться в любом порядке; единственное требование - использование не должно предшествовать описанию/объявлению»

e) «цикл for (; ;) является бесконечным циклом, и поэтому его использование в Си запрещено»

f) «семантика операторов цикла while и do-while в Си различается только тем, что тело цикла while может не выполниться ни разу, а тело цикла do-while выполнится хотя бы один раз.»

g) «каждая ветвь в операторе switch должна быть помечена одной или несколькими различными целочисленными константами или константными выражениями»

h) «если в операторе switch нет ветви default, то значение выражения выбора должно совпадать с одной из констант ветвей case»

i) «в операторе switch ветви case и ветвь default можно располагать в любом порядке»

3.15. Верно ли утверждение: « действие оператора continue; в приведенных ниже примерах эквивалентно действию оператора go to next; ».

a) while (E) { S; ... continue; ... S; next: ; }
b) do { S; ... continue; ... S; } while (E); next: ; ...
c) for (E1; E2; E3) { S; ... continue; ... S; next: ; }
d) while (E) { S; ... for (E1;E2;E3) { S; ... continue; ... S; } ... S; next:; }
e) while (E) { S; ... for (E1;E2;E3) { S; ... continue; ... S; next: ; } ... S;}
f) switch (E) { case C1: S;
case C2: S; continue;

case C3: S; }

next:; ...

g) switch (E) { case C1: S;
case C2: S; continue;

case C3: next: S; }

h) next: switch (E) { case C1: S;
case C2: S; continue;

case C3: S; }

Замечание: здесь E, E1, E2, E3 - выражения допустимого в этом случае типа ; S - произвольный оператор; C1, C2 C3 - константы подходящего типа.

3.16. Верно ли утверждение: « действие оператора break; в приведенных ниже примерах эквивалентно действию оператора go to next; ».

a) while (E) { S; ... break; ... S; next: ; }

b) while (E) { S; ... break; ... S; } next: ; ...

c) do { S; ... break; ... S; } while (E); next: ; ...

d) for (E1; E2; E3) { S; ... break; ... S; next: ; }

e) while (E) { S; ... for (E1; E2; E3) { S; ... break; ... S; } next: ; ... S; }

f) while (E) { S; ... for (E1; E2; E3) { S; ... break; ... S; } ... S; next: ; }

g) while (E) { S; ... for (E1; E2; E3) { S; ... break; ... S; } ... S; } next: ;

h) switch (E) { case C1: S;

case C2: S; break;

case C3: S; }

next:; ...

i) switch (E) { case C1: S;

case C2: S; break; S;

case C3: next: S; }

Замечание: здесь E, E1, E2, E3 - выражения допустимого в этом случае типа; S - произвольный оператор; C1, C2 C3 - константы подходящего типа.

3.2 Обработка числовых данных

Замечание: при решении некоторых задач этого раздела необходимы минимальные знания о «стандартном» вводе и выводе целых и вещественных чисел.

3.17. Для данных чисел a, b и c определить, сколько корней имеет уравнение $ax^2+bx+c = 0$, и распечатать их. Если уравнение имеет комплексные корни, то распечатать их в виде $v \pm iw$.

3.18. Подсчитать количество натуральных чисел n ($111 \le n \le 999$), в записи которых есть две одинаковые цифры.

3.19. Подсчитать количество натуральных чисел n ($102 \le n \le 987$), в которых все три цифры различны.

3.20. Подсчитать количество натуральных чисел n ($11 \le n \le 999$), являющихся палиндромами, и распечатать их.

3.21. Подсчитать количество цифр в десятичной записи целого неотрицательного числа n.

3.22. Определить, верно ли, что куб суммы цифр натурального числа n равен n^2.

3.23. Определить, является ли натуральное число n степенью числа 3.

3.24. Для данного вещественного числа a среди чисел 1, 1 + (1/2), 1 + (1/2) + (1/3), ... найти первое, большее a.

3.25. Для данного вещественного положительного числа a найти наименьшее целое положительное n такое, что $1 + 1/2 + 1/3 + ... + 1/n > a$.

3.26. Даны натуральное число n и вещественное число x. Среди чисел $\exp(\cos(x^{2k}))\sin(x^{3k})$ (k = 1, 2, ..., n) найти ближайшее к какому-нибудь целому.

3.27. Дано натуральное число n. Найти значение числа, полученного следующим образом: из записи числа n выбросить цифры 0 и 5, оставив прежним порядок остальных цифр.

3.28. Дано натуральное число n. Получить все такие натуральные q, что n делится на q^2 и не делится на q^3.

3.29. Дано натуральное число n. Получить все его натуральные делители.

3.30. Дано целое число m > 1.Получить наибольшее целое k, при котором $4^k < m$.

3.31. Дано натуральное число n. Получить наименьшее число вида 2^r, превосходящее n.

3.32. Распечатать первые n простых чисел (p - простое число, если p >= 2 и делится только на 1 и на себя).

3.33. Даны вещественные числа x и y (x > 0, y > 1). Получить целое число k (положительное, отрицательное или равное нулю), удовлетворяющее условию $y^{k-1} \le x < y^k$.

3.34. Распечатать первые n чисел Фибоначчи ($f_0 = 1$; $f_1 = 1$; $f_{k+1} = f_{k-1} + f_k$; k = 1, 2, 3,...)

3.35. Вычислить с точностью eps > 0 значение «золотого сечения» - $0.5*(1+\sqrt{5})$ - предел последовательности $\{ q_i \}$ при $i \to \infty$

$q_i = f_i / f_{i-1}$, i = 2, 3,...где f_i - числа Фибоначчи (см. предыдущую задачу).

Считать, что требуемая точность достигнута, если $| q_i - q_{i+1}| < eps$.

3.36. Распечатать числа Фибоначчи (см. задачу 3.34), являющиеся простыми числами со значениями меньше n.

3.37. Вычислить с точностью eps > 0 значение числа **e** - предел последовательности $\{ x_i \}$ при $i \to \infty$

$x_i = (1+1/i)^i$, i = 1, 2, ...

Считать, что требуемая точность достигнута, если $| x_i - x_{i+1}| < eps$.

3.38. Вычислить значение $\sum i!$ для i, изменяющихся от 1 до n. Воспользоваться соотношением $\sum i! = 1 + 1*2 + 1*2*3 +...+ 1*2*3*...*n = 1+2*(1+3*(1+ \ +n*(1)...))$.

3.39. Пусть a_0 и b_0 - положительные вещественные числа. Соотношениями $a_{n+1} = \sqrt{(a_n b_n)}$; $b_{n+1} = (a_n+b_n) / 2$ при n = 0, 1, 2, ... задаются две бесконечные числовые последовательности $\{a_n\}$и $\{b_n\}$, которые сходятся к общему пределу $M(a_0,b_0)$, называемому арифметико-геометрическим средним чисел a_0 и b_0. Найти приближенное значение $M(a_0,b_0)$ с точностью eps > 0. Поскольку при $a_0 < b_0$ $a_i < b_i$ и, более того, $a_0 < a_1 < ... < a_i < ... bi < ... < b_1 < b_0$, то в качестве подходящего критерия прекращения вычислений можно использовать соотношение $| a_i - b_i | <$ eps.

3.40. Вычислить квадратные корни вещественных чисел x = 2.0, 3.0, ... , 100.0. Распечатать значения x, √x, количество итераций, необходимых для вычисления корня с точностью eps > 0.

Для a > 0 величина √a вычисляется следующим образом:

$$a_0 = 1; a_{i+1} = 0.5*(a_i+a/a_i) \; i = 0, 1, 2,...$$

Считать, что требуемая точность достигнута, если $| a_i-a_{i+1}| <$ eps.

3.41. Найти приближенное значение числа π с точностью eps > 0. Для этого можно использовать представление числа 2/π в виде произведения корней √(1/2) *√(1/2+1/2√(1/2))*√(1/2+ 1/2√(1/2+1/2√(1/2)))*... . Вычисления прекращаются, когда два следующих друг за другом приближения для числа π будут отличаться меньше, чем на eps.

3.42. Для данного вещественного числа x и натурального n вычислить:

a) $\sin x + \sin^2 x + ... + \sin^n x$

b) $\sin x + \sin x^2 + ... + \sin x^n$

c) sin x + sin(sin x) + ... + sin (sin (... sin(sin x) ...))

3.43. Алгоритм Евклида нахождения наибольшего общего делителя (НОД) неотрицательных целых чисел основан на следующих свойствах этой величины: пусть m и n - одновременно не равные нулю целые неотрицательные числа и m ≥ n. Тогда, если n = 0, то НОД(n, m) = m, а если n ≠ 0, то для чисел m, n, и r, где r - остаток от деления m на n, выполняется равенство НОД(m, n) = НОД(n, r). Используя алгоритм Евклида, определить наибольший общий делитель неотрицательных целых чисел a и b.

3.44. Вычислить 1 - 1/2 + 1/3 - 1/4 + ...+1/9999 - 1/10000 следующими способами:

a). последовательно слева направо;

b). последовательно справа налево;

c). последовательно слева направо вычисляются 1 +1/3 + 1/5 + ... + 1/9999 и 1/2 + 1/4 + ... + 1/10000, затем второе значение вычитается из первого;

d). последовательно справа налево вычисляются 1 +1/3 + 1/5 + ... + 1/9999 и 1/2 + 1/4 + ... + 1/10000, затем второе значение вычитается из первого.

Сравнить и объяснить полученные результаты.

3.45. Натуральное число называется совершенным, если оно равно сумме всех своих делителей, за исключением самого себя. Дано натуральное число n. Получить все совершенные числа, меньшие n.

3.46. Определить, является ли число простых чисел, меньших 10000, простым числом.

3.47. Если p и q - простые числа и q = p+2, то они называются простыми сдвоенными числами или “близнецами” (twin primes). Например, 3 и 5 - такие простые числа. Распечатать все простые сдвоенные числа, меньшие N.

3.3 Обработка символьных данных

Замечание: при решении некоторых задач этого раздела необходимы минимальные знания о «стандартном» вводе и выводе литер.

3.48. Пусть во входном потоке находится последовательность литер, заканчивающаяся точкой (кодировка ASCII):

a) определить, сколько раз в этой последовательности встречается символ ‘a’;

b) определить, сколько символов ‘e’ предшествует первому вхождению символа ‘u’ (либо сколько всего символов ‘e’ в этой последовательности, если она не содержит символа ‘u’);

c) выяснить, есть ли в данной последовательности хотя бы одна пара символов-соседей ‘n’ и ‘o’, т.е. образующих сочетание ‘n’ ‘o’ либо ‘o’ ‘n’;

d) выяснить, чередуются ли в данной последовательности символы ‘+’ и ‘-‘, и сколько раз каждый из этих символов входит в эту последовательность;

e) выяснить, сколько раз в данную последовательность входит группа подряд идущих символов, образующих слово *C++*;

f) выяснить, есть ли среди символов этой последовательности символы, образующие слово char;

g) выяснить, есть ли в данной последовательности фрагмент из подряд идущих литер, образующий начало латинского алфавита (строчные буквы), и какова его длина. Если таких фрагментов несколько, найти длину наибольшего из них. Если такого фрагмента нет, то считать длину равной нулю;

h) выяснить, есть ли в данной последовательности фрагменты из подряд идущих цифр, изображающие целые числа без знака. Найти значение наибольшего из этих чисел. Если в этой последовательности нет ни одной цифры, то считать, что это значение равно нулю;

i) определить, имеет ли данная последовательность символов структуру, которая может быть описана с помощью следующих правил:

последовательность ::= слагаемое + последовательность | слагаемое

слагаемое ::= идентификатор | целое

идентификатор ::= буква | идентификатор буква | идентификатор цифра

буква ::= A | B | C | D | E | F | G | H | I | J | K

цифра ::= 0 | 1 | 2 | 3 | 4 | 5

целое ::= цифра | целое цифра

3.49. Пусть во входном потоке находится последовательность литер, заканчивающаяся точкой (кодировка ASCII). Вывести в выходной поток последовательность литер, измененную следующим образом:

a) заменить все символы ‘?’ на’!’;

b) удалить все символы ‘-‘ и удвоить все символы ‘&’;

c) удалить все символы, не являющиеся строчными латинскими буквами;

d) заменить все прописные латинские буквы строчными (другие символы копировать в выходной поток без изменения);

e) заменить все строчные латинские буквы прописными (другие символы копировать в выходной поток без изменения);

f) каждую группу рядом стоящих символов ‘+’ заменить одним таким символом;

g) каждую группу из n рядом стоящих символов ‘*’ заменить группой из n/2 рядом стоящих символов ‘+’ (n >= 2); одиночные ‘*’ копировать в выходной поток без изменения;

h) удалить из каждой группы подряд идущих цифр все начальные незначащие нули (если группа состоит только из нулей, то заменить эту группу одним нулем);

i) удалить все комбинации символов the;

j) оставить только те группы цифр, которые составлены из подряд идущих цифр с возрастающими значениями; все остальные цифры и группы цифр удалить (другие символы копировать в выходной поток без изменения);

k) заменить все комбинации символов child комбинациями символов children;

l) удалить группы символов, расположенные между фигурными скобками { и }. Скобки тоже должны быть удалены. Предполагается, что скобки сбалансированы, и внутри каждой пары скобок других фигурных скобок нет.

3.50. Пусть во входном потоке находится последовательность литер, заканчивающаяся маркером конца $ (кодировка ASCII). Вывести в выходной поток последовательность литер, измененную следующим образом:

a) удалить из каждой группы подряд идущих цифр, в которой более двух цифр и которой предшествует точка, все цифры, начиная с третьей (например, a+12.3456-b-0.456789+1.3-45678 преобразуется в a+12.34-b-0.45+1.3-45678);

b) удалить из каждой группы цифр, которой не предшествует точка, все начальные нули (кроме последнего, если за ним идет точка либо в этой группе нет других цифр, кроме нулей ; например, a-000123+bc+0000.0008-0000+0001.07 преобразуется в a-123+bc+0.0008-0+1.07).

3. ФУНКЦИИ И СТРУКТУРА ПРОГРАММЫ

4.1. Перечислите все существенные изменения, внесенные стандартом ANSI в правила объявления и описания функций. Какова цель этих изменений?

4.2. Перечислить все случаи, когда в Си используется тип void. Дать определение этого типа.

4.3. Перечислить классы памяти, определенные в Си. Что определяет класс памяти? В каких случаях и каким образом класс памяти определяется по умолчанию? Привести примеры.

4.4. Определен ли в Си класс памяти для функций? Если определен, то каким образом; если нет, то почему.

4.5. Допустима ли в Си вложенность функций? Можно ли в Си каким-то образом управлять видимостью функций?

4.6. Объяснить, чем различаются описание (объявление, declaration) и определение (definition) – по терминологии Б. Кернигана и Д. Ритчи [1, см. стр.71]. Привести примеры.

4.7. Эквивалентны ли следующие объявления функций:

a) double f ( );	и	double f (void);
b) char g (int i, char c);	и	char g (int, char);
c) h (double x);	и	int h (double x);
d) void h (int);	и	h (int);
e) extern int q (int);	и	int q (int);
f) static void s (char c);	и	void s (char c);

4.8. Определить, какие конструкции являются определениями, а какие описаниями; где они могут располагаться и что обозначают:

int i;	char c = ‘a’;	extern int f (int, char);
static int j;	register int b;	double g( ) { return 3.141592; };
extern long k;	int h (int i);	static char q(int, double);
auto short n;	s();	static void p(int i) { };

4.9. Верны ли следующие утверждения:

a) «тип выражения в операторе return должен совпадать с типом результата функции»

b) «функция, которая не возвращает результата (тип результата void), может не содержать оператор return;»

c) «функция, которая возвращает результат, может не содержать оператор return E; но вызывает другую функцию, которая содержит такой оператор»

d) «функция, которая возвращает результат, может содержать несколько операторов return E; »

e) «в Си аргументы функции всегда передаются по значению»

f) «в теле одной функции могут находиться два разных оператора, помеченных одинаковыми метками, если эти операторы находятся в разных блоках»; например,

```
void f(void)
  { ... label: S1; ...
      { ... label: S2; ... goto label; ...}
    ... goto label; ...
  }
```

g) «в Си нельзя использовать две (или более) взаимно рекурсивных функций»; например, если есть void f(void){... g();...} и void g(void){...f();...}, то программа, использующая такие функции, будет ошибочной.

h) «в Си можно войти в блок, минуя его заголовок; при этом память под локальные переменные, описанные в этом блоке, будет отведена, но инициализация (если она есть) выполняться не будет»

i) «любая функция, описанная в каком-либо файле, входящем в состав программы, может быть использована в этом и любом другом файле этой программы»

j) «в этом фрагменте программы нет ошибок »

```
#include <stdio.h>
int f(void) { return 100;}
void g(void) { printf("O.K.\n");}
main()
   { int i, j;
```

```
    i = f();
    j = g(), f();
    g(); f();
    printf("i=%d, j=%d f=%d\n", i, j, f());
}
```

4.10. В каких случаях в Си возможна инициализация переменных? Когда она происходит? Как определяется инициализация по умолчанию?

4.11. Допустимо ли в Си? Если "да" - опишите семантику этих действий; если "нет" - объясните почему.

a)
```
#include <stdio.h>
main()
  { int i; int sum = 0;
    for ( i = 1; i <= 3; i++)
      { sum += i;
        { int i;
          for ( i = 1; i <= 5; i++)
            sum *= i;
        }
      }
    printf("sum=%d\n", sum);
  }
```

b)
```
#include <stdio.h>
main()
  { int i; int sum = 0;
    for ( i = 1; i <= 3; i++)
      { sum += i;
        { for ( i = 1; i <= 5; i++)
            sum *= i;
        }
      }
    printf("sum=%d\n", sum);
  }
```

c)
```
#include <stdio.h>
main()
  { double x;
    scanf("%f", &x);
    if ( x >= 0 ) goto ok;
      { double y = 5.0;
```

d)
```
#include <stdio.h>
main()
  { double x;
    scanf("%f", &x);
    if ( x >= 0 ) goto ok;
      { double y;
```

```
        x = x ? x : -x;
        ok: y+=sqrt(x);
        printf("y = %f\n",y);}
       }
   }
```

```
        x = x ? x : -x;
        ok: y = sqrt(x);
        printf("y = %f\n",y);
      }
  }
```

4.12. Допустимо ли в Си? Если "да" - опишите семантику этих действий; если "нет" - объясните почему.

a) файл **f1:**

```
#include<stdio.h>
 main()
   { extern int f(int);
     int i;
     i = f(g(5));
     printf("i = %d\n", i);
   }
 int g(int k)
   { k++;
    return f(k);
   }
```

файл **f2:**

```
 int f(int i) { return i*i; }
```

b) файл **f1:**

```
#include<stdio.h>
 static int f(void)
   { int k;
    scanf("%d", &k); return k; }
 extern int g(int);
 main()
   { int i;
     i = f(); j = g(i);
     printf("i=%d,j=%d\n", i,j);
    }
```

файл **f2:**

```
 extern int f(void);
 int g(int i)
   { return  f()+i; }
```

c) файл **f1:**

```
#include<stdio.h>
 extern int i; extern void f(void);
 main()
   { f();
```

d) файл **f1:**

```
#include<stdio.h>
 extern int f(void);
 int i;
 main()
```

```
    printf("i = %d\n", i);
    f();
    printf("i = %d\n", i);
  }
```

файл **f2:**

```
int i = 1;
void f(void) { i++; }
```

```
{ printf("res = %d\n", i+f()+f());
 }
```

файл **f2:**

```
int f(void)
{ static int i = 10;
  return i--;
 }
```

4.13. Что напечатает следующая программа?

a).
```
#include <stdio.h>
  int i = 0; void f(void);
  main()
    { int i =1;
      printf("i1 = %d\n", i);
      f();
        { int i = 2; printf("i4 = %d\n", i);
            { i +=1; printf("i5 = %d\n", i); }
          printf("i6 = %d\n", i);
        }
      printf("i7 = %d\n", i);
    }
  void f(void)
    { printf("i2 = %d\n", i);
     i = i + 10; printf("i3 = %d\n", i); }
```

b).
```
#include <stdio.h>
 int f( int ); int b = 10;
 main()
   { int c = 3;
     b = f(c);
     printf("c=%d b=%d\n",c,b);
    }
 int f( int b )
   { int c = 5;
     c--; b++;
     printf("c=%d b=%d\n",c,b);
     return c+b;
   }
```

c).
```
#include <stdio.h>
```

d).
```
#include <stdio.h>
```

```
char g ( char c);
int f ( int i, char c)
  { int k = i+4; char b = g(c) + i;
    printf("c1 = %c k0 = %d\n", c, k);
      { int i = 3; k += i;
        printf("i1 = %d k1 = %d\n", i, k);
        c = g('b'); printf("c2 = %c\n", c);
      }       }
    i++; printf("i2 = %d k2 = %d\n", i, k);
    return i*(b-c); }
char g(char c)
   { c = c + 1; return c; }
main()
    { int k =2; char c = 'a'; k = f (k, c);
      printf("c3 = %c k3 = %d\n", c, k);
    }
```

```
int abc( int ); int x;
 main()
  { int b;
    b = abc(x);
    printf("b1=%d x1=%d\n",b,x);
    x = abc(b);
    printf("b2=%d x2=%d\n",b,x);

int abc( int b )
  { static int x = 5;
    x += 7; b++;
    printf("b0=%d x0=%d\n",b,x);
    return x+b;
  }
```

e). #include <stdio.h>

```
int i = 1;
int reset ( void );
int next ( int );
int last ( int );
int new ( int );
main()
  { int i, j; i = reset();
    for ( j = 1; j <= 3; j++ )
      { printf("i=%d j=%d\n", i, j);
        printf("%d %d\n", next(i), last(i));
        printf("%d\n", new(i+j));
```

f). #include <stdio.h>

```
int i = 1;
int reset ( void );
int next ( void );
int last ( void );
int new ( int );
main()
  { int i, j; i = reset();
    for ( j = 1; j <= 3; j++ )
      { printf("i=%d j=%d\n", i, j);
        printf("%d\n", next(),);
        printf("%d\n", last());
```

```
    } }
int reset ( void ) { return i; }
   int next ( int j ) { return j = i++; }
   int last ( int j )
      { static int i = 10; return j = i--; }
   int new ( int i )
      { int j = 10; return i = j += i; }
```

```
        printf("%d\n", new(i+j)); } }
```

в файле **f1**:

```
static int i =10;
int next ( void ) { return i += 1; }
int last ( void ) { return i -= 1; }
int new ( int i)
  { static int j = 5; return i=j+=i; }
```

в файле **f2**:

```
extern int i;
int reset ( void ) { return i; }
```

4.14. Программа. Описать рекурсивную функцию вычисления n! - факториала числа n, основанную на соотношении n! = n∗(n-1)!. С ее помощью найти факториалы натуральных чисел от1 до 10.

4.15. Программа. Описать рекурсивную функцию вычисления x^n для вещественного x ($x \neq 0$) и целого n:

$$x^n = \begin{cases} 1 & \text{при } n = 0 \\ 1/x^{|n|} & \text{при } n < 0 \\ x * x^{n-1} & \text{при } n > 0 \end{cases}$$

Протестировать эту функцию на подходящих наборах входных данных.

4.16. Программа. Описать рекурсивную функцию вычисления НОД(n,m) - наибольшего общего делителя неотрицательных целых чисел n и m, основанную на соотношении НОД(n,m) = НОД(m,r), где r - остаток от деления n на m (см. задачу 3.43). С ее помощью найти наибольший общий делитель натуральных чисел a и b. Сравнить эффективность рекурсивной и нерекурсивной функций вычисления НОД.

4.17. Программа. Описать рекурсивную функцию вычисления НОД (n_1, n_2, n_3, ..., n_m), воспользовавшись для этого соотношением: НОД (n_1, n_2, n_3, ... , n_k) = НОД (НОД (n_1, n_2, n_3, ... , n_{k-1}), n_k), k = 3, 4, ... , m. С ее помощью найти НОД (a_1, a_2, a_3, ..., a_{10}).

4.18. Программа. Описать рекурсивную функцию вычисления n–ого числа Фибоначчи: $f_0 = 1$; $f_1 = 1$; $f_{j+1} = f_{j-1} + f_j$; j = 1, 2, 3,.…С ее помощью вычислить 100-ое число Фибоначчи.

4.19. Программа. Описать рекурсивную функцию вычисления значения A(n,m) - функции Аккермана для неотрицательных целых чисел n и m:

$$A(n,m) = \begin{cases} m+1 & \text{если } n = 0 \\ A(n-1,1) & \text{если } n \neq 0,\ m = 0 \\ A(n-1, A(n, m-1)) & \text{если } n>0,\ m>0 \end{cases}$$

С помощью этой функции найти значение A(5,8).

4. УКАЗАТЕЛИ И МАССИВЫ

5.1. Допустимо ли в Си? Если "да" - опишите семантику каждого правильного действия (не принимая во внимание ошибочные); если "нет" - объясните почему.

a). . . .

```
int i,* p, j, *q;
p = &i;          q = &p;
j = *p = 1;              q = p-1;          *p += 1;
i = *++q + *p; q -= 1;          i = *q ++ + *q;
printf("i=%d, j=%d, *p=%d, *q=%d \n", i, j, *p, *q);
```

b) . . .

```
int x = 1, y; char c = ‘a’;
int *pi, *qi; char *pc;
pi = &x;        *pi = 3;        y = *pi;        *pi = c;        qi = pi;
pc = qi;        *qi+=1;         pi++;           *(- - pi) = 5;  y = *qi+1;
pc = &c;        ++*pc;          (*pc)++;        *pc++; *pc+=1;
```

```
x = (int)pi;     pi=(int*)pc;    pi=(int*)x;     x = 1+ *pi;     pc=(char*)pi;
c = *pc;         pc = &y;        x = qi – pi;    qi = 0;         qi+=pi;
y = &pi;         y = (int)&pi;   pi = pi +5;     *(pi+1)=0;      pi=&(x+0);
```

5.2. К любому ли объекту в Си можно применять операцию взятия адреса & ?

5. 3. Допустимо ли в Си? Если "да" - опишите семантику этих действий; если "нет" - объясните почему.

a)
```
int i = 2; const int j = 5;
int *pi;
const int *pci;
int *const cpi;
const int * const cpci;
pi = &i;        pci = &j;       cpi = &i;       cpci = &j;      pci = &i;
pi = (int*)&j;                  i = *pci + *pi;                 *pci = 3;
*pi = 3;        i=*pci++;       *(cpi++)=5;     *cpi++;
```

b)
```
int f(const int i, int j) { j++; return i+j; }
main()
  { int a, b; const int c = 5;
    scanf("%d", &a);
    b = f(c,a); printf("a=%d, b=%d, c=%d \n", a, b, c);
    b = f(c,c); printf("a=%d, b=%d, c=%d \n", a, b, c);
    b = f(a,a); printf("a=%d, b=%d, c=%d \n", a, b, c);
    b = f(a,c); printf("a=%d, b=%d, c=%d \n", a, b, c);
  }
```

5.4. Пусть целочисленный массив a соддержит 100 элементов. Верно ли решена задача: "написать фрагмент программы, выполняющий суммирование всех элементов массива a".

a)
```
int a[100], sum, i;
```

```
    sum = 0;
    for ( i = 0; i < 100; ++i ) sum += a[i];
```

b)
```
int a[100], *p, sum;
    sum = 0;
    for ( p = a; p < &a[100]; ++p ) sum = sum + *p;
```

c)
```
int a[100], *p, sum;
    sum = 0;
    for ( p = &a[0]; p < &a[100]; p++ ) sum += *p;
```

d)
```
int a[100], sum, i;
    sum = 0;
    for ( i = 0; i < 100; ++i ) sum += *(a+i);
```

e)
```
int a[100], sum, i;
    sum = 0;
    for ( i = 0; i < 100; ++a, ++i ) sum += *a;
```

f)
```
int a[100], *p, sum, i;
    sum = 0;
    for ( i = 0, p = a; i < 100; ++i ) sum += p[i];
```

g)
```
int a[100], *p, sum, i;
    sum = 0;
    for ( i = 0, p = a; i < 100; ++i ) sum += *(p+i);
```

5.5. Допустимо ли в Си? Если "да" - опишите семантику каждого правильного действия (не принимая во внимание ошибочные); если "нет" - объясните почему.

a) . . .
```
  int  a[5] = { 1, 2, 3, 4, 5 };
  int  *p, x, *q , i;
  p = a + 2;        * (p+2) = 7;
  *a += 3;          q=*&p-1;
```

```
x = ++ p - q ++;        x += ++ *p;    x=*p-- + *p++;
for (i = 0; i < 5; i++) printf("a [%d]=%d", i, a[ i ] ); printf("\n");
printf("x=%d, *p=%d, *q=%d \n", x, *p, *q);
```

b) . . .

```
char *str = "abcdef";
char *p, *q, *r; int k;
p = str; q = 0;            p++;
k = p - str;      r = p+k;
if ( k && p || q ) q = str + 6;
p = q ? r : q;             *(p-1) = 'a';   *r = 'x';
printf("str: %s\n", str);
```

c) . . .

```
char s[ ] = "0123456";
int *pi; char *pc1, *pc2;
pc2 = s;
pc1 = s + *(s+strlen(s) - 1) - '0';
pi = ( int* ) pc2;        *pc1-- = '8';
if ( pc1 - pc2 < 3 ) pc1 = pc2 = pi; else pc1 = ( pc1+pc2 )/2;
if ( s == pc2 ) *pc1 = *pc2 + 1; else *pc1 = '9';
printf("s: %s\n", s);
```

d) . . .

```
int i; char *c; int *pi;
i = 'a';
pi = &i;          c = (char*)pi + 3;      printf("c1=%c", *c);
i <<= 8;          c--;                    printf("c2=%c\n", *c);
```

e) . . .

```
char c1, c2; short i;
```

```
char *pc; short *ps;
c1 = '1';        c2 = '2';        ps =&i;
pc = (char*)ps;        *pc = c1;        pc++;        *pc = c2;
printf("i = %hd\n", i);
```

5.6. Эквивалентны ли следующие фрагменты программы на Си?

a[i] /= k+m	и	a[i] = a[i]/k+m
a[i] /= k+m	и	a[i] = a[i]/(k+m)
a[i++]+=3	и	a[i++] = a[i++]+3
a[i++]+=3	и	a[i] = a[i++]+3
a[i++]+=3	и	a[i++] = a[i]+3
a[i++]+=3	и	a[i] = a[i]+3; i++;

5.7. Что напечатает следующая программа?

```
#include <stdio.h>
char str[ ] = "SSSWILTECH1\1\11W\1WALLMP1";
main()
  { int i, c;
    for ( i = 2; ( c = str [ i ] ) != '\0'; i++) {
          switch (c) {
            case 'a': putchar('i'); continue;
            case '1': break;
            case 1: while ( ( c = str [++ i ] ) != '\1' && c != '\0');
            case 9: putchar('S');
            case 'E': case 'L': continue;
            default: putchar(c); continue; }
          putchar(' '); }
    putchar('\n');
```

```
    }
```

5.8. Что напечатает следующая программа?

```
#include <stdio.h>
int a[ ] = { 0, 1, 2, 3, 4 };
main()
  { int i, *p;
    for ( i = 0; i <= 4; i++ ) printf("a[ i ]=%d  ", a[ i ]);  printf("\n");
    for ( p = &a[0]; p <= &a[4]; p++ ) printf("*p=%d  ", *p);  printf("\n");
   for ( p = &a[0], i = 0 ; i <= 4; i++ ) printf("p[ i ]=%d  ", p[ i ]);  printf("\n");
    for ( p = a, i = 0; p+i <= a+4; i++ ) printf("* (p+i)=%d ", * (p+i));
    printf("\n");
    for ( p = a+4; p >= a; p-- ) printf("*p=%d  ", *p );  printf("\n");
    for ( p = a+4, i=0; i <= 4; i++ ) printf("p[ -i ]=%d  ", p[ -i ]);
    printf("\n");
    for ( p = a+4; p >= a; p -- ) printf("a[ p - a ]=%d  ", a[ p - a ]);
    printf("\n");
  }
```

5.9. Что напечатает следующая программа?

```
#include <stdio.h>
int a[ ] = { 8, 7, 6, 5, 4 };
int  *p[ ] = { a, a+1, a+2, a+3, a+4 };
int  **pp = p;
main()
   { printf("*a=%d  **p=%d  **pp=%d\n", *a, **p, **pp );
     pp++;
     printf("pp-p=%d  *pp-a=%d  **pp=%d\n", pp-p, *pp-a, **pp );
     ++*pp;
```

```
    printf("pp-p=%d  *pp-a=%d  **pp=%d\n", pp-p, *pp-a, **pp );
    pp = p;
    ++**pp;
    printf("pp-p=%d  *pp-a=%d  **pp=%d\n", pp-p, *pp-a, **pp );
  }
```

5.10. Что напечатает следующая программа?

```
#include <stdio.h>
int a[ 3 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
int  *pa[ 3 ] = { a[ 0 ], a[ 1 ], a[ 2 ] };
int  *p = a[ 0 ];
main()
     { int i;
       for ( i = 0; i < 3; i ++ )
            printf(" a[ i ][ 2 – i ]=%d  *a[ i ]=%d  *(*(a+i)+i)=%d\n",
                 a[ i ][ 2 – i ], *a[ i ], *(*(a+i)+i));
       for ( i = 0; i < 3; i ++ )
            printf("*pa[ i ]=%d  p[ i ]=%d \n", *pa[ i ], p[ i ] );
     }
```

5.11. Что напечатает следующая программа?

```
#include <stdio.h>
char *c[ ] = { "ENTER", "NEW", "POINT", "FIRST" };
char ** cp[ ] = { c+3, c+2, c+1, c };
char ***cpp=cp;
main()
     { printf("%s", **++cpp );
```

```
    printf("%s  ", * -- *++cpp+3 );
    printf("%s", *cpp[ -2 ]+3 );
    printf("%s\n", cpp[ -1 ][ -1 ]+1 );
}
```

5.12. Какие соглашения о конце строки существуют в Си и Паскале? Укажите все «за» и «против» явного указания концов строк с помощью null-литеры ‘\0’.

5.13. В чем заключается проблема «висящей» ссылки? Приведите примеры.

5.14. Нужна ли в Си «сборка мусора»? Почему возникает такая проблема и как она решается в Си?

5.15. Прочитайте следующие описания и определения:

```
int *ip, f( ), *fip( ), (*pfi)( );          char *str[10];      char * (*cp)[5];
int (*r) ( );                               double (*k)(double,int*);
float * (* (*x) [6] )( );                   double (* (* ( y( ) )[ ] )( );
int * (*const *name[9])(void);              char * const p;
```

5.16. Определите переменную x как массив указателей на функцию, имеющую два параметра типа int и возвращающую результат типа указатель на double.

5.17. Определите переменную y как указатель на массив указателей на функцию без параметров, возвращающую результат типа указатель на функцию с одним параметром типа int и результатом типа float.

5.18. Что будет напечатано? Объяснить, почему результат будет таким.

a)
```
#include <stdio.h>
int  try_to_change_it(int);
 main()
   { int i = 4, j;
     j = try_to_change_it(i);
```

b)
```
#include <stdio.h>
void compare (int , int *);
main()
 { int i = 4, j = 5;
   compare(i, &j);
```

```
    printf("i=%d, j=%d\n", i, j);
  }
int  try_to_change_it(int k)
  { printf("k1=%d\n", k);
    k+=33;
    printf("k2=%d\n", k);
    return k;
  }
```

```
    printf("i=%d, j=%d\n", i, j);
  }
void compare (int k, int *m)
  { printf("k1=%d,*m1=%d\n",k, *m);
      k++; (*m)++;
    printf("k2=%d,*m2=%d\n", k, *m);
   }
```

5.19. Верно ли решена задача: « Описать функцию, меняющую местами значения двух переменных символьного типа. Использовать эту функцию для изменения значений символьных переменных a и b.»

a)
```
void swap ( char x, char y)
   { char t; t = x; x = y; y = t;}
main()
   { char a,b;
     scanf("%c%c", &a, &b);
     swap(a,b);
     printf("a=%c,b=%c\n",a,b);
   }
```

b)
```
void swap ( char *x, char *y)
   { char *t; t = x; x = y; y = t;}
main()
   { char a,b;
     scanf("%c%c", &a, &b);
     swap(&a, &b);
     printf("a=%c,b=%c\n",a,b);
   }
```

c)
```
void swap ( char *x, char *y)
   { char t; t = *x; *x = *y; *y = t;}
main()
   { char a,b;
     scanf("%c%c", &a, &b);
     swap(a,b);
     printf("a=%c,b=%c\n",a,b);
   }
```

d)
```
void swap ( char *x, char *y)
   { char t; t = *x; *x = *y; *y = t;}
main()
   { char a,b;
     scanf("%c%c", &a, &b);
     swap(&a, &b);
     printf("a=%c,b=%c\n",a,b);
   }
```

e)
```
void swap ( char x, char y)
```

f)
```
void swap ( char &x, char &y)
```

```
{ char *t; t = &x; &x = &y; &y = t;}
main()
{ char a,b;
  scanf("%c%c", &a, &b);
  swap(&a, &b);
  printf("a=%c,b=%c\n",a,b);
}
```

```
{ char t; t = x; x = y; y = t;}
main()
{ char a,b;
  scanf("%c%c", &a, &b);
  swap(a, b);
  printf("a=%c,b=%c\n",a,b);
}
```

5.20. Допустимо ли в Си? Если "да" - опишите семантику этих действий; если "нет" - объясните почему.

```
int ques ( char *s1, char *s2)
{ while (*s1 && *s2 && *s1++ == *s2++ );
 return *--s1 - *--s2;
 }
```

5.21. Допустимо ли в Си? Если "да" - опишите семантику этих действий; если "нет" - объясните почему.

```
void ques ( char *s1, char *s2, int n)
 { while (*s1 && *s2 && n-- && (*s1 ++ = *s2 ++ ) ); }
```

5.22. Описать функцию, определяющую упорядочены ли строго по возрастанию элементы целочисленного массива из n элементов.

5.23. Описать функцию, определяющую индекс первого элемента целочисленного массива из n элементов, значение которого равно заданному числу x. Если такого элемента в массиве нет, то считать номер равным –1.

5.24. Описать функцию, вычисляющую значение $x_0 + x_0*x_1 + x_0*x_1*x_2 + \ldots + x_0*x_1*x_2 *\ldots *x_m$, где x_i - элементы вещественного массива x из n элементов, m - индекс первого отрицательного элемента этого массива либо число n-1, если такого элемента в массиве нет.

5.25. Описать функцию, вычисляющую значение max($x_0 + x_{n-1}$, $x_1 + x_{n-2}$, $x_2 + x_{n-3}$,…, $x_{(n-1)/2} + x_{n/2}$), где x_i - элементы вещественного массива x из n элементов.

5.26. Описать функцию, вычисляющую значение min($x_0 * x_1$, $x_1 * x_2$, $x_2 * x_3$,…, $x_{n-3} * x_{n-2}$, $x_{n-2} * x_{n-1}$), где x_i - элементы вещественного массива x из n элементов.

5.27. Описать функцию, вычисляющую значение $x_0*y_0+x_1*y_1+ \ldots + x_k*y_k$, где x_i – отрицательные элементы вещественного массива a из n элементов, взятые в порядке их следования; y_i – положительные элементы этого массива, взятые в обратном порядке; k = min(p,q), где p – количество положительных элементов массива a, q – количество отрицательных элементов этого массива.

5.28. Описать функцию, которая упорядочивает элементы целочисленного массива по неубыванию, используя следующий алгоритм сортировки:

a) сортировка выбором: находится максимальный элемент массива и переносится в его конец; затем этот метод применяется ко всем элементам массива, кроме последнего (т.к. он уже находится на своем месте), и т.д.

b) сортировка обменом (метод пузырька): последовательно сравни-ваются пары соседних элементов x_k и x_{k+1} (k = 0, 1, … ,n-2) и, если $x_k > x_{k+1}$, то они переставляются; в результате наибольший элемент окажется на своем месте в конце массива; затем этот метод применяется ко всем элементам, кроме последнего, и т.д.

c) сортировка вставками: пусть первые k элементов массива (от 0 до k-1) уже упорядочены по неубыванию; тогда берется x_k и размещается среди первых k элементов так, чтобы упорядоченными оказались уже k+1 первых элементов; этот метод повторяется при k от 1 до n-1.

5.29. Описать функцию, определяющую индекс первого элемента целочисленного массива из n элементов, значение которого равно заданному числу x. Если такого элемента в массиве нет, то считать номер равным –1. Элементы массива упорядочены по возрастанию; использовать метод двоичного (бинарного) поиска.

5.30. Программа. Описать функцию f(a, n, p), определяющую, чередуются ли положительные и отрицательные элементы в целочисленном массиве a из n элементов и вычисляющую целочисленное значение p. Если элементы чередуются, то p - это сумма

положительных элементов, иначе p - это произведение отрицательных элементов. С помощью этой функции провести анализ целочисленного массива x [50].

5.31. Программа. Описать функцию f(a, n, p), определяющую, упорядочены ли строго по возрастанию элементы в целочисленном массиве a из n элементов, и вычисляющую целочисленное значение p. Если элементы упорядочены, то p - это произведение разностей рядом стоящих элементов, иначе p - это количество нарушений порядка в массиве a. С помощью этой функции провести анализ целочисленного массива b [60].

5.32. Программа. Описать функцию f (s, n, x), определяющую, какой символ чаще других встречается в строке s и сколько раз он в нее входит. Если таких символов несколько, то взять первый из них по алфавиту. С помощью этой функции провести анализ строки str.

5.33. Программа. Описать функцию f(s, n, x), определяющую, какой символ реже других (но не нуль раз) встречается в строке s и сколько раз он в нее входит. Если таких символов несколько, то взять первый из них по алфавиту. С помощью этой функции провести анализ строки str.

5.34. Программа. Для целочисленного массива a, содержащего n элементов, описать функцию f(a, n, last, k, nlast), определяющую last - значение последнего из элементов массива a, значение которого принадлежит диапазону [-k, k], и nlast - индекс этого элемента. С помощью этой функции вычислить соответствующие значения last и nlast для целочисленных массивов x[20] и y[30].

5.35. Программа. Для вещественного массива a, содержащего n элементов, описать функцию G, определяющую значения максимального и минимального элементов этого массива. С помощью этой функции для вещественных массивов x[25] и y[40] вычислить соответствующие значения.

5.36. Описать функцию, которая изменяет заданную строку следующим образом: сначала записывает все элементы с четными индексами, а затем все элементы с нечетными индексами (с сохранением их относительного порядка в каждой группе).

Например, abcdefgh => acegbdfh, vwxyz => vxzwy.

5.37. Описать функцию, которая в заданной строке меняет местами ее первую и вторую половины.

Например, abcdefgh => efghabcd, vwxyz => yzxvw.

5.38. Описать функцию, осуществляющую циклический сдвиг на n позиций вправо элементов целочисленного массива, содержащего m элементов (n<m).

5.39. Описать функцию, осуществляющую циклический сдвиг на n позиций влево элементов целочисленного массива, содержащего m элементов (n<m).

5.40. Написать программу, обнуляющую каждую четную двоичную единицу в коде, размещенном в переменной типа int. Вывести исходные данные и полученный результат в виде, удобном для анализа проведенных преобразований.

5.41. Написать программу, обнуляющую каждую нечетную двоичную единицу в коде, размещенном в переменной типа int. Вывести исходные данные и полученный результат в виде, удобном для анализа проведенных преобразований.

5.42. Описать функцию, которая в каждом элементе беззнакового целочисленного массива заменяет старший байт нулевым кодом, если в этом байте размещен код латинской буквы.

5. СТРУКТУРЫ, ОБЪЕДИНЕНИЯ

6.1 Основные сведения

6.1. Верны ли следующие утверждения:

a) описание структуры начинается с ключевого слова struct и содержит список объявлений членов структуры, заключенный в фигурные скобки;

b) за словом struct должен следовать идентификатор, называемый тегом структуры;

c) тег структуры используется в качестве имени типа при описании переменных;

d) имена членов структуры могут совпадать с именами переменных в той же области видимости;

e) имя тега структуры может совпадать с именами переменных в той же области видимости;

f) имя тега структуры может совпадать с именами членов этой структуры;

g) имена членов разных структур могут совпадать;

h) за описанием структуры (после правой закрывающей фигурной скобки) обязательно должен следовать список переменных;

i) переменные x, y, z разных типов

1) struct s { int a; float f; } x, y;
   struct s z;

2) typedef struct { int a; float f;} s;
   s x, y;
   struct { int a; float f; } z;

3) struct s { int a; float f; };
   typedef struct s new_s;
   struct s x; new_s y, z;

4) struct s { int a; float f; };
   typedef struct s s1;
   typedef struct s s2;
   s1 x, y; s2 z;

j) переменные x, y, z одного типа

1) struct { int a; float f; } x, y;
   struct { int a; float f; } z;

2) struct { int a; float f; } x, y;
   struct { float f; int a; } z;

k) для доступа к членам структуры используется операция **.** (точка);

l) структуры не могут быть вложенными;

m) структурную переменную при ее описании можно инициализировать списком константных выражений, заключенным в фигурные скобки;

6.2. Каким образом в Си определяется эквивалентность типов? Какая эквивалентность типов рассматривается: структурная или именная? Чем они отличаются?

6.3. Описать в виде структуры следующие понятия:

a) дата (число, месяц, год);

b) адрес (страна, город, улица, дом, квартира);

c) треугольник (две стороны и угол между ними);

d) окружность (радиус и центр);

e) расписание занятий студента 209 группы факультета ВМК (день недели, предметы (с указанием – лекции или семинары), часы занятий, аудитория, фамилия преподавателя)

f) результаты проверки контрольной работы (номер группы, номер контрольной работы, тема, 25 строчек с полями: фамилия студента, вариант, информация о каждой из пяти задач (ее номер, оценка за ее решение, характеристика ошибок), итоговая оценка студента за эту контрольную работу.

6.4. Используя определенный в задаче 6.3 тип, описать переменную этого типа и присвоить ей значение:

a) дата – 16 ноября 1999 года;

b) адрес – Россия, Москва, Ильинка, дом 3, кв. 34;

c) треугольник – 5, 6.7, 35°;

d) окружность – радиус 4.567, центр (1.4, 5.6);

e) расписание занятий студента 209 группы факультета ВМК – понедельник, математический анализ (лекция) –1 пара, П-12, Ломов И.С., математический анализ (семинар) – 2 пара, 706, Григорьев Е.А., программирование (семинар) – 3 пара, 713, Пильщиков В.Н.

6.5. Что напечатает программа?

```
#include <stdio.h>
main()
  { struct data1 { char c[4]; char *s; } d1 = { "abc", "def" };
    struct data2 { char * cp; struct data1 inf; } d2 = { "ghi", { "jkl", "mno"} };
    printf("d1.c[0]=%c  *d1.s=%c\n", d1.c[0], *d1.s);
    printf("d1.c=%s  d1.s=%s\n", d1.c, d1.s);
    printf("d2.cp=%s  d2.inf.s=%s\n", d2.cp, d2.inf.s);
    printf("++d2.cp=%s  ++d2.inf.s=%s\n", ++d2.cp, ++d2.inf.s);
  }
```

6.6. Верны ли следующие утверждения:

a) описание объединения начинается с ключевого слова union и содержит список объявлений членов объединения, заключенный в фигурные скобки;

b) каждый член объединения располагается в памяти с одного и того же адреса; объем памяти для каждого члена выделяется в соответствии с его размером;

c) для каждого из членов объединения выделяется одна и та же область памяти;

d) все проблемы, связанные с выравниванием, решает компилятор;

e) в каждый момент времени объединение может содержать значение только одного из его членов;

f) все операции, применимые к структурам, применимы и к объединениям;

g) «рассогласованность» при работе с активным вариантом объединения контролируется компилятором.

6.7. Можно ли в Си создать аналог вариантных записей Паскаля?

6.8. Описать тип, с помощью которого можно организовать хранение данных о различных видах транспорта: грузовиках, автобусах, легковых автомобилях и мотоциклах. Для каждого вида транспорта имеются как общие характеристики (владелец, год производства и модель), так и индивидуальные (для грузовиков - число осей, грузоподъемность, для автобусов - число мест для пассажиров, для легковых автомобилей - число дверей (2 или 4), для мотоциклов - тип двигателя (двух- или четырехтактный)).

6.2 Структуры и функции. Указатели на структуры.

6.9. Верны ли следующие утверждения:

a) к структурам одного типа применима операция присваивания;

b) к структурам одного типа, не содержащим вложенных структур, применима операция сравнения (выполняется почленное сравнение);

c) параметром функции может быть указатель на структуру, но не сама структура;

d) параметры функции – структуры передаются по значению;

e) результатом работы функции может быть структура;

f) результатом работы функции может быть указатель на структуру;

g) функция sizeof(struct any) выдает результат, равный сумме длин всех полей этой структуры;

h) к структурам применима операция взятия адреса;

6.10. Перечислить все операции, применимые к структурам.

6.11. Пусть точка на плоскости описана следующим образом:

```
struct point { int x; int y;}
```

Верно ли решена задача: «описать функцию, которая присваивает значение структуре типа struct point »

a)
```
void assign_to_point ( struct point p, int a, int b)
{ p.x = a;  p.y = b; }
```

b)
```
void assign_to_point ( struct point *p, int a, int b)
{ (*p).x = a;  (*p).y = b; }
```

c)
```
void assign_to_point ( struct point *p, int a, int b)
{ *p.x = a;  *p.y = b; }
```

d)
```
void assign_to_point ( struct point *p, int a, int b)
{ p -> x = a;  p -> y = b; }
```

6.12. Пусть точка на плоскости описана следующим образом:

```
struct point { int x; int y;}
```

Верно ли решена задача: «описать функцию, которая создает точку из двух целых чисел»

a)
```
struct point create_point ( int a, int b)
{ struct point p;
  p.x = a;  p.y = b;  return p;
}
```

b)
```
struct point *create_point ( int a, int b)
{ struct point p;
  p.x = a;  p.y = b;  return &p;
}
```

c)
```
struct point *create_point ( int a, int b)
{ struct point *pp;
  pp -> x = a;  pp -> y = b;  return pp;
}
```

d) struct point *create_point (int a, int b)

```
{ struct point *pp;
  pp = (struct point *) malloc(sizeof(struct point));
  pp -> x = a;  pp -> y = b;  return pp;
}
```

6.13. Пусть точка на плоскости описана следующим образом:

```
struct point { int x; int y;}
```

Описать функцию, которая по трем точкам, являющимися вершинами некоторого прямоугольника, определяет его четвертую вершину;

6.14. Описать в виде структуры

a) точку на плоскости;

b) цветную точку на плоскости;

c) комплексное число;

d) рациональное число.

Разработать совокупность операций для данных этого типа; реализовать каждую из них в виде функции.

6.15. Пусть «целочисленная» окружность на плоскости описана следующим образом:

```
struct point { int x; int y;};
struct circle { int radius; struct point center;};
```

Пусть есть массив struct circle plane [50], содержащий информацию об окружностях на плоскости. Описать функцию, определяющую

a) есть ли среди этих окружностей хотя бы две концентрические окружности;

b) есть ли среди этих окружностей хотя бы две вложенные (не обязательно концентрические) окружности;

c) есть ли среди этих окружностей три попарно пересекающихся окружности;

d) есть ли среди этих окружностей хотя бы одна «уединенная», т.е. не имеющая общих точек ни с какой другой окружностью массива plane.

6.16. Пусть результаты анализа некоторого текста, состоящего из английских слов, содержатся в следующем частотном словаре dictionary:

```
#define MAXSIZE 1000

#define LENGHT 20 /* максимальная длина слова */

struct elem { char * word; struct info *data;};

struct info { int count; /* количество повторений слова в данном тексте */

              char *alias; /* синоним данного слова */

            };

struct { struct elem *tabl [ MAXSIZE];

         int number; /* количество слов в словаре */

       }

dictionary;
```

Каждое слово (word) встречается в словаре только один раз; синонимы (alias) могут быть одинаковыми у разных слов.

Описать функцию, определяющую

a) встречается ли данное слово в этом словаре: результат – указатель на соответствующий элемент либо NULL:

1) слова неупорядочены по алфавиту;
2) слова упорядочены по алфавиту;

b) какое слово чаще других встречается в анализируемом тексте; если таких слов несколько, то взять первое по алфавиту; результат работы функции – указатель на соответствующий элемент:

1) слова неупорядочены по алфавиту;
2) слова упорядочены по алфавиту;

c) на каждую ли букву латинского алфавита в этом словаре найдется хотя бы одно слово:

1) слова неупорядочены по алфавиту;
2) слова упорядочены по алфавиту;

d) на какие буквы (букву) латинского алфавита в этом словаре больше всего слов; результат работы функции – указатель на строку, составленную из этих букв, перечисленных в алфавитном порядке:

1) слова неупорядочены по алфавиту;
2) слова упорядочены по алфавиту;

e) есть ли в словаре различные слова, имеющие одинаковые синонимы.

6.17. Пусть результаты анализа некоторого текста содержатся в частотном словаре (см. предыдущую задачу).

Описать функцию

a) struct elem *add_word (char * word, char *alias), вставляющую новое слово и информацию о нем в словарь dictionary (предполагается, что такого слова в словаре еще нет, оно первый раз встретилось в тексте);

1) слова неупорядочены по алфавиту;
2) слова упорядочены по алфавиту;

Результат работы функции – указатель на вставленный элемент.

b) struct elem *update_word (char * word, char *alias), изменяющую значение синонима для данного слова (предполагается, что такое слово в словаре обязательно есть);

1) слова неупорядочены по алфавиту;
2) слова упорядочены по алфавиту;

Результат работы функции – указатель на элемент словаря, где изменено значение синонима.

c) struct elem *delete_word (char *word), удаляющую данное слово из словаря; результат работы функции – указатель на структуру, содержащую информацию об удаленном слове либо NULL, если такого слова нет в словаре;

1) слова неупорядочены по алфавиту;
2) слова упорядочены по алфавиту;

6.18. Программа. Определить число вхождений каждого служебного слова в данную программу на Си. Тщательно продумать способ представления информации о служебных словах.

6.19. Допустимо ли в Си? Если "да" - опишите семантику этих действий, объясните, что будет напечатано; если "нет" - объясните почему.

```
#include <stdio.h>
struct data { char *s; int i; struct data *dp; };
main()
  { static struct data a[ ] = {  { "abcd", 1, a+1 },
                                 { "efgh", 2, a+2 },
                                 { "ijkl",3, a }
                               };
    struct data  *p = a;  int i;
    printf("a[0].s=%s  p -> s=%s  a[2].dp -> s=%s\n",
            a[0].s, p -> s, a[2].dp -> s);
    for ( i = 0; i < 2; i++ ) printf("--a[i].i=%d  ++a[i].s[3]=%c\n",
                                    --a[i].i, ++a[i].s[3]);
    printf("++(p->s)=%s\n", ++(p->s) );
    printf("a[(++p) -> i].s=%s\n", a[(++p) -> i].s);
    printf("a[--(p -> dp -> i)].s=%s\n", a[--(p -> dp -> i)].s);
  }
```

6.20. Пусть

```
struct s { int k; float *f; char *p[2];};
struct s *ps;
```

Верно ли присвоено значение переменной ps и всем объектам, с ней связанным:

```
char str[5] = “abcd”;
ps = (struct s *) malloc(sizeof(struct s));
(*ps).k = 5;
ps -> f = (float *) malloc(sizeof(float)); *(ps -> f) = 3.1415;
(*ps).p[0] = (char*) malloc(5*sizeof(char));
(*ps).p[1] = (char*) malloc(10*sizeof(char));
(*ps).p[0] = str;
```

(*ps).p[1] = “abcdefghi”;

6.21. Присвоить значение переменной q и всем объектам, с ней связанным:
struct data { double **p; char *s; int *a[2]; };

struct data *q;

6.22. Присвоить значение переменной a и всем объектам, с ней связанным:
struct b { double *q; int * (*p)[2]; };

struct b **a[1];

6.23. Присвоить значение переменной x и всем объектам, с ней связанным:
struct r { double *a[3]; char **s;

union { int i; float f; } u;

}

struct r x[2];

6.24. Присвоить значение переменной x и всем объектам, с ней связанным:
struct a { char ***s;

char (*p)[2];

};

typedef struct a * data;

data x[2];

6.25. Присвоить значение переменной pt и всем объектам, с ней связанным:
struct t { int **pi;

double (*k)(double,int*);

char *p[2];

};

struct t *pt;

6.26. Присвоить значение переменной a и всем объектам, с ней связанным:
struct data { int *i; int (*f)(int); char **s; };

struct data a[2];

6.3 Структуры со сылками на себя

Замечание: в задачах 6.27 – 6.40 речь идет об однонаправленных списках без заглавного звена (если не сказано особо в постановке задачи);

```
struct lnode {  type data; /* поле данных */
                  struct lnode *next; /*указатель на следующее звено списка */
               }
```

Здесь struct lnode определяет структуру звена списка. Термин “элемент” будем использовать и для звена, и для поля данных в звене, если это не приводит к неоднозначности. Тип данных type уточняется в каждой задаче.

6.27. Описать функцию, которая

a) находит сумму всех элементов списка;

b) находит максимальный элемент в заданном непустом списке;

c) проверяет, упорядочены ли по возрастанию элементы списка.

d) находит сумму минимального и максимального элементов в списке ;

Тип данных – int .

6.28. Описать функцию, которая

a) меняет местами первый и последний элементы списка;

b) удаляет из списка первое вхождение элемента с заданным значением (если оно есть);

1) список с заглавным звеном;
2) список без заглавного звена;

c) удаляет из списка все вхождения элемента с заданным значением (если они есть);

1) список с заглавным звеном;
2) список без заглавного звена;

d) после каждого звена с заданным значением вставляет еще одно звено с таким же значением.

Тип данных - double*, анализируются вещественные числа.

6.29. Описать функцию, которая определяет, есть ли в заданном списке хотя бы два одинаковых элемента. Тип данных – int.

6.30. Описать функцию, которая печатает в обратном порядке значения элементов списка. Тип данных - double.

6.31. Описать функцию, которая заменяет в списке все вхождения данного слова на удвоенное. Тип данных - char*.

6.32. Описать функцию, которая строит список L2 - копию списка L1.

```
struct lnode {  struct data *p;
                struct lnode *next; };
struct data { double f; char *s[2];};
```

6.33. Описать функцию, которая переворачивает список, изменяя ссылки.

```
struct lnode { struct data *p;
                struct lnode *next; };
struct data { double f; char *s[2];};
```

6.34. Описать функцию, которая проверяет, входит ли список L1 в список L2. Тип данных - char*, анализируются строки, указатели на которые хранятся в звеньях списка.

6.35. Описать функцию, которая выполняет слияние двух упорядоченных по возрастанию списков L1 и L2, строя третий список L3:

a) список L3 состоит из звеньев списков L1и L2;

b) список L3 строится из копий звеньев списков L1 и L2; списки L1 и L2 не изменяются.

Тип данных - int*.

6.36. Описать функцию, которая после последнего вхождения элемента E (структуры типа data) в список L1 вставляет список L2, изменяя ссылки.

struct lnode { struct data * dtpr; struct lnode *next; };

struct data { int i; char *s; };

6.37. Описать функцию, которая в упорядоченный по возрастанию список вставляет элемент, сохраняя упорядоченность. Тип данных - char*.

6.38. Описать функцию, которая формирует список L3, включая в него элементы списка L1, которые не входят в список L2.

a) список L3 состоит из звеньев списка L1;

b) список L3 строится из копий звеньев списка L1; список L1 не изменяется.

Тип данных - int*.

6.39. Описать функцию, которая формирует список L3, включая в него в одном экземпляре элементы, входящие в список L1 и в список L2. Список L3 формируется из копий звеньев списков L1 и L2; списки L1 и L2 не изменяются.

Тип данных - char*.

6.40. Описать функцию, которая формирует список L3, включая в него элементы, которые входят в один из списков (L1 или L2), но при этом не входят в другой. Список L3 формируется из копий звеньев списков L1 и L2; списки L1 и L2 не изменяются.

Тип данных - char*.

Замечание: в задачах 6.41 – 6.44 требуется разработать и реализовать несколько абстрактных типов данных (АТД). Абстрактный тип данных – это тип, определяемый программистом, для которого он описывает структуру значений этого типа и множество операций с такими данными. Детали реализации АТД по возможности максимально скрыты от пользователя, и оперировать с такими данными можно только с помощью предоставленных операций (аналогично тому, как пользователь работает с предопределенными в языке типами данных). Поэтому, создавая АТД, надо тщательно

продумать, какие операции предоставить пользователю, чтобы их было достаточно для выполнения традиционных действий с этими типами данных.

6.41. Разработать способ представления «разреженных» многочленов (многочленов с целыми коэффициентами, большинство из которых равно нулю). Продумать набор операций для работы с данными такого типа (создание такого многочлена, вычисление значения многочлена в некоторой точке, сложение двух многочленов с приведением подобных членов, дифференцирование многочлена и т.п.). Каждую из этих операций реализовать в виде функции.

6.42. Описать эффективный способ представления АТД «очередь»: описать структуру этого типа данных, разработать и реализовать набор операций для данных этого типа (в частности, позаботиться о действиях при переполнении очереди).

6.43. Описать способ представления АТД «стек»: описать структуру этого типа данных, разработать и реализовать набор операций для данных этого типа.

6.44. Описать способ представления АТД «набор» Набор – это аналог множества, но в наборе (в отличие от множества) может содержаться несколько экземпляров одного элемента. Разработать и реализовать набор операций для данных этого типа.

Замечание: в задачах 6.45 – 6.47 речь идет о двоичных деревьях;

```
struct tnode {  type data; /* поле данных */
                    struct tnode *left; /*указатель на левый узел */
                    struct tnode *right; /*указатель на правый узел */
                }
```

Здесь struct tnode определяет структуру узла дерева. Термин “элемент” будем использовать и для узла, и для поля данных в узле, если это не приводит к неоднозначности. Тип данных type уточняется в каждой задаче.

6.45. Используя определенные в задачах 6.42 и 6.43 АТД «очередь» и «стек», описать нерекурсивную функцию, которая

а) определяет число вхождений данного элемента в двоичное дерево;

b) вычисляет сумму элементов двоичного дерева;

c) находит длину (количество узлов) на пути от корня дерева до ближайшего узла, содержащего данный элемент (если такого узла в дереве нет, то считать результат равным -1);

d) определяет, является ли данное дерево деревом двоичного поиска (т.е. по отношению к любому узлу в этом дереве его левое поддерево содержит только те данные, значения которых меньше значения данного узла, а его правое поддерево содержит только те данные, значения которых больше значения данного узла);

e) подсчитывает количество узлов на N-ом уровне непустого двоичного дерева (корень считать узлом нулевого уровня);

f) печатает все элементы двоичного дерева по уровням, начиная с корня, на каждом уровне – слева направо.

Тип данных – int .

6.46. Описать рекурсивную функцию, которая

a) определяет число вхождений данного элемента в двоичное дерево;

b) вычисляет сумму элементов двоичного дерева;

c) определяет, входит ли данный элемент в двоичное дерево;

d) печатает значения данных из всех узлов дерева, не являющихся листьями;

e) проверяет, идентичны ли два двоичных дерева;

Тип данных – int .

6.47. Описать функцию, которая в дерево двоичного поиска вставляет новый элемент (определение дерева двоичного поиска см. задачу 6.45(d)).

6.48. Программа. Упорядочить по алфавиту и распечатать все слова входного текста.

6. ВВОД-ВЫВОД

7.1 Стандартный ввод-вывод

7.1. Программа. Даны натуральные числа i, n ($i \le n$), вещественные числа a_1, a_2, …,a_n. Найти среднее арифметическое всех чисел, кроме a_i.

7.2. Программа. Даны вещественные числа a_1, a_2, …,a_{50}. Распечатать “сглаженные” значения a_1, a_2, …,a_{50} , заменив в исходной последовательности все члены, кроме первого и последнего, по формуле

$$a_i = (a_{i-1} + a_i + a_{i+1})/3 \quad i = 2, 3, \dots ,49;$$

считая, что

a) после того, как получено новое значение некоторого члена последовательности, оно используется для вычисления нового значения следующего за ним члена последовательности;

b) при “сглаживании” используются лишь старые члены последовательности.

7.3. Программа. Даны вещественные числа a_1, a_2, … Известно, что $a_1 > 0$ и что среди a_2, a_3, … есть хотя бы одно отрицательное число. Пусть a_1, a_2, …,a_n - члены данной последовательности, предшествующие первому отрицательному члену (n заранее неизвестно). Распечатать

a) $a_1 + a_2 + \dots + a_n$

b) $a_1 * a_2 * \dots * a_n$

c) среднее арифметическое a_1, a_2, …,a_n

d) a_1 , $a_1 * a_2$, $a_1 * a_2 * a_3$,…, $a_1 * a_2 * \dots * a_n$

e) $a_1 + 2*a_2 + 3*a_3 + \dots + (n-1)*a_{n-1} + \dots n*a_n$

f) $| a_1 - a_2 |, | a_2 - a_3 | , \dots , | a_{n-1} - a_n |, | a_n - a_1 |$

7.4. Программа. Даны целые положительные числа n, a_1, a_2, …, a_n ($n \ge 4$). Считать, что a_1, a_2, …, a_n - это измеренные в сотых долях секунды результаты n спортсменов в беге на 100 метров. По этим результатам составить команду из четырех лучших бегунов для участия в эстафете 4×100, т.е. распечатать номера спортсменов, имеющих четыре лучших результата.

7.5. Верно ли решена следующая задача: «читать символы из стандартного входного потока, пока код каждого следующего символа больше кода предыдущего; определить, сколько символов было прочитано»

a) ... i = 0;

```
while ( getchar() < getchar() ) i = i + 2;
```

b) …

```
i = 0; c = getchar();
while ( c < ( c = getchar() ) ) i++;
```

c) …

```
i = 0; c = getchar();
while ( c < ( d = getchar()) ) { i++; c = d;}
```

d) …

```
i = 0; c = getchar();
while (d = getchar(), c<d)) { i++; c = d; }
```

e) …

```
i = 0; c = getchar();
while ( c != EOF && ( d = getchar()) != EOF && c<d ) { i++; c=d; }
```

7.6. Сравнить следующие фрагменты программы:

a) `while (c = getchar() = EOF)`

b) `while (c = getchar() == EOF)`

c) `while ((c = getchar()) == EOF)`

d) `while ((c = getchar()) = -1)`

7.7. Допустимо ли в Си? Если "да" - опишите семантику этих действий; если "нет" - объясните почему.

```
int i,k,sum;
for ( i=1; scanf("%d",&k) == 1; i++)
   printf("i = %d, k = %d, sum = %d\n", i, k, sum+=k );
```

7.8. Программа. Дана непустая последовательность слов, разделенных одним или несколькими пробелами. Признак конца текста – точка. Распечатать этот текст, удалив из него лишние пробелы (каждую группу из нескольких пробелов заменить одним пробелом).

7.9. Программа. Дана непустая последовательность слов из прописных (больших) латинских букв. Слова разделены пробелом; признак конца текста – точка.

а) подсчитать количество слов в этом тексте;

b) подсчитать количество слов, у которых совпадают первая и последняя буквы;

c) подсчитать количество слов, являющихся некоторым фрагментом латинского алфавита;

d) подсчитать количество слов, содержащих все буквы, которые входят в состав слова UNIX.

7.10. Программа. Дана непустая последовательность слов, разделенных пробелом; признак конца текста – точка. Длина каждого слова – не более 20 литер.

a) распечатать все слова, у которых не совпадают первая и последняя буквы;

b) распечатать все слова, являющиеся «перевертышами», т.е. словами, одинаково читающимися слева направо и справа налево;

c) распечатать текст, оставив из рядом стоящих одинаковых слов только одно;

d) распечатать текст, удалив все слова, где есть символы, отличные от латинских букв.

7.11. Программа. Дана непустая последовательность слов, разделенных пробелом; признак конца текста – точка. Длина каждого слова – не более 20 литер. Распечатать данный текст следующим образом: все строки должны быть одинаковой длины (длина строки задается в командной строке); каждое слово должно быть распечатано в одной строке без переносов; если в строке несколько слов, то пробелы между ними должны быть равномерно распределены; если в строке помещается только одно слово и его длина меньше длины строки, то оно должно быть выровнено по левому краю; если длина слова больше длины строки, то такие слова из текста удаляются, при этом после распечатки текста о каждом таком слове выдается предупреждение.

7.12. Программа. Дана непустая последовательность слов из строчных (малых) латинских букв. Слова разделены пробелом; признак конца текста – точка. Напечатать все буквы, которые

a) чаще других встречаются в данном тексте;
b) входят в каждое слово данного текста;
c) входят в наибольшее количество слов данного текста;

7.2 Работа с файлами

Замечание: в задачах 7.13 – 7.21 содержимое файлов только анализируется; в остальных задачах этого раздела создается новый файл(ы) либо изменяется содержимое данного.

7.13. Программа. Определить, сколько раз в данном файле f встречается символ ‘A’.

7.14. Программа. Определить, сколько раз в данном файле g встречается строка UNIX.

7.15. Программа. Распечатать все строки данного файла, содержащие заданную строку в качестве подстроки. Имя файла и строка задаются в командной строке.

7.16. Написать программу, определяющую какой символ чаще других встречается в данном файле. Имя файла задается в командной строке.

7.17. Написать программу, определяющую сколько строк, состоящих из одного, двух, трех и т.д. символов, содержится в данном файле. Считать, что длина каждой строки - не более 80 символов. Имя файла задается в командной строке.

7.18. Программа. Определить, какая строка является самой длинной в заданном файле. Если таких строк несколько, то в качестве результата выдать первую из них. Имя файла задается в командной строке.

7.19. Программа. Даны два непустых файла. Определить номер строки и номер символа в этой строке, где встречается первый символ, отличающий содержимое одного файла от другого. Если содержимое файлов полностью совпадает, то результат – 0, 0 и соответствующее сообщение; если один из файлов является началом другого, то результат - n+1, 1, где n - количество строк в коротком файле, и соответствующее сообщение. Имена файлов задаются в командной строке.

7.20. Программа. В файле записана непустая последовательность целых чисел (целое число – это непустая последовательность десятичных цифр, возможно начинающаяся знаком + или -). Имя файла задается в командной строке.

a) найти наибольшее из этих чисел;
b) определить, сколько четных чисел содержится в файле;
c) определить, составляют ли эти числа арифметическую прогрессию;

d) определить, образуют ли эти числа возрастающую последовательность;

e) определить, сколько чисел этой последовательности являются точными квадратами;

7.21. Написать программу, определяющую, какая из строк чаще других встречается в данном файле.

7.22. Написать программу, создающую файл - копию заданного файла. Имена файлов задаются в командной строке.

7.23. Программа. Создать файл, являющийся конкатенацией других файлов. Имена файлов задаются в командной строке: fres f1 f2 …, где fres - имя файла-результата, f1, f2, … - файлы, содержимое которых должно быть записано в файл-результат.

7.24. Программа. Дан файл f. Создать файл g, полученный из файла f заменой всех его прописных латинских букв соответствующими строчными.

7.25. Программа. Дан файл f. Создать два файла f1 и f2 следующим образом: в файл f1 записать в том же порядке все строки из файла f, состоящие только из латинских букв (прописных и строчных);в файл f2 – строки файла f, состоящие только из цифр; все остальные строки файла f не записываются ни в один из этих файлов.

7.26. Программа. В конец файла f приписать строку FINISH.

7.27. Программа. В конец файла f приписать содержимое файла g.

7.28. Программа. Даны два файла, строки в которых упорядочены по алфавиту. Написать программу, осуществляющую слияние этих двух файлов в третий, строки которого тоже упорядочены по алфавиту. Имена всех трех файлов задаются в командной строке.

7.29. Программа. Дан файл и две строки. Все вхождения первой строки в файл заменить второй строкой (вхождения первой строки в качестве подстроки не рассматривать). Имя файла и строки задаются в командной строке.

7.30. Программа. Дан файл и две строки. Все вхождения первой строки в файл (в том числе и в качестве подстроки) заменить второй строкой. Имя файла и строки задаются в командной строке.

7.31. Программа. В данном файле символы каждой строки упорядочить по алфавиту. Имя файла задается в командной строке.

7.32. Программа. Строки данного файла упорядочить по алфавиту. Имя файла задается в командной строке.

7.33. Программа. В данном файле упорядочить все строки по возрастанию их длин. Имя файла и максимальная длина строки задаются в командной строке.

7.34. Программа. В файле записана непустая последовательность целых чисел, являющихся числами Фибоначчи (см. задачу 3.34). Приписать еще одно, очередное число Фибоначчи.

7.35..Программа. В файле записана непустая последовательность целых чисел (целое число – это непустая последовательность десятичных цифр, возможно начинающаяся знаком + или -). Создать новый файл, где

a) все отрицательные числа заменены нулем;

b) минимальный элемент последовательности поставлен в ее начало, а максимальный – в конец;

c) переставлены максимальный и минимальный элементы этой последовательности;

d) удалены все числа, являющиеся полными квадратами.

Имена файлов задаются в командной строке.

7. ИНТЕРФЕЙС С СИСТЕМОЙ UNIX

8.1 Низкоуровневый ввод-вывод

Замечание: во всех задачах этого раздела при вводе-выводе использовать низкоуровневые средства системы UNIX.

8.1. Верно ли решена задача: «написать версию функции int getchar(void), которая осуществляет небуферизованный ввод, читая по одной литере из входного потока»

a) int getchar (void)

```
{  char c;
   return ( read (0, &c, 1) == 1) ? c : EOF);
}
```

b) int getchar (void)

```
{  int c;
   return ( read (0, &c, 1) == 1) ? c : EOF);
}
```

c) int getchar (void)

```
{  char c;
   return ( read (0, &c, 1) == 1) ? (unsigned char)c : EOF);
}
```

d) int getchar (void)

```
{  char c;
   return ( read (0, &c, 1) == 1) ? (int)c : EOF);
}
```

8.2. Написать буферизованный вариант функции int getchar(void), когда функция осуществляет ввод большими порциями, но при каждом обращении к ней выдает только одну литеру.

8.3. Используя низкоуровневый ввод-вывод, реализовать следующие функции:

a) int putchar (int c)

b) char *gets (char *s)

c) int puts (const char *s)

8.4. Написать программу, копирующую свой стандартный ввод в стандартный вывод.

8.5. Написать программу, создающую файл - копию заданного файла. Имена файлов задаются в командной строке.

a) копирование по одной литере;

b) копирование блоками;

8.6. Программа. Создать файл, являющийся конкатенацией других файлов. Имена файлов задаются в командной строке (см. задачу 7.23).

8.7. Описать функцию, удваивающую в заданном файле каждую очередную четверку байт.

8.8. Программа. В каждом из данных файлов удалить те N–ки байт, в которых первый байт равен коду символа s. Имена файлов, символ s и величина N задаются в командной строке.

8.9. Описать функцию, определяющую количество символов s в тексте, состоящем из нечетных N-ок байт заданного файла. Имя файла, символ s и величина N – параметры функции.

8.10. Программа. Создать файл, содержащий значения функции $\sin(x)*\cos(x)*\exp(x)$ на отрезке [a,b] в точках $x_i = a+i*h$, $h = (b-a)/n$, $i = 0,1,\ldots,n$; имя файла и значения a, b, n задаются в командной строке.

8.11. Программа. В файле записана последовательность целых чисел. Создать файл, состоящий из чисел данного файла, значения которых меньше N. Имена файлов и величина N задаются в командной строке.

8.12. Программа. В конец файла f приписать

a) число 1234;

b) строку “end”;

8.13. Программа. В конец файла f приписать содержимое файла g.

8.14. Написать программу, приписывающую в конец файла f его содержимое.

8.15. Описать функцию char *get (char *f, int n, int pos), читающую n байт из файла f, начиная с позиции pos.

8.16. Программа. Содержимое файлов, длина которых меньше N байт, переписать в новый файл-результат и удалить такие файлы. Файлы, длина которых больше либо равна N байт, не изменяются и не удаляются. Имена файлов и величина N задаются в командной строке: fres f1 f2 …, где fres - имя файла-результата, f1, f2, … - файлы, содержимое которых должно быть проанализировано.

8.17. Написать программу слияния двух файлов в третий. Файл -результат формируется чередованием N-ок символов первого и второго файлов (если один из файлов длиннее другого, то его оставшаяся часть приписывается в конец файла-результата). Имена файлов и величина N задаются в командной строке.

8.2 Процессы, сигналы

8.2.1 Конвейер, перенаправление ввода-вывода

8.18. Написать программу, моделирующую команду SHELL: (здесь pri - имена процессов, argj - аргументы процессов, f.dat - файл входных данных, f.res - файл результатов; в каждом из процессов pri использован стандартный ввод-вывод). Аргументы, необходимые этой программе, задаются в командной строке.

a) pr1 | pr2 | pr3

b) pr1 | pr2 > f.res

c) pr1 arg11 arg12 < f.dat | pr2 arg21 agr22

d) pr1 < f.dat > f.res

e) pr1 < f.dat | pr2 | pr3 > f.res

f) pr1 | pr2 >> f.res

g) pr1; pr2 | pr3 > f.res

h) ((pr1 | pr2); pr3) | pr4

i) pr1 arg1 < f.dat; pr2 | pr3 >>f.res

j) pr1 arg1 < f.dat | pr2 | pr3 >f.res &

k) pr1 arg1 > f.res ; pr2 | pr3 >> f.res

l) pr1 < f.dat | pr2 arg2 ; pr3 > f.res

m)pr1; pr2; ... ; prn

n) pr1; pr2; ... ; prn &

o) pr1 | pr2 | ... | prn

p) pr1 | pr2 | ... | prn &

8.19. Написать программу, моделирующую команду SHELL pr1&&pr2 (выполнить pr1; в случае успешного завершения pr1 выполнить pr2, иначе завершить работу). Имена процессов задаются в командной строке.

8.20. Написать программу, моделирующую команду SHELL pr1 || pr2 (выполнить pr1; в случае неудачного завершения pr1 выполнить pr2, иначе завершить работу). Имена процессов задаются в командной строке.

8.21. Написать программу, моделирующую выполнение команды (pr1;pr2) | pr3 > f.res (конкатенация результатов работы процессов pr1 и pr2 передается в качестве входных данных процессу pr3; результаты его работы перенаправляются в файл f.res; в процессах pr1, pr2 и pr3 использован стандартный ввод-вывод).

a) аргументы задаются в командной строке в виде pr1 pr2 pr3 f.res

b) команда (pr1;pr2) | pr3>f.res вводится как строка во время работы программы.

Подсказка: для разбиения строки на лексемы удобно использовать функцию strtok из <string.h>

8.22. Написать программу, реализующую конвейер из n процессов. Информация, необходимая для запуска каждого процесса, задается в командной строке: имя_процесса количество_аргументов аргументы

Например, pr1 3 a1 a2 a3 pr2 0 pr3 2 b1 b2 означает, что программа должна выполнить команду pr1 a1 a2 a3 | pr2 | pr3 b1 b2.

8.23. Пусть есть исполняемый файл exp_x со стандартным вводом-выводом, вычисляющий значение exp(x). Результат работы exp_x - значения x и exp(x). Написать программу, использующую этот файл для нахождения таблицы значений exp(x_i) на

отрезке [a,b] в точках x_i=a+i*h, i = 0, 1, ..., n; h = (b-a)/n. Значения a, b и n вводятся с клавиатуры. Таблица выводится в стандартный поток вывода.

8.24. Пусть есть исполняемый файл func со стандартным вводом-выводом, вычисляющий значение функции f в точке x. Результат работы - значения x и f(x). Написать программу, использующую этот файл для нахождения таблицы значений f(x) на отрезке [a,b] в точках x_i =a+i*h, i = 0,1,...,n; h = (b-a)/n. Значения a, b и n задаются в командной строке. Таблица записывается в файл f.tab.

8.25. Пусть есть исполняемый файл modify, который удаляет в текстовом файле все четные строки (строки нумеруются с единицы; пустые строки тоже анализируются). Имя файла задается в командной строке при вызове modify. Написать программу, использующую modify для обработки файлов, имена которых задаются в командной строке. Если первая строка файла совпадает со второй его строкой, то в файле оставить только нечетные строки; иначе файл не изменять. Анализ файлов выполняет основной процесс, изменения в фоновом режиме осуществляет modify (для каждого файла – свой экземпляр modify).

8.26. Пусть есть исполняемый файл modify, который удаляет в текстовом файле все четные строки (строки нумеруются с единицы; пустые строки тоже анализируются). Имя файла задается в командной строке при вызове modify. Написать программу, использующую modify для обработки файлов, имена которых задаются в командной строке. Если после сортировки файла, состоящего из нечетных строк исходного файла, оказалось, что каждая очередная строка начинается со следующей по алфавиту буквы (начиная с буквы ‘a’ до буквы ‘z’ и далее циклически), то имя этого файла выдать на экран, иначе - файл удалить. Для сортировки использовать команду sort.

8.27. Написать программу, определяющую количество литер, слов и строк в тексте, состоящем из нечетных N-ок байт заданного файла. Для подсчета количества литер, слов и строк использовать команду wc. Результаты поместить в файл. Имена файлов и величина N задаются в командной строке.

8.28. Написать программу, сортирующую по алфавиту строки текста, состоящего из четных строк данного файла. Для сортировки использовать команду sort. Результаты сортировки поместить в файл. Имена файлов задаются в командной строке.

8.29. Написать программу, определяющую количество литер, слов и строк в тексте, состоящем из тех строк заданного файла, которые содержат данную строку-

шаблон в качестве подстроки. Используйте команды wc и grep. Результаты поместить в файл. Имена файлов и строка-шаблон задаются в командной строке.

8.30. Написать программу, которая выводит на экран имена файлов f_k, содержащих не менее n_k строк, включающих заданную строку str_k в качестве подстроки. Имена файлов f_k, величины n_k и строки-шаблоны str_k задаются в командной строке в виде f_1 n_1 str_1 f_2 n_2 str_2 ... f_k n_k str_k. Использовать команды grep и wc.

8.31. Пусть есть исполняемый файл file_dbl, который удваивает в обрабатываемом файле каждую очередную порцию из 128 байт. Имя файла задаётся в командной строке при запуске file_dbl. Написать программу, использующую file_dbl для обработки файлов, имена которых задаются в командной строке. Если длина файла меньше 1024 байт, то увеличить его размер с помощью file_dbl, иначе оставить без изменения. Анализ файлов выполняет основной процесс, изменения в фоновом режиме осуществляет file_dbl (для каждого файла свой экземпляр file_dbl).

8.32. Пусть файл содержит текст и команды форматирования. Эти команды располагаются на отдельных строках и начинаются символами **./**. Написать программу для подсчета символов в тексте (без учета символов форматирующих команд). Головной процесс анализирует содержимое файла и передает вспомогательному процессу текст без форматирующих команд. Вспомогательный процесс подсчитывает количество символов в получаемом тексте и возвращает полученный результат головному процессу. Имя файла головной процесс получает из командной строки.

8.33. Что делает программа?

```
#include <stdio.h>
void Start ( char *name, int in, int out)
  { if (fork() == 0)
      { dup2(in,0);
        dup2(out,1);
        close(in); close(out);
        execlp(name,name,0);
      }
```

```
  }
main(int argc, char *argv[])
  { int i, fd[2], in=0, out;
    for ( i = 1; i < argc-1; i++)
       { pipe(fd); out=fd[1];
         Start(argv[i], in, out);
         close(in); close(out);
         in = fd[0];
       }
    out = 1;
    Start(argv[ i ], in, out);
  }
```

8.2.2 Сигналы. Фоновые процессы.

8.34. Написать программу игры в "пинг-понг" двух процессов через два канала. Первый процесс посылает второму 1, второй первому – 2, первый второму – 3, второй первому – 4 и т.д. Игра завершается при нажатии клавиш Ctrl+C.

8.35. Написать программу игры в "пинг-понг" двух процессов (см. предыдущую задачу) через один канал. Для синхронизации использовать сигнал. Игра завершается при нажатии клавиш Ctrl+C.

8.36. Написать программу игры в "волейбол" трех процессов: первый посылает второму "1", второй третьему - "2", третий первому - "3", первый второму - "4" и т.д. Игра завершается при нажатии клавиш Ctrl+C. Работу процессов синхронизировать с помощью сигналов.

8.37. Написать программу игры в "волейбол" трех процессов (см. предыдущую задачу). Игра завершается при нажатии клавиш Ctrl+C. Работу процессов синхронизировать с помощью канального чтения.

8.38. Написать программу игры одного процесса с двумя другими: первый процесс посылает второму 1, затем третьему 'a'; после этого он получает от второго 2, затем от третьего - 'b.' На следующем шаге первый посылает второму 3, третьему - 'c'; получает от второго 4, от третьего - 'd' и т.д. именно в такой последовательности с увеличением числа и изменением символа от 'a' до 'z' циклически. Игра завершается при нажатии клавиш Ctrl+C. Для синхронизации использовать сигналы.

8.39. Написать программу, определяющую самую длинную строку в заданном файле. Если таких строк несколько, то в качестве результата выдать первую из них. Обеспечить возможность работы программы в фоновом режиме и в обычном (с обработкой прерываний по Ctrl+C: при каждом нажатии этих клавиш программа должна выдавать промежуточный результат - самую длинную из уже просмотренных строк). Имя файла задается в командной строке.

8.40. Написать программу, заполняющую файл N строками. Аргументы (имя файла, количество строк N и строка-заполнитель) задаются в командной строке. Обеспечить возможность работы программы в фоновом режиме и в обычном (с обработкой прерываний по Ctrl+C: при каждом нажатии этих клавиш программа должна выдавать промежуточный результат - количество строк, записанных в файл к этому моменту).

8.41. В файле записана непустая последовательность целых чисел (целое число – это непустая последовательность десятичных цифр, возможно начинающаяся знаком + или -). Написать программу для нахождения наибольшего из этих чисел. Во время ее работы каждую секунду выдается промежуточный результат - наибольшее из уже просмотренных чисел. Имя файла задается в командной строке.

8.42. Даны два файла, строки в которых упорядочены по алфавиту. Написать программу, осуществляющую слияние этих двух файлов в третий, строки которого тоже упорядочены по алфавиту. Имена всех трех файлов задаются в командной строке. Обеспечить возможность работы программы в фоновом и в обычном режиме (с обработкой прерываний по Ctrl+C: первое нажатие этих клавиш не влияет на работу программы; все последующие нажатия вызывают печать количества литер, слов и строк в частично сформированном файле-результате. Для подсчета количества литер, слов и строк использовать команду wc).

8.43. Написать программу слияния двух файлов в третий. Файл -результат формируется чередованием N-ок символов первого и второго файлов (если один из файлов длиннее другого, то его оставшаяся часть приписывается в конец файла-результата). Имена файлов и величина N задаются в командной строке. Исходные файлы читаются разными процессами; эти же процессы по очереди записывают по N байт из обрабатываемых ими файлов в файл-результат. Синхронизацию их работы организовать с помощью сигналов.

8.44. Написать программу нахождения корня уравнения f(x) = 0 с точностью eps > 0 на некотором отрезке [a,b] (любым известным Вам методом: деления отрезка пополам, хорд, касательных, комбинированным), которая после каждого нажатия клавиш Ctrl+C выдает очередное приближение и запрос о дальнейших действиях:

C - продолжать вычисления;

A - закончить работу программы;

R - начать поиск корня этого же уравнения на другом отрезке (новые значения a и b вводятся с клавиатуры).

Затем выполняет эти действия. Если корень был найден (с заданной точностью eps) до нажатия клавиш Ctrl+C, то выдается соответствующее сообщение, печатается результат и программа прекращает работу.

8.45. Написать программу вычисления определенного интеграла функции f(x) на отрезке [a,b] с точностью eps (любым известным Вам методом: прямоугольников, трапеций, Симпсона), которая при возникновении сигнала SIGFPE (арифметическая ошибка: деление на 0 или переполнение) выдает значение частичной суммы, количество точек разбиения и запрос о дальнейших действиях:

A - закончить работу программы;

R - вычислять значение интеграла на другом отрезке (новые значения a и b вводятся с клавиатуры; реакция на сигнал SIGFPE сохраняется).

Затем выполняет эти действия. Если вычисление интеграла (с заданной точностью) успешно завершилось, то выдается соответствующее сообщение, печатается результат и программа прекращает работу.

8.46. Написать программу копирования содержимого одного файла в другой. Копирование осуществляет вспомогательный процесс. Если во время копирования считывается строка, длина которой больше N, то этот процесс сообщает процессу-родителю о возникшей ситуации. Головной процесс спрашивает пользователя о том, что делать с этой строкой:

D - не записывать строку в формируемый файл

C - записать только первые N символов

A - прекратить копирование

и сообщает вспомогательному процессу о принятом пользователем решении. Головной процесс ждет, когда вспомогательный закончит свою работу, сообщает пользователю о том, что копирование завершено и завершается сам. Имена файлов и величина N задаются в командной строке.

8.47. Написать программу копирования из одного файла в другой только тех восьмерок байт, в которых первый символ равен заданному. Использовать низкоуровневый ввод/вывод. Все аргументы (файлы и символ) задаются в командной строке. Программа в ответ на первые два нажатия клавиш Ctrl+C выдает количество восьмерок байт, записанных в файл-результат к этому моменту, после третьего нажатия - прекращает работу, выдав содержимое сформированного к этому моменту файла. Если не было третьего (второго, первого) Ctrl+C, то работа продолжается до тех пор, пока не будет проанализирован исходный файл и создан файл-результат.

8.48. Написать программу, выдающую на экран содержимое файла порциями по N строк: каждая последующая порция выдается после нажатия клавиш Ctrl+C. Имя файла и величина N задаются в командной строке.

ГЛОССАРИЙ

Узбекский язык	Английский язык	Русский язык	СОДЕРЖАНИЕ
Kirish	**Access**	**Access**	– это универсальная система управления базами данных. Предназначена для создания и ведения баз данных, для организации запросов, всевозможных выборок и отчетов. Содержит средства для связывания таблиц и связис другими пакетами прикладных программ.
CMOS	**CMOS**	**CMOS**	-память предназначена для хранения наиболее важной информации о параметрах настройки компьютера. В ней запоминается пароль пользователя, если он был установлен, текущее время и дата.
DVD disklari.	**DVD discs**	**DVD-диски.**	Изначально эти диски предназначены для записи видеофильмов. DVD могут иметь по два несущих слоя с каждой стороны. За счет увеличения плотности записи каждый слой имеет информационную емкость 4,7 Мб.
Excel	**Excel**	**Excel**	Excel для WINDOWS является мощным программным средством для работы с таблицами данных, позволяющим упорядочивать, анализировать и графически представлять различные виды данных. Но электронные таблицы ориентированы преимущественно на числовые данные и имеют ограниченные возможности для ведения баз данных.
FTP xizmati	**FTP service**	**FTP-сервис**	FTP-сервис – возможность обмена файлами с удаленным компьютером (FTP-сервером). Передача возможна в обе стороны, но в основном PTP серверы используются в качестве хранилища файлов, размещенных там для публичного доступа (считывания).
Internet	**Internet**	**Internet**	Internet (Интернет) – внешняя сеть, сеть сетей. Это возможность общения со всеми компьютерами мира, подключенными к Internet.
	Internet		Internet Mail (Электронная почта) делает то, что и обычная почта, только

Internet-pochta	**Mail**	**Интернет-ПОЧТА**	во много раз быстрее и надежнее.
Internet yangiliklari	**Internet News**	**Интернет-новости**	Internet News (телеконференции) предназначена для общения с группами лиц или группами новостей в отличие от электронной почты, где переписка идет на уровне отдельных лиц.
Jaz	**Jaz**	**Jaz**	Jaz – накопители, появившиеся на отечественном рынке в 1996 году, обладают емкостью 100 Мбайт или 1 Гб. Они также выпускаются как во внутреннем, так и во внешнем исполнении.
WINDOWS XP	**WINDOWS XP**	**WINDOWS XP**	WINDOWS XP – это операционная система, которая содержит новые экраны с понятным интерфейсом, упрощенные меню и многое другое. И в этом ее главное отличие от предыдущих версий графических оболочек
			WINDOWS 3.1 или оболочки NORTON COMMANDER, являющихся только надстройкой над операционной системой MS-DOS.
Word 2003	**Word 2003**	**Word 2003**	Word 2003 для Windows XP – это многофункциональная программа обработки текстов.
World Wild Web (WWW)	**World Wild Web (WWW)**	**World Wild Web (WWW)**	- всемирная паутина Internet.
Zip	**Zip**	**Zip**	Zip представляет собой внешний накопитель со сменными носителями формата 3,5 дюйма и емкостью 25 или 100 Мб. Устройство Zip подключается к параллельному порту любого компьютера и обеспечивает полное заполнение носителя за 5 минут. Один такой носитель заменяет 65-70 дискет. Внутренние накопители имеют более высокую скорость чтения/записи, обеспечивая заполнение 100 Мб за 2 минуты.
Antiviruslar	**Antiviruses**	**Антивирусы**	Антивирусы – программы, предназначенные для обнаружения и уничтожения вирусов.

Arxiv	Arxiv	Архив	Архив – совокупность данных или программ, хранимых в сжатом виде.
arxivlari	**Archivers**	**Архиваторы**	Архиваторы – программы, предназначенные для сжатия выбранных файлов, помещения их в архив и записи полученного архива на дискету. Естественно, что архиватор должен уметь и разархивировать файлы, то есть вернуть их в первоначальное состояние.
Arxivlash	**Archiving**	**Архивация**	Архивация – процесс сжатия файла или группы файлов.
Faylning atributi	**Attribute of the file**	**Атрибут файла**	Атрибут файла – характеристика, определяющая файл.
Ma'lumotlar bazasi	**Database**	**База данных**	База данных – таблица, в строках которой представлены объекты с их характеристиками, а в столбцах – однородные характеристики. Первая строка содержит название полей (характеристик), остальные строки являются записями таблицы.
Bayt	**Bayt**	**Байт**	Байт – единица измерения памяти, равняется 8 битам.
Bit	**Bit**	**Бит**	Бит – самая малая единица измерения информации.
Qulflash	**Locking**	**Блокировка**	Блокировка – запрет на выполнение последующих операций до завершения выполнения текущих операций.
Notepad -	**Notepad -**	**Блокнот –**	Блокнот – программа-редактор для работы с небольшими текстовыми файлами, входит в стандартные программы Windows.
Brauzer	**Brauzer**	**Браузер**	Браузер – универсальное средство передвижения по сетям, с помощью которого Вы получите доступ ко всем ресурсам Интернета, будь то электронная почта, хранилища файлов, Web-странички, базы данных или другие ресурсы.
Video kartalar	**VideoArtar**	**Видеокарты**	Видеокарты – платы, через которые монитор подключается к компьютеру.

Winchester	**Winchester**	**Винчестер**	Винчестер – см. Накопитель на жестком диске.
Virus "hayalet"	**The virus "ghost"**	**Вирус «призрак»**	Вирус «призрак» - вирус, не имеющий ни одного постоянного участка кода (использует при шифровке разные ключи).
Ko'rinmas virus	**The virus "invisible**	**Вирус «невидимый**	Вирус «невидимый» - вирус, перехватывающий обращение DOS к зараженным файлам и областям диска.
Virusni yuklash mumkin	**Virus Bootable**	**Вирус загрузочный**	Вирус загрузочный – вирус, поражающий загрузчик DOS и главную загрузочную запись жесткого диска.
Virus bo'lmagan shaxs	**Virus non-resident**	**Вирус нерезидентный**	Вирус нерезидентный – вирус, который не записывает себя в оперативную память (при запуске выполняется программа-вирус, затем – программа).
Virus rezidenti	**Virus resident**	**Вирус резидентный**	Вирус резидентный – вирус, активизирующийся в оперативной памяти.
Tarmoq virusi	**The network virus**	**Вирус сетевой**	Вирус сетевой – вирус, распространяющийся по компьютерной сети.
Tashqi xotira	**External memory**	**Внешняя память**	Внешняя память – это диски для длительного хранения информации, а также для чтения и записи. Диски делятся на жесткий винчестер и гибкие – дискеты. Последние удобны для создания резервных копий и обмена информацией между пользователями. Внешняя память – это жесткий диск (винчестер, или HDD – hard disk drive), дискеты (floppy disk) и компакт-диск (CD-ROM). Каждому из них соответствует свой дисковод: HDD, FDD, CD-ROM.

Ichki modem	**Internal modem**	**Внутренний модем**	Внутренний модем представляет собой отдельную плату, устанавливаемую внутрь системного блока. Он компактен. Не требует автономного питания и, как правило, дешевле внешнего. Недостаток его заключается в том, что из-за отсутствия световой панели индикаторов уменьшается наглядность при работе с ним.
Moslashuvchan magnit disk	**Flexible magnetic disk**	**Гибкий магнитный диск**	Гибкий магнитный диск – сменный магнитный диск на гибком носителе, используемый в ПЭВМ в качестве внешней памяти прямого доступа.
Gipermatn	**Hypertext**	**Гипертекст**	Гипертекст – это текст, выделенный цветом или подчеркиванием. С таким текстом Вы уже сталкивались, обращаясь неоднократно к Справке, а также при работе со Справочно-правовыми системами. Щелкнув на этом тексте, Вы переходили в другие связанные документы. Всемирная паутина World Wide Web (WWW) состоит сплошь из гипертекстов и, тыкая мышью по ссылкам, можно путешествовать по сети, попадая в самые разные компьютеры, города, страны.
Asosiy menyu	**Main menu**	**Главное меню**	Главное меню содержит все необходимые на начальном этапе работы с компьютером приложения, информацию и вспомогательные программы.
Global tarmoqlar	**Global networks**	**Глобальные сети**	Глобальные сети объединяют как индивидуальных пользователей, так и локальные сети. Примером глобальной сети служит Интернет.
Nest	**Nest**	**Гнездо**	Гнездо – прямоугольник, ограниченный тонким пунктиром и предназначенный для ввода символов, относящихся к данному шаблону.
Chegara	**Frontier**	**Граница**	Граница – предел изменения некоторой величины.
Oyna chegarasi	**Window border**	**Граница окна**	Граница окна – вертикальные и горизонтальные линии, идущие по периметру окна.
Grafik nusxa	**Graphic**	**Графический**	Графический редактор Paint является стандартной программой

muharriri	**Paint Editor**	**редактор Paint**	WINDOWS 98 и поставляется вместе с ним.
Dastur guruhi	**Program Group**	**Группа программ**	Группа программ – набор программ, объединенных по определенному признаку.

ЎЗБЕКИСТОН РЕСПУБЛИКАСИ
ОЛИЙ ВА ЎРТА МАХСУС ТАЪЛИМ ВАЗИРЛИГИ

Рўйхатга олинди:
№ БД – 5130100 – 2.01
2017 йил "18" 08

Олий ва ўрта махсус таълим вазирлиги
2017 йил "24" 08

ДАСТУРЛАШ АСОСЛАРИ
ФАН ДАСТУРИ

Билим сохаси: 100 000 – Гуманитар соҳа
Таълим сохаси: 130000 – Математика
Таълим йўналиши: 5130100 – Математика

Тошкент – 2017

Ўзбекистон Республикаси Олий ва ўрта махсус таълим вазирлигининг 2017 йил "24" 08 даги "605"-сонли буйругининг 2-иловаси билан фан дастури рўйхати тасдикланган.

Фан дастури Олий ва ўрта махсус, касб-хунар таълими йўналишлари бўйича Ўкув-услубий бирлашмалар фаолиятини Мувофиклаштирувчи Кенгашининг 2017 йил "18" 08 даги 4 - сонли баённомаси билан маъкулланган.

Фан дастури Мирзо Улуғбек номидаги Ўзбекистон Миллий университетида ишлаб чикилди.

Тузувчи:

Полатов А.М. - физика-математика фанлари доктори, ЎзМУ проф.в.б.;

Такризчи:

Гулямов Ш.М. - техника фанлар доктори, ТДТУ профессори;

Фан дастури Мирзо Улуғбек номидаги Ўзбекистон Миллий университети Услубий кенгашида кўриб чикилган ва тавсия килинган (2017 йил "14" 07 даги 4 - сонли баённома).

I. Ўқув фанининг долзарблиги ва олий касбий таълимдаги ўрни

"Дастурлаш асослари" фанининг бош мақсади талабаларга қўйилган масалани ечадиган компьютер дастурини тузиш асосларини ўргатишдир. Шу мақсадда дастурлаш тиллари ва муҳитлари ҳақида таянч тушунчалар берилади ва бу тиллардан фойдаланишга ўргатилади.

Фан назарий ва амалий қисмлардан иборат. Назарий қисм информатика ва ҳисоблаш техникаси, алгоритмлар, C/C++ дастурлаш тили, C++Builder объектга йўналтирилган дастурлаш муҳитларида ишлаш бўйича кўрсатмалар бўлимларидан ташкил топган.

Дастурда компьютерда дастурлашга киришнинг назарий асоси бўлган алгоритмларга алоҳида эътибор қаратилган. Бу ерда алгоритмларни тавсифлаш ва кейинчалик компьютерда амалга ошириш учун зарур бўлган бир қатор математик тушунчалар - такрорлаш, ёрдамчи алгоритм, рекурсия, хотира, массив, индекс, функция, параметр ва х.к. киритилиб, турли хил синф масалаларининг алгоритмлари тузилади.

Дастурлаш тили - тузилган алгоритмни компьютер амалга ошириши учун воситадир. Бу ўринда турли мураккабликдаги синтаксис ва семантикага эга бўлган тиллардан фойдаланиш мумкин.

"Дастурлаш асослари" фани йўналишнинг ўқув режасидаги "Эҳтимоллар назарияси", "Сонли усуллар", "Дискрет математика ва математик мантиқ" фанлари билан узвий боғлиқ. Фан мазмуни йўналишнинг ўқув режасидаги "Математик статистика", "Илмий ҳисоблашлар", "Механика", "Оддий дифференциал тенгламалар", "Хусусий ҳосилали дифференциал тенгламалар" фанларини ўзлаштиришда таянч ҳисобланади.

"Дастурлаш асослари" фани умумкасбий фан ҳисобланади ва ўқув йилининг 1-2 семестрларида ўқитилади. Фанни ўқитиш маъруза, амалий машғулот ва мустақил таълим шаклида олиб борилади.

Мазкур дастурга кўра ушбу фан доирасида кўплаб модел масалалар ўрганиладики, бу мазкур фанни чуқур ўрганган ҳар бир бакалавр олган билим ва кўникмаларини ишлаб-чиқаришда, илмий-тадқиқот ишларида, шунингдек, таълим тизимида самарали фойдаланиши имконини беради.

II. Фаннинг мақсад ва вазифалари

Фанни ўқитишдан мақсад – "Математика" йўналишининг бакалавр боскичи талабаларига дастурлаш асосларини етарли даражада ўқитиш, шу билимларга таянган ҳолда компьютер ёрдамида моделлаштиришга келадиган талбикий масалаларнинг дастур таъминотини амалга оширишга ўргатиш ва ихтисослик фанларини ўзлаштиришда таянч билимларга эга бўлиш.

Фаннинг вазифалари – масала ечишнинг алгоритмик асосларини ўргатиш, компьютер ишлашининг тамоили, дастурлаш тилларини синфлаш, компьютерда берилганлар ва буйрукларни тасвирланиши, C++ тилида дастурлаш, объектга йўналтирилган дастурлаш технологиялари, визуал дастурлаш муҳитида ишлаш бу фаннинг асосий вазифалари ҳисобланади.

"Дастурлаш асослари" фанини ўзлаштириш жараёнида амалга ошириладиган масалалар доирасида бакалавр ахборот, уни сақлаш усуллари, қайта ишлаш ва узатиш, ҳисоблаш тизимларининг математик ва дастурий таъминоти, уларни фан соҳаларида, ишлаб чиқариш ва таълимда қўллаш хусусиятлари, компьютерни дастурий таъминоти, дастур турлари ва хусусиятлари, структурали, объектга йўналтирилган ва умумлашган дастурлаш, дастурни оптималлаштириш ва умумлаштириш, дастурлашда модулли тамойилларини қўллаш, компьютер технологиялари ютуқларини замонавий ҳисоблаш тизимларининг математик ва дастурий таъминотида қўллаш, дастурлашнинг тараққиётининг янъаналари ҳақида тасаввурга эга бўлиши, юкори даражадаги дастурлаш тилларини, дастурий таъминотни, дастурлаш технологияларини, татбикий ва ҳисоблаш математикаси масалаларини ечиш алгоритмларини, модулли таҳлил ва модулли дастурлаш асосларини, объектга йўналтирилган ва умумлашган дастурлаш усулларини, самарали дастур ва дастурлар комплексини яратиш усулларини билиши ва улардан фойдалана олиши, талбикий масалаларни ечиш алгоритмини тузиш, математик (компьютер) моделини куриш ва унинг дастурий таъминотини яратиш, структурали, объектга йўналтирилган ва умумлашган дастурлаш парадигмаларини қўллаш асосида иловаларни ярата олиш, дастурлашда, ҳисоблаш техникаси ва дастурий таъминот имкониятларидан самарали фойдаланиш, муаммога ва объектга йўналтирилган тиллардан фойдаланиш, яратилган иловаларни баҳолаш кўникмаларига эга бўлиши керак.

III. Асосий назарий қисм (маъруза машғулотлари)

C++ тили синтаксиси ва унинг лексик асоси. C++ тили дастурининг тузилиши ва шакли. Берилганлар турлари. C++ тилининг таянч турлари. Ўзгарувчилар ва ифодалар. Амаллар: инкремент, декремент, *sizeof*, мантиқий, разрядли, таққослаш. Ўқиш-ёзиш оқимлари (cin, cout).

Операторлар. Шарт операторлари. Такрорлаш операторлари. Бошқарувни узатиш операторлари.

Статик массивлар. Функциялар эълон қилиш ва шикллаш. Локал ва глобал ўзгарувчилар. Рекурсив функциялар. Стандарт кутубхона функциялари. Кўрсаткичлар ва адрес олувчи ўзгарувчилар. Динамик массивлар. Функция ва массивлар.

Сатр ва улар устида амаллар. Тузилмалар ва бирлашмалар. Динамик тузилмалар.

Файл тушунчаси. Матн ва бинар файллар. Файл ва сатр оқимлари. Файллдан ўқиш-ёзиш функциялари. Файл кўрсаткичини бошқариш функциялари.

C++ тилида синфлар. Синфни ва объектларни тавсифлаш. Синф майдонлари ва методлари. Конструктор ва Деструкторлар. Операторларни кайта юклаш. Ворислик.

IV. Амалий машғулотларни ташкил этиш бўйича кўрсатма ва тавсиялар

Амалий машғулотлар ўтказилишидан мақсад дастурлаш бўйича олинган назарий билимларни амалда мустаҳкамлаш ва турли тоифадаги масалаларни ечишга қўллашдан иборат. Амалий машғулотларни бир қисми аудиторияда доскада ечилиши билан ўтказилса, унинг катта қисми бевосита компьютерда амалга оширилиши керак.

Амалий машғулотлар учун тавсия этилаётган тахминий мавзулар рўйхати

1. Саноқ системалари. Бир саноқ системасидан иккинчисига ўтиш.
2. Берилганларни компьютер хотирасида тасвирланиши. Кодлаш.
3. Масалани ечиш алгоритмини тузиш ва унинг кўринишлари.
4. Бутун сонли арифметика масалалари.
5. Ичма-ич жойлашган такрорланувчи жараёнлар, итерацион жараёнлар.
6. Кетма-кетликларни тартиблаш, оддий саралаш масалалари.
7. C++ тили синтаксиси. C++ тилида дастур тузилиши.
8. Visual C++ муҳитида ишлаш.
9. Ўзгарувчилар, амаллар, ифодалар билан ишлаш.
10. Ўқиш-ёзиш оқимларида (cin, cout) киритиш-чиқариш усулларидан фойдаланиш.
11. Шарт операторлари билан ишлаш.
12. Такрорлаш операторлари билан ишлаш.
13. Бошқарувни узатиш операторлари билан ишлаш.
14. Статик массивлар билан ишлаш.
15. Функцияларни эълон ва аниқлаш. Оддий функциялар тузиш.
16. Функция параметрлари ва қайтарувчи қийматлари билан ишлаш.
17. Рекурсив функциялар билан ишлаш.
18. Стандарт кутубхона функцияларидан фойдаланиш.
19. Кўрсаткичлар билан ишлаш.
20. Динамик массивлар билан ишлаш.
21. ASCIIZ сатрлар ва улар устида амаллар.
22. string туридаги сатрлар ва улар устида амаллар.
23. Тузилмалар билан ишлаш. Бирлашмалар билан ишлаш.
24. Матн файллар ва улар устида амаллар.
25. Бинар файллар ва улар устида амаллар.
26. Файл структуралар билан ишлаш, саралаш ва кидириш алгоритмлари, ифодаларини ҳисоблаш.
27. Динамик тузилмалар билан ишлаш.
28. Синф объектларини яратиш ва улар устида амаллар.
29. Операторларни қайта юклаш.

Изоҳ: Амалий машғулот соатлари ҳажмларидан келиб чиккан холда ишчи дастурда мазкур мавзулар ичидан амалий машғулот мавзулари шакллантирилади.

V. Мустақил таълимни ташкил этишнинг шакли ва мазмуни

Талаба мустақил таълимнинг асосий мақсади – ўқитувчининг раҳбарлиги ва назоратида муайян ўқув ишларини мустақил равишда бажариш учун билим ва кўникмаларини шакллантириш ва ривожлантириш.

Мустақил ишларни бажариш жараёнида талабалар қуйидаги ишларни бажарадилар:

- дарслик ва ўқув қўлланмалар асосида фан мавзулари бўйича назарий тайёргарлик кўриш, амалий ва лаборатория машғулотларига тайёрланиш;
- тарқатма материаллар бўйича маърузаларни чуқур ўзлаштириш;
- фан мазмунида кўрсатилмаган дастурлаш тиллари ва муҳитлари билан танишиш ва киёсий таҳлил килиш;
- масофавий таълим орқали дастурлаш билан турдош фанлар бўйича ўқув курсларида қатнашиш ва мос сертификатларга эга бўлиш тавсия қилинади.

Талаба мустақил ишини ташкил этишда қуйидаги шакллардан фойдаланади:

- берилган мавзулар бўйича ахборот (реферат) тайёрлаш;
- назарий билимларни амалиётда қўллаш;
- макет, модел ва намуналар яратиш;
- илмий мақола, анжуманга маъруза тайёрлаш ва ҳ.к.

Тавсия этилаётган мустақил ишларнинг мавзулари

1. C ва C++ тили синтаксислари.
2. C++ ва бошқа тилларда (Pascal, C#) модулли дастурлаш.
3. Стандарт кутубхона. Оқим синфлари.
4. Алгоритмлар кутубхонаси. Сонли ҳисоблаш функцияларининг кутубхонаси.
5. .NET технологияси, C# тилини хусусиятлари.
6. 8,16,32 разрядли процессорлар.
7. Турли муҳитларда яратилган дастур объектларини боғлаш.
8. C++ тили учун консол режимидаги дастурлаш муҳитлари.
9. Динамик структураларни қайта ишлаш алгоритмлари.
10. Саралаш ва излашнинг самарали алгоритмлари.
11. C++ тили стандартлари.
12. C++ тили асосида яратилган тиллар (C# ва Java тиллари).

Изоҳ: Мустақил таълим соатлари ҳажмларидан келиб чиккан ҳолда ишчи дастурда мазкур мавзулар ичидан мустақил таълим мавзулари шакллантирилади.

Фойдаланиладиган адабиётлар рўйхати

Асосий адабиётлар

1. Bjarne Stroustrup. The C++ Programming Language (3th Edition). Addison-Wesley, 1997.

2. D.S. Malik. C++ Programming: From Problem Analysis to Program Design. Fifth Edition. Course Technology, 2011.
3. Мадрахимов Ш.Ф., Гайназаров С.М. С++ тилида дастурлаш асослари// Тошкент, ЎзМУ, 2009, 196 бет.
4. Madraximov Sh.F., Ikramov A.M., Babajanov M.R. C++ tilida programmalash bo'yicha masalalar to'plami. O'quv qo'llanma // Toshkent, O'zbekiston Milliy Universiteti, "Universitet" nashriyoti, 2014. - 160 bet.

Қўшимча адабиётлар

5. Sh. Mirziyoyev Buyuk kelajagimizni mard va olijanob xalqimiz bilan birga quramiz. Toshkent "O'zbekiston" 2017. 488 b.
6. Sh. Mirziyoyev. Qonun ustuvorligi va inson manfaatlarini ta'minlash – yurt taraqqiyoti va xalq farovonligining garovi. O'zbekiston Respublikasi Konstitutsiyasi qabul qilinganligining 24 yilligiga bag'ishlangan tantanali marosimdagi ma'ruza. 2016-yil 7-dekabr. Toshkent - "O'zbekiston" - 2017. 32 b.
7. Мирзиёев Ш.М. Танқидий таҳлил, қатъий тартиб-интизом ва шахсий жавобгарлик – ҳар бир раҳбар фаолиятининг кундалик қоидаси бўлиши керак. Ўзбекистон Республикаси Вазирлар Маҳкамасининг 2016 йил якунлари ва 2017 йил истиқболларига бағишланган мажлисидаги Ўзбекистон Республикаси Президентининг нутқи. // Халқ сўзи газетаси. 2017 йил 16 январь, №11.
8. Ш.Мирзиёев. Эркин ва фаровон, демократик Ўзбекистон давлатини биргаликда барпо этамиз. Ўзбекистон Республикаси Президенти лавозимига киришиш тантанали маросимига бағишланган Олий Мажлис палаталарининг қўшма мажлисидаги нутқи. Тошкент – "Ўзбекистон". 2016. 56 б.
9. ЎзР «Ахборотлаштириш тўғрисида» 2003 йил 11 декабрдаги Қонуни.
10. Bjarne Stroustrup. The C++ Programming Language (4th Edition). Addison-Wesley, 2013. 1363 page.
11. Bjarne Stroustrup. Programming: Principles and Practice using C++ (Second Edition)" Addison-Wesley, 2014, 1305 page.
12. Павловская Т.А. С++. Программирование на языке высокого уровня – СПб.: Питер. 2005.- 461 с.
13. Walter Savitch. Absolute C++, 5th edition. Addison-Wesley/Pearson, 2012. 984 page.
14. Павловская Т.С. Щупак Ю.С. С/С++. Структурное программирование. Практикум.-СПб.: Питер,2002-240с
15. Глушаков С.В., Коваль А.В., Смирнов С.В. Язык программирования С++: Учебный курс.- Харьков: Фолио; М.: ООО «Издательство АСТ», 2001.- 500с.
16. Культин Н.Б. С++Builder в задачах и примерах.-СПб.: БХВ-Петербург, 2005.-336с.

17. Абрамов С.А., Гнездилова Капустина **В.Н. и др. Задачи по программированию.** - М.: Наука, 1988.

Электрон манбалар

1. http://cppstudio.com – С++ тилида программалаш бўйича намуналар изохлари билан келтирилган
2. http://cplusplus.com – С++ тилида мавжуд конструкциялар таърифи, ишлатиш намуналари билан келтирилган.
3. http://www.compteacher.ru/programming – дастурлаш бўйича видео дарсликлар мавжуд.
4. http://www.intuit.ru – интернет университет, дастурлаш бўйича ёзма ва видео маърузалар ўқиш, тест синовларидан ўтиш ва сертификат олиш имконияти мавжуд.
5. http://www.ziyonet.uz – дастурлаш асослари бўйича рефератлар топиш мумкин.

ЎЗБЕКИСТОН РЕСПУБЛИКАСИ

ОЛИЙ ВА ЎРТА МАХСУС ТАЪЛИМ ВАЗИРЛИГИ

ГУЛИСТОН ДАВЛАТ УНИВЕРСИТЕТИ

АХБОРОТ ТЕХНОЛОГИЯЛАРИ

КАФЕДРАСИ

“ТАСДИҚЛАЙМАН”

Ўқув ишлари бўйича проректор

Н.Р. Баракаев

_____

«___» _____2017 й.

Дастурлаш асослари

фани бўйича

ишчи ўқув дастури

Билим соҳаси: Гуманитар

Таълим соҳаси: 130000 – Математика

Таълим йўналиши: 5130100-Математика

Умумий ўқув соати – 366

Шу жумладан:

Маъруза – 94

Амалиёт машғулотлари – 122

Мустақил таълим соати – 150

ГУЛИСТОН – 2017 й.

Фаннинг ишчи ўқув дастури намунавий ўқув дастури ва ўқув режасига мувофиқ ишлаб чиқилди.

Тузувчи: Қаландаров А.А. – ГулДУ “Ахборот технологиялари” кафедраси катта ўқитувчиси _____ (имзо)

Тақризчи: Тоштемиров Д.Э. – ГулДУ “Ахборот технологиялари” кафедраси доценти _____ (имзо)

Фаннинг ишчи ўқув дастури “Ахборот технологиялари” кафедрасининг 2017 йил “___” _____ даги ___ - сонли мажлисида кўриб чиқилиб, факультет Илмий-услубий Кенгашида кўриб чиқиш учун тавсия қилинди.

Кафедра мудири: **доц. Абдурахимов Д.Б.**

Фаннинг ишчи ўқув дастури “Физика-математика” факультети Илмий-услубий Кенгашининг 2017 йил “___” _____ даги “____” - сонли мажлисида тасдиқланди.

Факультет Илмий-услубий

Кенгаши раиси: доц. Ш. Аширов

I. Ўқув фанининг долзарблиги ва олий касбий таълимдаги ўрни

“Дастурлаш асослари” фанининг бош мақсади талабаларга қўйилган масалани ечадиган компьютер дастурини тузиш асосларини ўргатишдир. Шу мақсадда дастурлаш тиллари ва муҳитлари ҳақида таянч тушунчалар берилади ва бу тиллардан фойдаланишга ўргатилади.

Фан назарий ва амалий қисмлардан иборат. Назарий қисм информатика ва ҳисоблаш техникаси, алгоритмлар, C/C++ дастурлаш тили, C++Builder объектга йўналтирилган дастурлаш муҳитларида ишлаш бўйича кўрсатмалар бўлимларидан ташкил топган.

Дастурда компьютерда дастурлашга киришнинг назарий асоси бўлган алгоритмларга алоҳида эътибор қаратилган. Бу ерда алгоритмларни тавсифлаш ва кейинчалик компьютерда амалга ошириш учун зарур бўлган бир қатор математик тушунчалар – такрорлаш, ёрдамчи алгоритм, рекурсия, хотира, массив, индекс, функция, параметр ва ҳ.к. киритилиб, турли хил синф масалаларининг алгоритмлари тузилади.

Дастурлаш тили – тузилган алгоритмни компьютер амалга ошириши учун воситадир. Бу ўринда турли мураккабликдаги синтаксис ва семантикага эга бўлган тиллардан фойдаланиш мумкин.

“Дастурлаш асослари” фани йўналишнинг ўқув режасидаги “Эхтимоллар назарияси”, “Сонли усуллар”, “Дискрет математика ва математик мантиқ” фанлари билан узвий боғлиқ. Фан мазмуни йўналишнинг ўқув режасидаги “Математик статистика”, “Илмий ҳисоблашлар”, “Механика”, “Оддий дифференциал тенгламалар”, “Хусусий ҳосилали дифференциал тенгламалар” фанларини ўзлаштиришда таянч ҳисобланади.

“Дастурлаш асослари" фани умумкасбий фан ҳисобланади ва ўкув йилининг 1-2 семестрларида уқитилади. Фанни ўқитиш маъруза, амалий машғулот ва мустақил таълим шаклида олиб борилади.

Мазкур дастурга кўра ушбу фан доирасида кўплаб модел масалалар ўрганиладики, бу мазкур фанни чуқур ўрганган ҳар бир бакалавр олган билим ва кўникмаларини ишлаб-чиқаришда, илмий-тадқиқот ишларида, шунингдек, таълим тизимида самарали фойдаланиши имконини беради.

II. Фаннинг мақсад ва вазифалари

Фанни ўқитишдан мақсад – “Математика” йўналишининг бакалавр босқичи талабаларига дастурлаш асосларини етарли даражада ўқитиш, шу билимларга таянган ҳолда компьютер ёрдамида моделлаштиришга келадиган тадбиқий масалаларнинг дастур таъминотини амалга оширишга ўргатиш ва ихтисослик фанларини ўзлаштиришда таянч билимларга эга бўлиш.

Фаннинг вазифалари – масала ечишнинг алгоритмик асосларини ўрганиш, компьютер ишлашининг тамоили, дастурлаш тилларини синфлаш, компьютерда берилганлар ва буйруқларни тасвирланиши, C++ тилида дастурлаш, объектга йўналтирилган дастурлаш технологиялари, визуал дастурлаш муҳитида ишлаш бу фаннинг асосий вазифалари ҳисобланади.

“Дастурлаш асослари" фанини ўзлаштириш жараёнида амалга ошириладиган масалалар доирасида бакалавр ахборот, уни сақлаш усуллари, қайта ишлаш ва

узатиш, ҳисоблаш тизимларининг математик ва дастурий таъминоти, уларни фан соҳаларида, ишлаб чиқариш ва таълимда қўллаш хусусиятлари, компьютерни дастурий таъминоти, дастур турлари ва хусусиятлари, структурали, объектга йўналтирилган ва умумлашган дастурлаш, дастурни оптималлаштириш ва умумлаштириш, дастурлашда модулли тамойилларини қўллаш, компьютер технологиялари ютуқларини замонавий ҳисоблаш тизимларининг математик ва дастурий таъминотида қўллаш, дастурлашнинг тараққиётинииг анъаналари ҳақида тасаввурга эга бўлиши, юқори даражадаги дастурлаш тилларини, дастурий таъминотни, дастурлаш технологияларини, татбиқий ва ҳисоблаш метематикаси масалаларини ечиш алгоритмларини, модулли таҳлил ва модулли дастурлаш асосларини, объектга йўналтирилган ва умумлашган дастурлаш усулларини, самарали дастур ва дастурлар комплексини яратиш усулларини билиши ва улардан фойдалана олиши, тадбиқий масалаларни ечиш алгоритмини тузиш, математик (компьютер) моделини қуриш ва унинг дастурий таъминотини яратиш, структурали, объектга йўналтирилган ва умумлашган дастурлаш парадигмаларини қўллаш асосида иловаларни ярата олиш, дастурлашда, ҳисоблаш техникаси ва дастурий таъминот имкониятларидан самарали фойдаланиш, муаммога ва объектга йўналтирилган тиллардан фойдаланиш, яратилган иловаларни баҳолаш куникмаларига эга бўлиши керак.

Фанда ўтиладиган мавзулар ва улар бўйича машғулот турларига ажратилган соатларнинг тақсимоти

№	Фаннинг бўлими ва мавзуси, маъруза мазмуни	Соатлар			
		Жами	Маъруза	Амалий машғулот	Мустақил иш
1	Амалий масалаларни ечиш алгоритмлари	6	2	2	2
2	Алгоритмларни блок-схема кўринишида ифодаланиши	10	2	4	4
3	С++ тили синтаксиси ва унинг лексик асоси	8	2	2	4
4	С++ тили дастурининг тузилиши ва шакли	6	2	2	2
5	Берилганлар турлари. С++ тилининг таянч турлари.	8	2	2	4
6	Берилганларни компьютер хотирасида тасвирланиши	6	2	2	2
7	Ўзгарувчилар ва ифодалар	8	2	2	4
8	Амаллар: инкремент, декремент, sizeof, мантиқий, разрядли, таққослаш	10	2	4	4
9	Амалларнинг устунликлари ва бажарилиш	6	2	2	2

	йўналишлари				
10	Ўқиш-ёзиш оқимлари (cin, cout)	8	2	2	4
11	Операторлар	6	2	2	2
12	Шарт операторлари	10	2	4	4
	ОН				
13	Танлаш оператори	8	2	2	4
14	for такрорлаш операторлари	8	2	2	4
15	while, do-while такрорлаш операторлари	10	2	4	4
16	Бошқарувни узатиш операторлари	8	2	2	4
17	Структурали дастурлаш	6	2	2	2
18	Бир ўлчовли статик массивлар	8	2	2	4
19	Бир ўлчовли статик массивлар устида амаллар	8	2	4	2
20	Кўп ўлчовли статик массивлар	8	2	2	4
21	Кўп ўлчовли статик массивлар устида амаллар	8	2	4	2
22	Функциялар эълон қилиш ва аниқлаш	6	2	2	2
23	Локал ва глобал ўзгарувчилар	6	2	2	2
24	Функция параметрларида статик массивлардан фойдаланиш	7	2	2	3
	ОН				
	ЯН				
25	Рекурсив функциялар	8	2	2	4
26	Стандарт кутубхона функциялари	10	2	4	4
27	Кўрсаткичлар ва адрес олувчи ўзгарувчилар	10	2	4	4
28	Динамик массивлар	14	4	4	6
29	Функция ва массивлар	10	2	4	4
30	Сатр ва улар устида амаллар	20	6	6	8
31	Тузилмалар ва бирлашмалар	10	2	4	4

32	Динамик тузилмалар	8	2	2	4
	ОН				
33	Файл тушунчаси	12	4	4	4
34	Матн ва бинар файллар	12	2	4	6
35	Файл ва сатр оқимлари	8	2	2	4
36	Файлдан ўқиш-ёзиш функциялари	10	2	4	4
37	Файл кўрсаткичини бошқариш функциялари	6	2	2	2
38	C++ тилида синфлар	9	2	4	3
39	Синфни ва объектларни тавсифлаш	8	2	2	4
40	Синф майдонлари ва методлари	6	2	2	2
41	Конструктор ва Деструкторлар	10	2	4	4
42	Операторларни қайта юклаш. Ворислик	12	4	4	4
	ОН				
	ЯН				
Жами		**366**	**94**	**122**	**150**

Амалий машғулотларини ташкил этиш бўйича кўрсатма ва тавсиялар

Амалий машғулотлар ўтказилишидан мақсад дастурлаш бўйича олинган назарий билимларни амалда мустаҳкамлаш ва турли тоифадаги масалаларни ечишга қўллашдан иборат. Амалий машғулотларни бир қисми аудиторияда доскада ечилиши билан ўтказилса, унинг катта қисми бевосита компьютерда амалга оширилиши керак.

Мустақил таълимни ташкил этишнинг шакли ва мазмуни

Талабалар мустақил таълимнинг асосий мақсади-ўқитувчининг раҳбарлиги ва назоратида муайян ўқув ишларини мустақил равишда бажариш учун билим ва кўникмаларини шакллантириш ва ривожлантириш.

Мустақил ишларни бажариш жараёнида талабалар қуйидаги ишларни бажарадилар:

- дарслик ва ўқув қўлланмалар асосида фан мавзулари бўйича назарий тайёргарлик кўриш, амалий ва лаборатория машғулотларига тайёрланиш;
- тарқатма материаллар бўйича маърузаларни чуқур ўзлаштириш;

- фан мазмунида кўрсатилмаган дастурлаш тиллари ва муҳитлари билан танишиш ва қиёсий таҳлил қилиш;
- масофавий таълим орқали дастурлаш билан турдош фанлар бўйича ўқув курсларида қатнашиш ва мос сертификатларга эга бўлиш тавсия қилинади.
  Талаба мустақил ишини ташкил этишда қуйидаги шакллардан фойдаланади:

- берилган мавзулар бўйича ахборот (реферат) тайёрлаш;
- назарий билимларни амалиётда қўллаш;
- макет, модел ва наъмуналар яратиш;
- илмий мақола, анжуманга маъруза тайёрлаш ва ҳ.к.

Рейтинг баҳолаш тизими:

КУЗГИ СЕМЕСТР

№			Сентябр				Октябр					Ноябр				Декабр				Январь, Февраль						Жами
			4-9	11-16	18-23	25-30	2-7	9-14	16-21	23-28	30-4	6-11	13-18	20-25	27-2	4-9	11-16	18-23	25-30	1-6	8-13	15-20	22-27	29-3		
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
1	ОБ	Лабор.													5					Т	Т	5	**Я**	**Я**		**5**
		Амал.													5					Т	Т	5	**Я**	**Я**		**5**
		Муст. таълим													5					Т	Т	5	**Я**	**Я**		**5**
		Ёзма иш													5					Т	Т	5	**Я**	**Я**		**5**
		Оғзаки сўров													5					Т	Т	5				**5**
2	ЯБ																			Т	Т		5			**5**

Балл	5	4	3	2

БАҲОРГИ СЕМЕСТР

№			Февраль			Март					Апрел				Май				Июн				Жами:
			5-10	12-17	19-24	26-3	5-10	12-17	19-24	26-31	2-7	9-14	16-21	23-28	30-5	7-12	14-19	21-26	28-2	4-9	11-16	18-23	
			24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	
1	ОБ	Лабор.										5								**5**			**5**
		Амал.										5								**5**			**5**

		Муст. таълим										5								5		5
		Тест										5								5		5
		Ёзма иш										5								5		5
2	ЯБ																				5	

Балл	5	4	3	2

Ёзма ишларни баҳолаш мезонлари:

Баҳолаш мезонлари	Баҳолаш бали
Етарли назарий ва амалий билимга ега. Берилган саволларга тўлиқ жавоб беради. Масаланинг моҳиятига тўлиқ тушунади. Хатоларга йўл қўймайди. Назарий саволларга ҳаётий мисоллар орқали изоҳлар келтиради. Саволларга жавобларни намунали расмийлаштиради.	5
Етарли назарий билимга ега. Топшириқларни ечган. Берилган саволларга етарли жавоб беради. Масаланинг моҳиятини тушунади. Саволларга жавобларни намунали расмийлаштиради.	4
Берилган топшириқларни ечишга ҳаракат қилади. Берилган саволларга жавоб беришга ҳаракат қилади. Масаланинг моҳиятини чала тушунган.	3
Берилган топшириқларга нотўғри жавоб берса, ёзма иш бўйича масала, мисол ва саволларига жавоб бермаса, билим даражаси қониқарсиз баҳоланади.	2

Фойдаланиладиган адабиётлар рўйхати

Асосий адабиётлар

1. Bjarne Stroustrup. The C++ Programming Language (3th Edition). Addison-Wesley, 1997.
2. D.S. Malik. C++ Programming: From Problem Analysis to Program Design. Fifth Edition. Course Technology, 2011.
3. Мадрахимов Ш.Ф., Гайназаров С.М. С++ тилида дастурлаш асослари// Тошкент, ЎзМУ, 2009, 196 бет.
4. Madraximov Sh.F., Ikramov A.M., Babajanov M.R. C++ tilida programmalash bo'yicha masalalar to'plami. O'quv qo'llanma // Toshkent, O'zbekiston Milliy Universiteti, "Universitet" nashriyoti, 2014. - 160 bet.

Қўшимча адабиётлар

1. Ш.Мирзиёев. Буюк келажагимизни мард ва олийжаноб халқимиз билан бирга қурамиз. Тошкент. "Ўзбекистон" 2017. 488 б.
2. Ш.Мирзиёев. Қонун устуворлиги ва инсон манфаатларини таъминла-юрт тараққиёти ва халқ фаровонлигининг гарови. Ўзбекистон Республикаси Конституцияси қабул қилинганлигининг 24 йиллигига бағишланган тантанали маросимидаги маъруза. 2016-йил 7-декабрь. Тошкент-"Ўзбекистон" – 2017. 32 б.
3. Мрзиёев Ш.М. Танқидий таҳлил, қатъий тартиб-интизом ва шахсий жавобгарлик-ҳар бир раҳбар фаолиятининг кундалик қоидаси бўлиши керак. Ўзбекистон Республикаси Вазирлар Маҳкамасининг 2016 йил якунлари ва 2017 йил истиқболларига бағишланган мажлисдаги Ўзбекистон Республикаси Президентининг нутқи. // Халқ сўзи газетаси. 2017 йил 16 январь, №11.
4. Ш.Мирзиёев. Эркин ва фаровон, демократик Ўзбекистон давлатини биргаликда барпо этамиз. Ўзбекистон Республикаси Президенти лавозимига киришиш тантанали маросимига бағишланган Олий Мажлис палаталарининг қўшма мажлисидаги нутқи. Тошкент –"Ўзбекистон", 2016. 56 б.
5. ЎзР "Ахборотлаштириш тўғрисида" 2003 йил 11 декабрдаги Қонуни.
6. Bjarne Stroustrup. The C++ Programming Language (4th Edition). Addison-Wesley, 2013. 1363 page.
7. Bjarne Stroustrup. Programming: Principles and Practice using C++ (Second Edition)" Addison-Wesley, 2014, 1305 page.
8. Павловская Т.А. С++. Программирование на языке высокого уровня – СПб.: Питер. 2005.- 461 с.
9. Walter Savitch. Absolute C++, 5th edition. Addison-Wesley/Pearson, 2012. 984 page.
10. Walter Savitch. Problem Solving with C++, 9th edition. Addison-Wesley/Pearson, 2015. 1088 page.
11. Павловская Т.С. Щупак Ю.С. С/С++. Структурное программирование. Практикум.-СПб.: Питер,2002-240с
12. Глушаков С.В., Коваль А.В., Смирнов С.В. Язык программирования С++: Учебный курс.- Харьков: Фолио; М.: ООО «Издательство АСТ», 2001.-500с.
13. Культин Н.Б. С++Builder в задачах и примерах.-СПб.: БХВ-Петербург, 2005.-336с.
14. Абрамов С.А., Гнездилова Капустина Е.Н. и др. Задачи по программированию. - М.: Наука, 1988.

Электрон манбалар

1. http://cppstudio.com – C++ тилида программалаш бўйича намуналар изохлари билан келтирилган
2. http://cplusplus.com – C++ тилида мавжуд конструкциялар таърифи, ишлатиш намуналари билан келтирилган.
3. http://www.compteacher.ru/programming – дастурлаш бўйича видео дарсликлар мавжуд.
4. http://www.intuit.ru – интернет университет, дастурлаш бўйича ёзма ва видео маърузалар ўқиш, тест синовларидан ўтиш ва сертификат олиш имконияти мавжуд.
5. http://www.ziyonet.uz – дастурлаш асослари бўйича рефератлар топиш мумкин.

ТЕСТЫ

Найдите слово которым нельзя назвать переменную
int
a
A7
Pi
Найдите размер типа int для 32 рязрядной системы
4 bayt
2 bayt
6 bayt
8 bayt
Как пишется |X| для вещественных чисел
fabs(x)
abs(x)
labs(x)
absf(x)
Какие переменные объявляются ключевым словом “bool”
Логические
Целые
Вещественные
Символьные
Каким ключевым словом объявляются переменные символьного типа
Char
Float
Int
Bool
Каким ключевым словом объявляются переменные целого типа
Int
Float
Char
Bool

Диапозоп переменных типа char
0..255
-128..127
0..32
-32768..32767
Найдите размер переменной типа double?
8 bayt
4 bayt
2 bayt
1 bayt
Каким символом разделяется целая и дробная части вещественных чисел
(‘.’)
(‘,’)
(‘;’)
(‘/’)
Как производится считование значений переменных
cin>> <переменная>
cout<< <переменная>
cin<< < переменная >
cout>>< переменная >
Как производится вывод на экран
cout<< < переменная >
cin>> < переменная >
cout>> < переменная >
cin<< < переменная >
Логическое умножение «и»
&&
||
!
%
Логическое отрицание «не»
!
||
&&
^

enum Hafta {dush=5, sesh, chorsh=0, paysh, juma, shanba=paysh-1, yaksh} чему равно значение переменной «yaksh»?
1
6
7
0
Какой размер памяти отделяется переменной вещественного типа
4 bayt
3 bayt
2 bayt
1 bayt
Определити порядок выполнения операций? a / b + a % b * c
/, %, * , +
%, /, * ,+
%, / , + , *
/ , + , %, *
Вычислите значение 24 / (3*4) – 24/3/4 + 24/ 3*4
32
4
12
16
int n,m = 2; n = 1; m+ = n + +; *cout* << *n* << " " << *m*; что распечатается на экране?
2 3
2 6
3 7
3 5
Какие значения могут принимать переменные логического типа ?
0(false) или 1 (true)
0 (false)
True
0(true) или 1(false)
Над какими числами можно производить огругление?
Вещественные
Целые
Логические
Любые
Какой тип занимает больше памяти
long double
Double
int
float

int n=7; bool k; k=n%2; cout << k; что распечатается на экране?
1
True
False
0
что распечатается на экране? int n=123,a ; a=n%100; cout << a;
23
12
3
0.3
что распечатается на экране? int a=17; cout << a%10<<a/10;
71
17
18
16
sin2x + 2cosy как пишется на с++?
$\sin(2*x)+2*\cos(y)$
$\sin(2x)+2\cos(y)$
$\sin 2*x+2*\cos y$
$\sin*2*x+2*\cos*y$
Если стороны треугольника равны a, b , c как определяется возможность построить треугольник?
t = (a<b+c) && (b<a+c)&&(c<a+b)
t = (a<b+c) || (b<a+c)
t = (a<b+c) && (b>a+c)&&(c<a+b)
t = (a<b+c) || (b<a+c)||(c<a+b)
Определите оператор цыкла с предусловием
while
for ( )
do… while
if ( )
Чему равно значение выражения на С++? $12/5+125\%(4+3*7)/2$
2
0

7
1
что распечатается на экране? int s=1; for(int i= 0;i<=20; i ++) {s+=i; if(s == 10) break;} cout << s ;
211
191
10
1
Определите постфикс
x=y++
x=++y
x+=y
x=x+y
что распечатается на экране? int a=6789; cout<<a/100;
67
6789
89
678
что распечатается на экране? int s=2468; cout<<(s%1000)/10
46
68
468
6
Найдите диапозон значений short int ?
-32768...32767
0..65535
0..32767
-2147483648.. 2147483647
Каким символом отделяются операторы?
;
“ ” пробел
:
,
До каких пор будит продолжатся тело цыкла while ?
До получения логического выражения значение лож
Бесконечно

Сколько угодно
Такого цыкла не существует
Оператор break это - …
Осуществляет выход из тела цыкла
Осуществляет перехон на следующую итерацию цыкла
Такого оператора не существует
Оператор цыкла
Опреатор continue это - …
Осуществляет перехон на следующую итерацию цыкла
Осуществляет выход из тела цыкла
Такого оператора не существует
Оператор цыкла
Как производится табуляция?
\t
\n
\b
\v
Как производится переход на новую строку?
\n
\t
\f
\v
Определите правильно написанный оператор цыкла for
for(<выражение1>;<выражение2>; < выражение 3>)<Блок операторов>
for(<выражение >)оператор>
for(<выражение 1>,< выражение 2>, < выражение 3>)<оператор>
for(<выражение 1>; < выражение 3>)<оператор
Метка это -…
Идентификатор после которого ставится двоеточие(‘:’)
Идентификатор перед которого ставится двоеточие(‘:’)
Идентификатор после которого ставится точка(‘.’)
Идентификатор после которого ставится решотка (#)

Сколько раз будет повторяться тело цыкла? for(int i=1; i<=10; i+=3) i--;
5
10
Безконечно
0
int $b = 23$; a = 30; $a+ = a + b$; определите значение a
83
54
55
80
int $b = 54$; $a = 30$; a$- = a + b$; определите значение a
-54
84
-50
80
что распечатается на экране? int s=0,i; for(i=1;i<10;i++) s+=i; cout<<s;
45
55
10
1
что распечатается на экране? int a=10, b=23; switch(a){ case 1: c=a+b; break; case 2: c = a * b, break; default: c = (a + b) * b, break; } cout<<c;
759
33
230
Ошибка 3542
что распечатается на экране? int n=11; bool m; m=!(n%2==1); cout << m;
0
1
6.5
6
что распечатается на экране? int a = 20; bool t = false; if(t) a=200; else a =100; cout <<a;
100
20
300
150
что распечатается на экране? int a; bool t=true, T=true; if(t && T) a = 50; else a=200; cout <<a;

50
100
200
150
что распечатается на экране? int a; bool t=true, T=false; if(t && T) a=100; else a=200; cout <<a;
200
50
150
300
что распечатается на экране? int a; bool t=false, T=false; if(t && T) a=100; else a=200; cout <<a;
200
100
400
500
что распечатается на экране? int a; bool t=false, T=true; if(t ||T) a=500; else a=800; cout <<a;
500
1000
800
1300
что распечатается на экране? int i=0, s=0; bool t=true, T=true; for(;t&&T;) {i+=20; s += i; if (100 < i) t = !T; } cout<<s;
420
530
100
890
Можноли задать значения a= 5.0 и b=6.2 ? int a; float b; cin>>a; cin>>b; cout<<a<<b;
Нет
Да
В некоторых случаях да
Вывод а
Если X = false чему равно !X = ?
True

False
0
false или true
Если X = true чему равно !X = ?
False
True
1
2
С помошью какого оператора можно выйти из бесконечного цыкла?
break;
continue;
return;
Switch
Как производится обращение на одномерный массив
<имя массива>[<индекс>]
< имя массива >{< индекс >}
< индекс >[< имя массива >]
< имя массива >(< индекс >)
Блок операторов это-…
Операторы между ‘{’ и ‘}’
Опреаторы между ‘(’ и ‘)’
Операторы между‘[’ и ‘]’
Нет правильного ответа
Найдите правильно объявленную функцию
int funk(int);
funk(int)
function funk(int);
funk(int) int;
Найдите оператор цыкла с пост условием
do-while
while
for
If
Найдите оператор без условного перехода

Goto
go
go … to
continue
Как определяется обсалютная величена целых чисел?
abs(x)
fabs(x)
exp(x)
ceil(x)
что распечатается на экране? int a=234; for(;a>10;) a/=10; cout<<a;
2
3
4
0
что распечатается на экране? int i; for(i=1;i<10;i++) ; cout<<i;
10
Цыфры от 1 до 9
Цыфры от 1 до 10
1
что распечатается на экране? int i,s=0; for(i=1;i<10;i++){s+=i; i++; } cout<<s;
25
45
1
10
Как определяется квадратный корень
Sqrt
Pow
Fmod
Floor
Как записывается x^y

pow(x,y)
pow(y,x)
fmod(x,y)
hypot(x,y)
int Summa = 0; for (int i = 10, i <= 20, Summa + = i + +); определите значение переменной Summa
Сумма чисел от 10 до 20
Сумма чисел 10 и 20i
0
Невозможно определить
Найдите правильно написанный префикс
x=--i
x+=i
x=i++
x=i-x
что распечатается на экране? int a=10, b; b=a%10; cout<<b;
0
1
10
100
что распечатается на экране? int a=10, b; if(a%2==0) $b = a * a$; else b=-a; cout<<b;
100
-10
10
0
что распечатается на экране? int x=1; x+=5; cout<<x;
6
5
0
Ошибка
что распечатается на экране? int a=4, b=8, y; y=a>b?a:b; cout<<y;
8
5
12
0.5
что распечатается на экране? int i, s=0; for(i = 0; i < 6; i++) { $s* = i$; }cout << s ;

0
120
720
24
что распечатается на экране? int i, s=1; for(i = 0; i < 6; i++) { $s* = i$; }cout << s;
0
720
24
120
что распечатается на экране? int s=1; for(int i = 0; i <= 6; i += 2) s+=i; cout<<s;
13
12
22
21
что распечатается на экране? int s=0; for(int i=0; i<=20; i++) {s+=i; if(s == 10) break;} cout <<s;
10
15
1
0
что распечатается на экране? int s=1; for(int i=0; i<6; i++) { $s* = (i+1); i++$; } s++;cout << s;
16
15
0
1
что распечатается на экране? float a=1234; cout<<a/100;
12.34
12
34
1.234

что распечатается на экране? int s=0;for (int i = 0; i <= 5; i++) if (i % 2) s+= i; cout<<s;
9
19
10
100
что распечатается на экране? int n=20, s=0; for (int i = 1; i <= 20; i++) if (n%i== 0) s += i; cout << s;
42
100
55
45
что распечатается на экране? int s=0, a=0, b=12; do{a+=2; b -=2; s+=a+b;} while(a!=b); cout << s;
36
45
50
24
Если ввести числа 1 2 3 что выведит на экран следующая программа? int a,b; cin>>a>>b>>a; cout<<a<<' '<<b<<' '<<a;
3 2 3
3 3 3
1 3 1
2 1 3
Как называется множество переменных одного типа?
Массив
Множество
Строка
Переменная
Чтобы использовать в программе математические функции надо объявить …
math.h
conio.h
vcl.h
string.h
С помощью какого служебного слова объявляется константы?
Const
Var

Type
Arg
Как пишется квадратный корень?
sqrt
Sqr
Kop
Exp
Как можно записать tg(x) ?
sin(x)/cos(x)
sin(y)/cos(y)
sin(x)/cos(y)
sinx/cosx
Как объявляются переменные вещественного типа?
Float
Real
Boolean
Int
Как объявляются переменные целого типа?
Int
Float
Boolean
Char
Как объявляются переменные переменные логическаго типа?
Boolean
Int
Float
Char
Какими операторами пользуемся для программирования разветвлённых алгоритмов?
Оператор условного перехода и оператор выбора
Операторы ввода и вывода
Крутые операторы
Простые операторы
Найдите правильно описанный оператор условного перехода
IF (< логическое выражение >) S1; else S2;
IF< логическое выражение > then S1;
IF< логическое выражение > then;
Нет правильного ответа
Сколько сушествует операторов цыкла?
3
5
2
4
Найдите правильно написанный оператор цыкла с предусловием
While (<условие>); M;
For V: = L1 to (down to) L2 do M;
Repeat m until L;
If < условие > then S1 else S2;
Найдите правильно написанный оператор цыкла с постусловием
do m while(<условие>);
For V: = L1 to (down to) L2 do M;
While L do M;
If < условие > then S1 else S2;

Присоить переменной a значение переменной b
a=b;
a= =b;
b=a;
a <> b
Чему равен индекс последнего элемента массива, у которога 19 элементов?
18
17
19
20
Обший вид оператора условного перехода?
if (условие) <оператор1> ; else <оператор2>;
if (условие) then <оператор1> ; else <оператор2>;
<оператор1> if <условие>else<оператор2>
<оператор> if <условие>;
что распечатается на экране? int a = 5; float b = 2.0; cout<< b / a;
0.4
2
2.5
Ошибка
что распечатается на экране? float a = 2; int b = 32; cout<< (pow(a, 5.0) == b);
1
0
32
Ошибка
что распечатается на экране? int a =1; cout<< pow(a, 1 / 3);
1
3,107233
3

0
что распечатается на экране? int a = 4; int b = 2; cout<< b / a;
0
0,5
2
1
что распечатается на экране? int a = 5; cout<< a / 2;
2
2,5
1
3
что распечатается на экране? int i = 6; int j = i; cout<< (i * j);
36
12
15
78
На каком месте программы можно объявить переменные?
Где угодно
Внутри функции main
Вне функции main
Только внутри функций
Сколько раз будет повторяться следующий оператор for(; ;) ;
Безконечно
1
0
Ошибка
Самое большое значение типа int
2147483647
32767

32768
2147483648
Самое маленькое значение типа int
-2147483648
-32767
-32768
-2147483647
Чему служит функция pow ?
Возведение в степень
Квадратный корень
Оператор цыкла
Оператор выбора
int a=4; int b=3; double s=a/b; чему равно значение переменной S?
1
1.333333
2
3.11111
что распечатается на экране? int s = 0; for (int i = 0; i<= 7; i++) s=s+i; cout<<s;
28
25
22
20
x=6;y=6; if (x > 5) { if (y > 5) cout<< " x and y are > 5 " ; } else cout<< " x is <= 5 "; чта разпечатается на экране ?
x and y are > 5
x is <= 5
«x and y are > 5» ва « x is <= 5»
«x and y are > 5» ёки « x is <= 5»
Если int x; float y; найдите правильно написанный оператор
y = x + 2.3;

x:=2.0
x = y / x
Нет
Определите функцию которая возвращает целое значение и аргумент целого типа
int func1 (int a);
int func1 (float a);
float (float a);
float func1 (int a);
Определите функцию которая возвращает целое значение и аргумент вещественного типа
int func1 (float a);
int func1 (int a);
float (float a);
float func1 (float a);
Сколько раз будет повторяться тело цыкла? for (x=3; x<=0; x--) cout<<”+”<<”_”<<endl;
0
2
3
4
Сколько раз будет повторяться тело цыкла? for (x=3;x>=0;x- -) cout<<”+”<<”_”;
4
3
6
5
Сколько раз будет повторяться тело цыкла? for (x=4; x>=0; x--) cout<<”+”<<”_”<<endl;
5
6
3
4
Какая функция обязательно должна присудствовать в программе?

main()
local()
friend()
global()
Целые числа?
Int
Char
Void
Float
Оператор выбора?
Switch
Throw
Public
Struct
Как можно выйти из тела цыкла?
break
new
void
delete
Оператор с условным переходом?
If
Throw
Public
Switch
X>Y && A<B когда будет значение истина?
X>Y и A<B
X>Y и A>B
X<Y и A>B
X<Y и A<B

Как называются переменные объявленные внутри основной функции?
Локальные
Глобальные
Основные
Допольнитеоьные
Каким типом являются числа с плавающей точкой?
Float
Char
Int
Void
Какой тип относится к символьным?
char
float
int
bool
Логический тип переменных
bool
int
char
double
Для чего служат && и || ?
Для сравнения двух логических выражений
Для вычисления значений
Для сложение операторов
Ошибка
Какой орператор служит для определения места в памяти
Sizeof
Typedef
Define

Struct
Как производится использование элемента массива?
По имени и индексу массива
По имени массива
По индексу массива
Как угодно
Поток ввода?
‘>>’
‘->’
‘<<’
‘=>’
Поток вывода?
‘<<’
‘>>’
‘<=’
‘<-’
Найдите оператор выбора?
Switch
Throw
Public
Struct
С какого служебного слова начинается оператор условного выбора?
If
Throw
Public
For
Куда передаёт выполнение оператор goto?
На метку
На переменную

На функцию
Ни куда
Какой оператор передаёт управление на следующую итерацию цыкла?
Continue
Switch
Break
Goto
С помощью какого оператора можно выйти из тела цыкла?
break
new
delete
Void
Найдите общий вид оператора с условным переходом?
if (логическое выражение) {…} else {…}
while (логическое выражение) {…}
else (логическое выражение) else {…}
if (логическое выражение) else {…}
Что означает конструкция "while (логическое выражение) {…}" ?
Использование цыкла с предусловием
Использование цыкла с постусловием
Использование оператора выбора
Использование оператора перехода
Какими скабками определяется блок операторов?
{ }
( )
[ ]
[}
Если внутри цыкла существует оператор continue…
он переводит выполнение программы на следующую итерацию цыкла

он переводит выполнение программы за тело цыкла
он переводит выполнение программы на другую метку
он переводит выполнение программы на другой цыкл
Как объявляется функция котораня не возвращает значение?
Void
Int
Bool
Float
Что означает объявление файла iostream.h ?
Возможность использования потоков вводы и вывода
Возможность использования операторов цыкла
Возможность использования операторов выбора
Возможность использования массивов
С помощью какого служебного слова объявляется структура?
Struct
Switch
Public
For
С помощью какого служебного слова возвращается значение из функции
Return
EXIT_SUCCESS
Getline
Clear
На каком месте программы объявляется переменные?
Где угодно
Внутри функции
Вне теле функции
В конце функции
int a = 10, n = 6; for(int i = 0; i < n; i ++) a += i; чему равно значение a ?

25
30
35
40
Определите глобальные переменные ? #include <stdio.h> int a, n; double s; int main(){ int i, j, k; char c; return 0; }
a, n, s
a, n, c
i, j, k
i, j, k, c