Федеральное агентство по образованию

Федеральное государственное образовательное учреждение
высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

М. Э. Абрамян

Язык программирования C++

Часть 1

Базовые типы и управляющие операторы

Для студентов 1 курса
факультета математики, механики и компьютерных наук

Ростов-на-Дону 2008

Публикуется в электронном виде
в централизованной библиотеке учебно-методических ресурсов
Южного федерального университета
по решению кафедры алгебры и дискретной математики
факультета математики, механики и компьютерных наук ЮФУ
от 17 мая 2010 г. (протокол № 10)

Рецензенты:
к. ф.-м. н., доцент Михалкович С. С.,
к. ф.-м. н., доцент Чечин Г. М.

Аннотация

Учебное пособие является первым в серии из трех пособий, посвященных языку программирования C++. Оно разбито на 4 модуля, включающих как основное содержание, так и контрольные разделы (проектное задание и тесты рубежного контроля), позволяющие определить уровень усвоения материала. Каждый из модулей посвящен определенной теме, относящейся к начальной ступени изучения программирования: базовым числовым типам, вводу-выводу и операции присваивания (модуль № 1), логическим выражениям и условным операторам (модуль № 2), операторам цикла (модуль № 3), функциям и алгоритмам обработки числовых последовательностей (модуль № 4).

Пособие предназначено для студентов факультета математики, механики и компьютерных наук (направления «Прикладная математика», «Информационные технологии»).

Автор: М. Э. Абрамян.

Предисловие

Учебное пособие является первым в серии из трех пособий, посвященных языку программирования C++ [3–5, 7, 8]. В нем рассматриваются темы, относящиеся к начальной ступени изучения программирования.

В соответствии с *модульной технологией* организации учебного материала (см. [6]) пособие разбито на 4 модуля, включающих как основное содержание, так и контрольные разделы (проектное задание и тесты рубежного контроля), позволяющие определить уровень усвоения материала. Перечислим темы, рассмотренные в пособии: базовые числовые типы, ввод-вывод и операция присваивания (модуль № 1), логические выражения и условные операторы (модуль № 2), операторы цикла (модуль № 3), функции и алгоритмы обработки числовых последовательностей (модуль № 4). Помимо базовых сведений о различных возможностях языка, настоящее пособие содержит решения большого числа типовых задач. Кроме того, в пособии приводятся указания к более чем 300 задачам, выполнение которых позволит закрепить изученный материал (формулировки задач в пособие не включены, так как они ранее были опубликованы в методических указаниях [1–2]). Наличие большого количества учебных задач позволило представить проектные задания к каждому модулю в виде набора из 24 вариантов, что дает возможность преподавателю снабдить каждого студента отдельным вариантом проектного задания.

В приложении А приводится дополнительный справочный материал по языку C++.

Все задачи, рассмотренные в настоящем учебном пособии, включены в электронный задачник по программированию Programming Taskbook. Использование электронного задачника существенно ускоряет процесс выполнения заданий, так как избавляет учащегося от дополнительных усилий по

даний, так как избавляет учащегося от дополнительных усилий по организации ввода/вывода, а также позволяет отобразить исходные и результирующие данные в наглядном виде. Предоставляя учащемуся готовые исходные данные, задачник акцентирует его внимание на разработке и программной реализации *алгоритма* решения задания, причем разнообразие исходных данных обеспечивает надежное *тестирование* предложенного алгоритма.

При описании решений типовых задач используются возможности, предоставляемые электронным задачником Programming Taskbook (в частности, применяются специальный поток *pt* для ввода/вывода данных). Благодаря этим возможностям тексты программ с решениями удалось сделать более компактными и наглядными. Используемые в пособии средства задачника Programming Taskbook описываются в приложении В.

1. Модуль № 1. Числовые типы, ввод-вывод, операция присваивания

1.1. Комплексная цель

Ознакомиться со структурой программы на языке C++, изучить числовые типы *double* и *int*. Освоить ввод-вывод, основанный на стандартных потоках *cin* и *cout*. Научиться выполнять учебные задания, используя электронный задачник Programming Taskbook или стандартные консольные приложения. Освоить базовые алгоритмы, связанные с операцией присваивания (в частности, перемену местами содержимого двух или нескольких переменных) и целочисленными операциями (в частности, выделение цифр из многозначных целых чисел).

1.2. Содержание

1.2.1. Ввод-вывод данных с использованием задачника Programming Taskbook, операция присваивания

В данном пункте подробно описывается процесс выполнения учебных заданий на языке C++ с применением электронного задачника Programming Taskbook. В качестве среды программирования используется среда Visual Studio 2005, однако аналогичные результаты будут получены и при выполнении приведенных программ в средах C++Builder версии 4.0 и 5.0, Visual C++ 6.0 и Visual Studio 2003 и 2008.

Для работы с задачником Programming Taskbook каждому учащемуся выделяется специальный каталог (*рабочий каталог*). При индивидуальном обучении рабочий каталог может определить сам учащийся в программе регистрации и настройки задачника PT4Setup; при организации занятий в группе рабочие

каталоги создает и настраивает преподаватель с помощью программы PT4Teach. Все задания учащийся должен выполнять в своем каталоге; это обеспечит сохранение информации о работе учащегося в специальном *файле результатов* results.dat. В этом файле содержатся данные об учащемся (фамилия, имя и, возможно, номер варианта, например, «Иванов Петр :1») и сведения о результатах всех запусков программ с учебными заданиями. Удалять файл результатов из рабочего каталога не следует, так как это приведет к потере всех сведений о результатах выполнения учебных заданий**.**

Кроме файла результатов, каталог учащегося обычно содержит три *ярлыка*, которые обеспечивают запуск вспомогательных программных модулей электронного задачника: Demo.lnk (просмотр в демонстрационном режиме всех заданий, включенных в задачник), Load.lnk (генерация программы-шаблона для требуемого учебного задания и ее немедленная загрузка в выбранную среду программирования), Results.lnk (расшифровка, анализ и отображение на экране содержимого файла результатов).

Порядок выполнения учебного задания опишем на примере задания Begin3.

Begin3. Даны стороны прямоугольника a и b. Найти его площадь $S = a \cdot b$ и периметр $P = 2 \cdot (a + b)$.

Чтобы создать проект, содержащий заготовку для требуемого задания, воспользуемся программным модулем PT4Load; для этого перейдем в рабочий каталог задачника и запустим ярлык Load.lnk. На экране появится окно модуля PT4Load, в котором надо ввести имя нужного задания и нажать кнопку «Загрузка».

Отметим, что в заголовке окна модуля PT4Load указывается имя той программной среды, для которой будет создана заготовка. Приведем имена сред, связанных с языком C++: [CB4] и [CB5] — среда Borland C++Builder версии 4.0

и 5.0, [VC6] — среда Microsoft Visual C++ 6.0, [VCNET1], [VCNET2] и [VCNET3] — среда Microsoft Visual Studio 2003, 2005 и 2008.

Проект-заготовка, созданный для системы Visual Studio 2005 (как и для других систем программирования на C++), состоит из нескольких файлов, однако для решения задания нам потребуется только файл Begin3.cpp. Именно этот файл будет загружен в редактор среды Visual Studio. Приведем начальную часть содержимого данного файла:

```
#include <windows.h>
#pragma hdrstop
#include "pt4.h"

void solve()
{
  task("Begin3");
}
```

Эта часть содержит описание функции *solve*, в которой необходимо запрограммировать решение задания Begin3 (хотя, разумеется, в этом решении могут использоваться вспомогательные функции, описанные в том же файле Begin3.cpp или в других файлах, подключенных к проекту). Функция *solve* уже содержит вызов функции *task*, инициализирующей задание Begin3. Имя задания (в нашем случае Begin3) передается в функцию *task* в виде строкового параметра и поэтому должно заключаться в двойные кавычки: *"Begin3"*. Более подробно работа с функциями описывается в п. 4.2.1.

Такой же вид имеет начальная часть файла Begin3.cpp и при создании его в программных средах Visual C++ 6.0 и Visual Studio 2003 и 2008.

☑ В среде C++Builder файл Begin3.cpp будет содержать другой список директив, предшествующих описанию функции *solve*:

```
#include <vcl.h>
#pragma hdrstop
```

```
#include "pt4.h"
USEUNIT("pt4.cpp");
```

Файл Begin3.cpp содержит также описание *стартовой функции WinMain*, в которой производится вызов функции *solve*. Данный фрагмент текста программы не требует редактирования при выполнении задания, поэтому приводить его здесь мы не будем.

Следует обратить внимание на то, что к файлу Begin3.cpp подключается заголовочный файл pt4.h, содержащий описания вспомогательных типов, функций и переменных (в частности, функции *task*). Реализация данных функций содержится в файле pt4.cpp. Поскольку содержимое файла pt4.cpp не требует редактирования, данный файл не загружается в редактор среды Visual Studio. Однако он входит в проект, в чем можно убедиться, посмотрев на окно Solution Explorer, приведенное на рис. 1 (данное окно обычно располагается в правой части окна среды Visual Studio; если оно отсутствует на экране, то для его отображения достаточно ввести клавиатурную комбинацию [Ctrl]+[W], [S]).

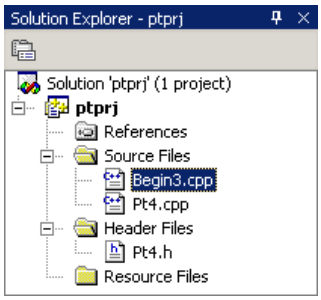

Рис. 1.

Имя созданного проекта ptprj не зависит от имени выбранного задания. Это сделано для того, чтобы при выполнении любого задания вспомогательные файлы, создаваемые системой Visual Studio и размещаемые в подкаталоге Debug, не дублировались, а просто заменяли ранее созданные. Тем самым обеспечивается существенная экономия дискового пространства, поскольку суммарный размер этих вспомогательных файлов может достигать нескольких мегабайтов.

Запустим программу, нажав клавишу [F5] или комбинацию клавиш [Ctrl]+[F5], запускающую программу в более быстром режиме с отключенным отладчиком (в среде Borland C++Builder для запуска программ предназначена клавиша [F9]). Перед компиляцией программы среда Visual Studio может вывести диалоговое окно, приведенное на рис. 2.

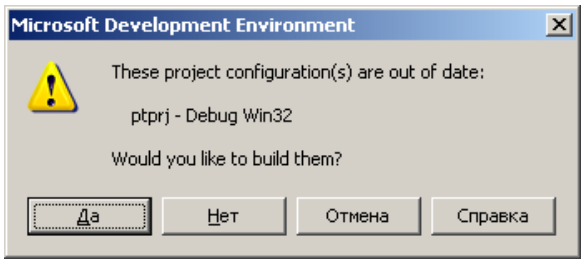

Рис. 2.

В этом случае надо выбрать вариант «Да» (достаточно нажать клавишу [Enter]). После компиляции и запуска программы мы сможем увидеть на экране окно задачника с формулировкой задания и примером исходных данных (см. рис. 3).

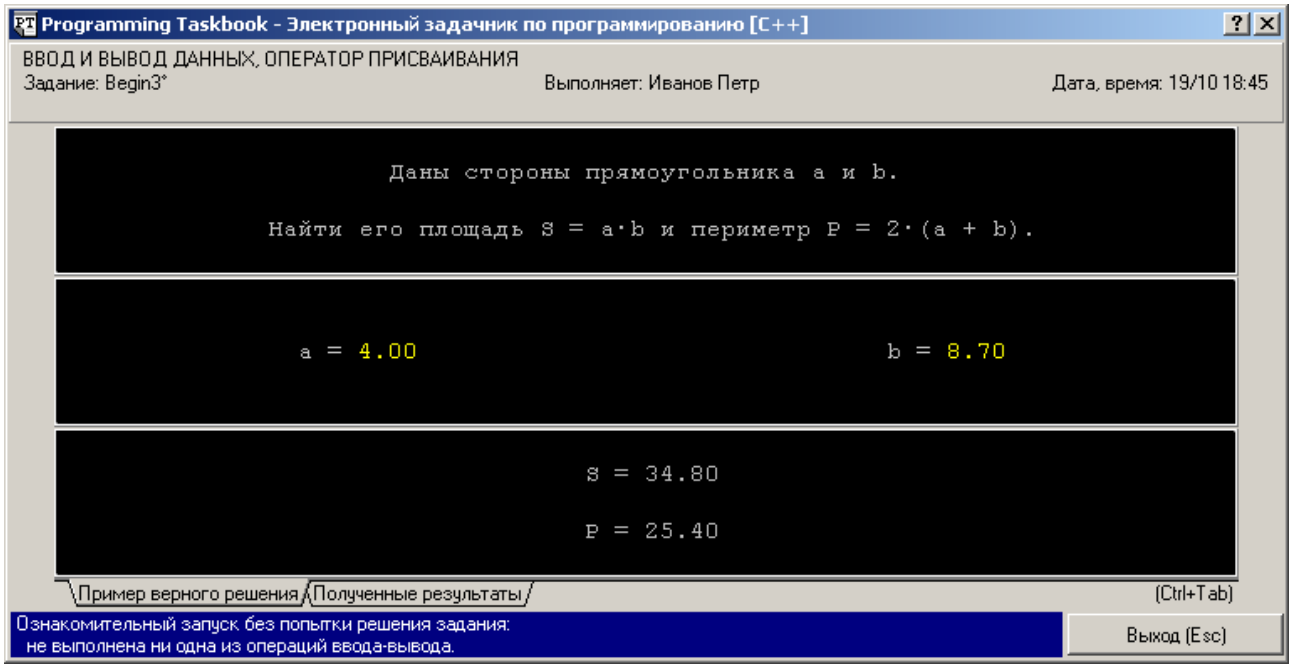

Рис. 3.

В верхней части этого окна выводится информация о теме, к которой относится выполняемое задание («ВВОД И ВЫВОД ДАННЫХ, ОПЕРАТОР ПРИ-

СВАИВАНИЯ»), название задания («Задание: Begin3»), сведения об учащемся («Выполняет: Иванов Петр»), дата и время запуска программы («19/10 18:45»). Затем следует раздел с *формулировкой задания*.

Далее располагается раздел с *исходными данными*. В нашем случае это числа *a* и *b* — длины сторон исходного прямоугольника. Наличие у чисел дробной части означает, что они являются *вещественными*. При необходимости данные могут снабжаться комментариями; здесь это строки «a =» и «b =», расположенные перед числами *a* и *b*.

Последним разделом является раздел *контрольных данных и результатов*, содержащий две вкладки: «Пример верного решения» и «Полученные результаты». Если в программе не выполняются действия по вводу-выводу данных, то при запуске программы активной становится вкладка «Пример верного решения» с *контрольными* (то есть «правильными») результирующими данными. В нашем случае в ней отображаются правильные значения площади и периметра, снабженные комментариями. Если перейти на вкладку с результатами, щелкнув мышью на ярлычке «Полученные результаты» или нажав комбинацию клавиш [Ctrl]+[Tab], то в ней отобразятся только комментарии, поскольку наша программа пока не выводит никаких результирующих данных.

Чтобы отличить комментарии от данных (как исходных, так и результирующих), используется *цветовое выделение*: комментарии выводятся светло-серым цветом, данные — желтым. Контрольные данные отображаются тем же цветом, что и комментарии (для того чтобы подчеркнуть, что эти данные *не найдены* программой учащегося).

В нижней части окна располагается *информационная панель* и кнопка «Выход». Так как наша программа не выполняет действий по вводу-выводу данных, ее текущий запуск рассматривается как *ознакомительный*, и проверка правильности решения не производится. Для выхода из программы надо нажать кнопку «Выход» или клавишу [Esc]. Можно также нажать клавишу [F5], то есть

ту же клавишу, которая использовалась для запуска программы из среды Visual Studio.

Запустив программу еще раз, мы увидим, что исходные данные изменились. При каждом запуске программы генерируется новый набор исходных данных, поэтому для успешного решения задания необходимо запрограммировать алгоритм, правильно работающий *для любого допустимого набора исходных данных*.

До сих пор мы не пытались решать задание, а только знакомились с ним. Теперь перейдем к решению задания. Для того чтобы найти площадь и периметр прямоугольника, требуется ввести длины его сторон. Стандартная библиотека языка C++ для выполнения операций ввода использует специальные классы, называемые *потоками ввода*, которые обеспечивают ввод исходных данных с клавиатуры (или их считывание из какого-либо файла). У нас — другая ситуация: исходные данные уже сформированы функцией *task* при инициализации задания; нам требуется лишь получить к ним доступ. Для этого надо использовать *специальный поток ввода pt*, предусмотренный в электронном задачнике Programming Taskbook (он реализован в файле pt4.cpp). Для получения данных из потока *pt*, как и из стандартных потоков ввода, используется операция >>.

В нашем случае необходимо ввести два элемента исходных данных. Добавим в функцию *solve* операторы, обеспечивающие описание и ввод исходных данных:

```
void solve()
{
  task("Begin3");
  double a, b;
  pt >> a >> b;
}
```

Оператор, начинающийся со слова *double*, предназначен для описания переменных *вещественного типа*, то есть переменных, предназначенных для хранения чисел, которые могут содержать дробную часть. Размер типа *double* зависит от реализации. Во всех рассматриваемых реализациях языка C++ для хранения вещественного числа типа *double* отводится 8 байтов. Вещественные числа можно хранить с точностью до 15–16 значащих цифр, а их значения по модулю не могут превосходить $1.7 \cdot 10^{308}$. Заметим, что для очень больших (и очень маленьких вещественных чисел) удобно использовать *экспоненциальный числовой формат*. Так, максимальное вещественное число записывается в этом формате следующим образом: *1.7e308* (вначале указывается *мантисса* числа — число, модуль которого лежит в диапазоне 1–10, затем буква *e*, а затем — *порядок* числа, то есть показатель степени 10, на которую надо умножить мантиссу для получения требуемого числа).

Поскольку переменные *a* и *b* описаны внутри функции *solve*, они являются локальными переменными и существуют только при выполнении этой функции. Начальное значение локальных переменных является неопределенным, поэтому перед их использованием необходимо явно присвоить им некоторые значения. В нашей функции это делается в последнем операторе, который считывает исходные данные из потока ввода *pt* и последовательно записывает их в переменные *a* и *b*.

Запустив программу, мы получим сообщение: «*Выведены не все результирующие данные. Количество выведенных данных: 0 (из 2)*», причем фон информационной панели станет красным. Ситуация понятна: приступая к вводу исходных данных, мы тем самым продемонстрировали намерение выполнить задание. Однако мы не вывели требуемые результирующие данные, и поэтому электронный задачник не смог их проанализировать и сравнить с контрольными данными.

Итак, для верного решения задания нам осталось получить требуемые результаты и вывести их. Здесь, как и в случае с вводом, нельзя использовать стандартные потоки вывода из библиотеки C++, поскольку нам требуется не только отобразить результаты на экране, но и передать их электронному задачнику для сравнения с контрольными данными. Нужные действия обеспечивает специальный поток вывода, реализованный в электронном задачнике Programming Taskbook. Имя этого потока совпадает с именем использованного ранее потока ввода: *pt* (иными словами, поток *pt* можно рассматривать как *поток ввода-вывода*). Для передачи данных в поток *pt*, как и в стандартные потоки вывода, используется операция <<.

Еще раз подчеркнем, что поток ввода-вывода *pt не входит в стандартную библиотеку C++*. Он описан в задачнике Programming Taskbook и может использоваться *только* при выполнении учебных заданий.

☑ Помимо потока *pt* для ввода и вывода данных в задачнике предусмотрены два набора *функций*: функции группы *get* для ввода и функции группы *put* для вывода (см. п. 6). Однако в настоящем пособии при организации ввода-вывода мы будем пользоваться только потоком *pt*.

Добавим в конец функции *solve* вычисление и вывод результатов:

```
void solve()
{
    task("Begin3");
    double a, b;
    pt >> a >> b;
    double S = a * b, P = 2 * (a + b);
    pt << P << S;
}
```

В добавленном к функции фрагменте мы описали еще две переменные вещественного типа и сразу инициализировали их требуемыми выражениями (в переменную *S* записано значение площади прямоугольника, а в переменную *P*

— значение его периметра). Последний оператор обеспечивает передачу найденных значений *P* и *S* в поток вывода *pt*.

После запуска программы мы увидим, что найденные значения выведены в разделе результирующих данных. Но из-за того что *вначале* мы вывели значение периметра *P*, а *затем* — значение площади *S*, решение опять будет признано неверным. Информационная панель теперь содержит сообщение «*Ошибочное решение*», означающее, что все исходные данные были введены, все результирующие данные выведены, но *значения полученных результатов не совпадают с контрольными*. Перейдя на вкладку «Пример верного решения», мы можем убедиться, что площадь и периметр найдены верно и нарушен только порядок их вывода.

Таким образом, важно не только найти правильные значения результирующих данных, но и *вывести их в нужном порядке*. Для того чтобы указать порядок вывода, в раздел результатов обычно добавляются *комментарии* (см. рис. 3). Исходные данные тоже важно вводить именно в том порядке, в котором они указаны в соответствующем разделе окна задачника. Общее правило здесь следующее: ввод и вывод данных производится по экранным строкам (*слева направо*), а строки просматриваются *сверху вниз*. Иными словами, данные вводятся и выводятся в том порядке, в котором они читались бы в обычном тексте (на русском языке).

Изменяя порядок вывода результирующих данных, мы, наконец, получим правильный вариант программы.

Следует заметить, что в использовании вспомогательных переменных *S* и *P* при выполнения задания Begin3 нет необходимости, поскольку при выводе данных можно указывать не только переменные, но и выражения. Учитывая это замечание, можно получить более короткий вариант решения задания Begin3:

```
void solve()
{
```

```
    task("Begin3");
    double a, b;
    pt >> a >> b;
    pt << a * b << 2 * (a + b);
}
```

После запуска правильного варианта информационная панель (фон которой теперь станет зеленым) будет содержать сообщение «*Верное решение. Тест номер 1 (из 3)*». Мы провели первое успешное испытание нашей программы. Однако однократное успешное испытание алгоритма еще не означает, что он является правильным. Для того чтобы получить уверенность в правильности алгоритма**,** *его надо протестировать несколько раз на различных наборах исходных данных*. Количество тестовых испытаний, требующихся для каждого учебного задания, различно и может меняться в пределах от 3 до 10 (для простого задания Begin3 число испытаний равно трем). Если нужное количество испытаний, *проведенных подряд*, прошло успешно, то на информационной панели появится сообщение «*Задание выполнено!*». После этого можно переходить к следующему заданию. Если же в ходе очередного испытания было получено неверное решение, то счетчик количества успешных испытаний для данного задания будет сброшен в 0, и после исправления алгоритма программу придется тестировать заново.

Напомним, что закрыть окно программы с учебным заданием можно не только нажатием клавиши [Esc], но и нажатием клавиши [F5], то есть той же клавиши, которая в среде Visual Studio запускает программу на выполнение. Таким образом, для проверки программы на нескольких наборах исходных данных достаточно несколько раз подряд нажать [F5].

Информация о ходе выполнения задания записывается в файл результатов. Выше было отмечено, что для просмотра этого файла можно использовать ярлык Results.lnk, находящийся в рабочем каталоге учащегося.

В заключение данного пункта приведем решение еще одного задания из группы Begin.

Begin22. Поменять местами содержимое переменных *A* и *B* и вывести новые значения *A* и *B*.

Поскольку решение должно оформляться в виде функции *solve*, здесь и далее в примерах решений будем приводить только операторы, входящие в эту функцию.

```
task("Begin22");
double a, b;
pt >> a >> b;
double temp = a;
a = b;
b = temp;
pt << a << b;
```

Стандартный алгоритм обмена содержимым двух переменных одного типа состоит в использовании вспомогательной переменной того же типа. В нашем случае этой переменной является переменная *temp*, в которую записывается исходное значение переменной *a*. После этого в переменную *a* можно записать значение переменной *b*, а в переменную *b* — исходное значение переменной *a*, хранящееся во вспомогательной переменной *temp*.

Для присваивания нового значения переменной используется специальная операция (*операция присваивания*), имеющая вид:

```
переменная = выражение;
```

В результате выполнения этой операции значение *выражения*, указанного справа от знака присваивания =, записывается в *переменную*, указанную слева от этого знака. Заметим, что тип переменной должен быть *совместим* по присваиванию с типом выражения. В частности, если переменная имеет тип *double*, то выражение может иметь числовой тип (как вещественный — *double*, так и целый, например, *int*).

1.2.2. Стандартный ввод-вывод, особенности консольных приложений

Пример выполнения задания Begin3, подробно описанный в предыдущем пункте, показывает, что использование электронного задачника существенно упрощает ввод и вывод данных, поскольку при этом не требуется ни вводить исходные данные с клавиатуры, ни специальным образом форматировать полученные данные при их выводе на экран. Однако задания можно выполнять и без использования электронного задачника; это лишь потребует дополнительных усилий, связанных с подготовкой и вводом исходных данных и с выводом результатов; кроме того, в этом случае сам учащийся должен будет подготовить набор тестовых данных, с помощью которого он сможет проверить правильность разработанной программы.

В данном пункте описывается, как выполнять задания без использования электронного задачника в системе Microsoft Visual Studio .NET 2005. При этом приводится информация об имеющихся в библиотеке C++ стандартных потоках ввода-вывода.

Перед началом выполнения задания необходимо создать заготовку для новой программы. Для выполнения учебных заданий проще всего использовать так называемые *консольные приложения*, которые осуществляют ввод исходных данных и вывод результатов в специальное *консольное окно*.

Заготовку для консольного приложения можно создать, выполняя следующие действия: в меню Visual Studio выберите команду «File | New | Project…», в появившемся окне «New Project» выберите в разделе «Project types», расположенном слева, группу «Visual C++» (эта группа может располагаться в группе первого уровня «Other Languages») и в этой группе вариант «Win32»; при этом в правой части окна появится список имеющихся видов проектов. В этом списке надо выбрать вариант «Win32 Console Application». Далее, в нижней части окна надо указать сведения, необходимые для генерации заготовки выбранного

проекта: *имя проекта* (строка «Name»; в качестве имени целесообразно указывать имя выполняемого задания, например, Begin3) и *каталог*, в котором этот проект будет размещен (строка «Location»; в этой строке следует указывать свой рабочий каталог). Для того чтобы не создавать сложной структуры подкаталогов, связанных с создаваемым проектом, флажок «Create directory for solution» надо снять.

После выполнения всех описанных действий надо нажать в окне «New Project» кнопку [OK]; при этом будет запущен *мастер по созданию заготовок для Windows-приложений* (Win32 Application Wizard). В первом окне мастера следует нажать кнопку [Next], а во втором («Application Settings») — установить флажок «Empty project», после чего можно нажимать кнопку [Finish]. В результате будет создан пустой проект с указанным именем, и этот проект немедленно загрузится в среду Visual Studio. Заметим, что для данного проекта будет создан подкаталог каталога, указанного в строке «Location»; имя этого подкаталога будет совпадать с именем проекта. Таким образом, каждый проект размещается в отдельном каталоге, что предотвращает опасность «перемешивания» файлов, относящихся к различным проектам.

Перед написанием программы в проект нужно добавить новый файл с исходным кодом. Для этого достаточно выполнить команду «Project | Add New Item…», в появившемся окне «Add New Item» выбрать в разделе «Categories», расположенном слева, группу «Visual C++» и в этой группе выбрать вариант «Code»; при этом в правой части окна появится список возможных типов файлов. В этом списке надо выбрать вариант «C++ File (.cpp)», а в нижней части окна указать имя файла (в качестве имени удобно указывать имя выполняемого задания, например, Begin3). После нажатия кнопки «Add» в раздел «Source Files» проекта будет добавлен пустой файл с указанным именем.

Поскольку при выполнении заданий мы собираемся использовать стандартные потоки ввода-вывода, а также классы, реализованные в библиотеке

C++, и распространенные функции, перенесенные в C++ из библиотеки C, в данном файле следует набрать следующую заготовку для основной программы:

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

int main()
{

    return 0;
}
```

Если при использовании задачника Programming Taskbook решение задания следовало вводить в функцию *solve*, то в консольном приложении решение надо запрограммировать непосредственно в функции *main*. При этом отличия проявятся лишь в *организации ввода-вывода*.

Откорректируем первый вариант решения задания Begin3, приведенный в предыдущем пункте, организовав ввод и вывод с помощью стандартных потоков:

```
int main()
{
    double a, b;
    cout << "a = ";
    cin >> a;
    cout << "b = ";
    cin >> b;
    double S = a * b, P = 2 * (a + b);
    cout << "S = " << S << endl;
    cout << "P = " << P << endl;
    system("pause");
    return 0;
```

```
}
```

Как видно из приведенного текста программы, для ввода данных с клавиатуры используется поток *cin*, а для вывода данных в консольное окно — поток *cout*, причем для перехода на новую строку следует выводить в поток специальный *манипулятор endl*. В выходной поток *cout* можно передавать данные любого базового типа, в частности строковые литералы, заключаемые в двойные кавычки. Подобным образом в начале программы выводятся *приглашения* к вводу исходных данных: *"a = "* и *"b = "*, а в конце — *комментарии* к выводимым результатам *"S = "* и *"P = "*.

Вызов функции *system("pause")* приостанавливает работу программы до тех пор, пока не будет нажата какая-либо клавиша (при этом на экран выводится текст «Для продолжения нажмите любую клавишу»). Таким образом, после вывода полученных результатов, программа будет ожидать нажатия какой-либо клавиши, консольное окно останется на экране, и мы сможем ознакомиться с его содержимым. После нажатия любой клавиши программа завершит работу, что приведет к немедленному закрытию консольного окна. Если бы вызов функции *system* в конце программы отсутствовал, то после вывода результатов консольное окно было бы немедленно закрыто, и мы не успели бы ознакомиться с его содержимым.

Приведем образец исходных данных и выведенных для них результатов (обратите внимание на использование точки в качестве десятичного разделителя):

```
a = 5.25
b = 7.5
S = 39.375
P = 25.5
```

Если при выполнении задания надо обработать много исходных данных (например, массив чисел), то для ввода данных с клавиатуры при каждом тестовом запуске программы может потребоваться много времени. В этом случае

удобно вместо ввода данных с клавиатуры генерировать их с помощью программного *датчика случайных чисел*; это позволит при каждом запуске испытывать программу на новом наборе исходных данных, который будет формироваться автоматически (подобно тому, как это происходит в электронном задачнике). Заметим, что на самом деле программный датчик генерирует *псевдослучайные* числа, поскольку для генерации действительно случайных чисел необходимо использовать специальные устройства.

Для генерации случайных чисел в стандартной библиотеке C++ предусмотрены функции *srand* и *rand*, а также константа *RAND_MAX* (см. п. 5.4).

Функция *srand* предназначена для инициализации датчика случайных чисел; ее достаточно вызвать один раз в начале программы. Для того чтобы датчик при каждом запуске генерировал различные последовательности случайных чисел, в качестве параметра функции *srand* следует указывать текущее время по системным часам компьютера. Для этого предназначена функция *time*, объявленная в заголовочном файле *ctime* (таким образом, к программе необходимо добавить директиву *#include <ctime>*). Использование этой функции в качестве параметра *srand* должно иметь следующий вид:

```
srand(static_cast<unsigned>(time(0)));
```

Конструкция *static_cast<unsigned>* обеспечивает преобразование значения *time*(0) к типу *unsigned* (по поводу преобразования типов см. п. 5.2).

Функция *rand* возвращает случайное целое число, лежащее в диапазоне от 0 до *RAND_MAX* включительно. Поэтому для получения очередного случайного вещественного числа, лежащего на отрезке [0, 1] достаточно выполнить деление сгенерированного случайного числа (*преобразованного к типу double*) на *RAND_MAX*. Заметим, что для получения вещественного числа, лежащего на отрезке [*A*, *B*], достаточно применить к случайному числу *x* из отрезка [0, 1] следующее преобразование: $A + x * (B - A)$. Если *A* и *B* — целые числа, то, при-

ведя полученный результат к типу *int*, можно получить *целое* случайное число, принимающее с равной вероятностью значения в диапазоне от A до $B - 1$.

Приведем измененный вариант программы, решающей задание Begin3, в котором длины сторон прямоугольника a и b генерируются с помощью датчика случайных чисел (предполагается, что эти данные могут принимать значения из промежутка [1, 10]):

```
int main()
{
    srand(static_cast<unsigned>(time(0)));
    double
        a = 1 + static_cast<double>(rand()) / RAND_MAX * 9,
        b = 1 + static_cast<double>(rand()) / RAND_MAX * 9;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    double S = a * b, P = 2 * (a + b);
    cout << "S = " << S << endl;
    cout << "P = " << P << endl;
    system("pause");
    return 0;
}
```

Поскольку теперь исходные данные не вводятся с клавиатуры, необходимо выводить их на экран с помощью оператора << потока *cout*.

Приведем результат, который будет получен при запуске нового варианта решения (разумеется, числа будут другими):

```
a = 1.66112
b = 8.37727
S = 13.9157
P = 20.0768
```

Этот результат показывает, что для вещественных чисел с большим количеством значащих цифр указывается 6 цифр. Часто бывает удобнее зафиксиро-

вать число цифр *в дробной части*, положив его равным, например, 2. Для этого надо установить для потока *cout* две настройки форматирования: во-первых, задать вывод вещественных чисел в формате с фиксированной точкой и, во-вторых, задать количество цифр в дробной части:

```
cout.setf(ios::fixed, ios::floatfield);
cout.precision(2);
```

☑ Если указать только второй из приведенных операторов, то он определит количество выводимых *значащих цифр* в любом вещественном числе (например, число 55.55 будет выведено в виде `56`).

После добавления в начало функции *main* данных операторов результат выполнения программы будет следующим:

```
a = 2.32
b = 5.96
S = 13.85
P = 16.57
```

Можно также указать ширину поля вывода, то есть число экранных позиций, отводимых для вывода очередного элемента данных. Если для всех четырех чисел в нашей программе указать одинаковую ширину поля вывода (например, 5), то эти числа будут *выровнены по правому краю*, то есть их дробные части будут располагаться на одном уровне по вертикали. Настройка ширины поля вывода, в отличие от настройки точности, должна выполняться для каждого элемента выводимых данных, поэтому в этом случае удобнее использовать манипулятор *setw*, описанный в заголовочном файле *iomanip*, который указывается непосредственно перед выводимым элементом в операторе <<, например:

```
cout << "a = " << setw(5) << a << endl;
```

После добавления перед каждым из выводимых чисел указанного манипулятора результат выполнения программы примет, наконец, наиболее наглядный вид:

```
a =  2.93
```

```
b =  6.84
S = 20.03
P = 19.54
```

Дополнительная проблема возникает при попытке использовать в комментариях (или выводимых строковых данных) русских букв. Эта проблема связана с тем, что кодировка, используемая в Windows (в частности, в редакторе среды Visual Studio), не совпадает для русских букв с кодировкой, используемой при отображении данных в консольном окне (так называемой *OEM-кодировкой*). Простейшим способом решения данной проблемы является использование функций *CharToOemA* и *OemToCharA*, реализованных в системной библиотеке Win32 API и доступных из С-программы после подключения заголовочного файла windows.h:

```
#include "windows.h"
```

Функция *CharToOemA* позволяет перевести строки из Windows-кодировки в OEM-кодировку; функция *OemToCharA* выполняет обратную перекодировку. Обе функции имеют по два параметра типа *char**. Первый параметр является входным и содержит исходную строку. Второй является выходным и содержит перекодированный вариант исходной строки. Так как для правильной работы данных функций необходимо, чтобы второй параметр — символьный массив — содержал достаточно элементов для хранения перекодированной строки, удобно заключить каждую из функций в «оболочку», упрощающую их использование. В качестве примера подобной оболочки приведем функцию *to_oem*(*s*), которая переводит строку *s* в OEM-кодировку и возвращает ее в виде массива типа *char**:

```
char* to_oem(char* s)
{
    char* res = new char[strlen(s) + 1];
    CharToOemA(s, res);
    return res;
```

```
}
```

Имея подобную функцию, мы получаем возможность выводить русский текст в консольное окно, например:

```
cout << to_oem("Периметр = ") << setw(5) << P << endl;
```

В заключение настоящего пункта нелишним будет привести итоговый текст программы, вместе со всеми подключенными заголовочными файлами:

```
#include "windows.h"
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <iomanip>
using namespace std;

char* to_oem(char* s)
{
    char* res = new char[strlen(s) + 1];
    CharToOemA(s, res);
    return res;
}

int main()
{
    srand(static_cast<unsigned>(time(0)));
    double
        a = 1 + static_cast<double>(rand()) / RAND_MAX * 9,
        b = 1 + static_cast<double>(rand()) / RAND_MAX * 9;
    cout.setf(ios::fixed, ios::floatfield);
    cout.precision(2);
    cout << "a = " << setw(5) << a << endl;
    cout << "b = " << setw(5) << b << endl;
    double S = a * b, P = 2 * (a + b);
```

```
    cout << to_oem("Площадь =  ") << setw(5) << S << endl;
    cout << to_oem("Периметр = ") << setw(5) << P << endl;
    system("pause");
    return 0;
}
```

Напомним, что данная программа решает достаточно простое задание Begin3, и основная часть использованных в ней средств языка C++ предназначена не для нахождения требуемого результата, а для подготовки исходных данных и наглядного вывода как исходных, так и результирующих данных. При выполнении заданий с использованием электронного задачника все эти вспомогательные действия берет на себя сам задачник, что позволяет учащемуся сосредоточить свои усилия на программной реализации *алгоритма* решения задания.

1.2.3. Работа с целыми числами

Для описания целых чисел в C++ используется описатель *int*. Размер данных этого типа зависят от реализации; во всех рассматриваемых средах данные типа *int* имеют размер 4 байта и принимают значения в диапазоне от –2147483648 до 2147483647.

☑ В языке C++ имеются и другие целочисленные типы, например, *short* и *long*. Размер целочисленного типа *short* в любой реализации не превосходит размера *int*, а размер типа *int* в любой реализации не превосходит размера *long*. Во всех рассматриваемых реализациях тип *short* имеет размер 2 байта и диапазон значений от –32768 до 32767, а тип *long* совпадает с типом *int*, то есть имеет размер 4 байта.

Для целых чисел (как и для вещественных) определены операции сложения +, вычитания –, умножения *, причем результат каждой из этих операций с целочисленными операндами является целым числом.

Операция / для операндов целого типа означает *деление нацело* (с отбрасыванием остатка от деления), а операция нахождения остатка от целочисленного деления обозначается символом %. Остаток от деления двух положительных чисел является положительным числом.

☑ Если один или оба операнда операции % являются отрицательными, то знак результата зависит от реализации. Во всех рассматриваемых средах знак выражения *a* % *b* совпадает с знаком первого операнда (*a*) и не зависит от знака второго (*b*); например, выражения –4 % –3 и –4 % 3 равны –1, а выражение 4 % –3 равно 1.

Использование операций целочисленного деления продемонстрируем на примере решения задания Integer8.

Integer8. Дано двузначное число. Вывести число, полученное при перестановке цифр исходного числа.

```
task("Integer8");
int n;
pt >> n;
int a = n / 10,
int b = n % 10;
pt << b * 10 + a;
```

В данном решении первая слева цифра (десятки) двузначного числа *n* сохраняется в переменной *a*, а вторая цифра (единицы) — в переменной *b*. При перестановке цифр эти переменные «меняются ролями»: *b* обозначает количество десятков, *a* — количество единиц.

Аналогичным образом можно выделять цифры и из чисел большей разрядности. Приведем в качестве примера решение задания Integer11.

Integer11. Дано трехзначное число. Найти сумму и произведение его цифр.

```
task("Integer11");
int n;
pt >> n;
int c = n % 10;
n = n / 10;
int b = n % 10;
int a = n / 10;
pt << a + b + c << a * b * c;
```

В данном решении цифры числа *n* определяются в обратном порядке (справа налево): в *c* записываются единицы, в *b* — десятки, в *a* — сотни.

☑ Оператор *n* = *n* / 10 можно было записать с использованием комбинированной операции *a* /= *b*, означающей, что вначале для операндов *a* и *b* выполняется операция /, а затем результат записывается в левый операнд *a* (который должен быть переменной):

```
n /= 10;
```

Аналогичные комбинированные операции предусмотрены для всех арифметических операций: +=, –=, *=, %=.

1.3. Проектное задание

Выполните задачи групп Begin и Integer, указанные в вашем варианте проектного задания (формулировки задач приводятся в [1]). Если вы не получили вариант проектного задания, то выполните задачи из какого-либо варианта.

ВАРИАНТ 1 Begin: 6, 11, 18, 24, 33 Integer: 4, 13, 21, 28	**ВАРИАНТ 2** Begin: 5, 10, 20, 27, 36 Integer: 3, 15, 21, 27	**ВАРИАНТ 3** Begin: 2, 9, 19, 27, 30 Integer: 4, 16, 20, 28
ВАРИАНТ 4 Begin: 7, 11, 19, 23, 31 Integer: 5, 12, 23, 25	**ВАРИАНТ 5** Begin: 7, 8, 20, 27, 40 Integer: 1, 14, 23, 27	**ВАРИАНТ 6** Begin: 1, 8, 16, 23, 39 Integer: 1, 13, 22, 24
ВАРИАНТ 7 Begin: 6, 9, 17, 28, 34 Integer: 3, 16, 19, 24	**ВАРИАНТ 8** Begin: 4, 12, 21, 23, 32 Integer: 2, 12, 23, 26	**ВАРИАНТ 9** Begin: 5, 13, 16, 24, 38 Integer: 6, 10, 20, 25
ВАРИАНТ 10 Begin: 2, 12, 17, 28, 37 Integer: 5, 14, 19, 28	**ВАРИАНТ 11** Begin: 1, 13, 21, 28, 29 Integer: 2, 15, 23, 26	**ВАРИАНТ 12** Begin: 4, 10, 18, 24, 35 Integer: 6, 10, 22, 28

ВАРИАНТ 13 Begin: 5, 9, 16, 24, 30 Integer: 3, 10, 21, 24	**ВАРИАНТ 14** Begin: 7, 10, 17, 23, 33 Integer: 1, 10, 19, 26	**ВАРИАНТ 15** Begin: 6, 11, 18, 28, 31 Integer: 2, 12, 19, 27
ВАРИАНТ 16 Begin: 1, 13, 18, 23, 38 Integer: 5, 14, 23, 25	**ВАРИАНТ 17** Begin: 4, 12, 19, 24, 35 Integer: 4, 15, 21, 28	**ВАРИАНТ 18** Begin: 4, 12, 17, 27, 40 Integer: 5, 14, 22, 25
ВАРИАНТ 19 Begin: 1, 8, 19, 28, 39 Integer: 6, 15, 20, 28	**ВАРИАНТ 20** Begin: 2, 11, 21, 27, 34 Integer: 4, 16, 20, 27	**ВАРИАНТ 21** Begin: 2, 10, 20, 23, 32 Integer: 3, 13, 23, 24
ВАРИАНТ 22 Begin: 5, 13, 21, 24, 36 Integer: 6, 13, 23, 26	**ВАРИАНТ 23** Begin: 6, 8, 16, 27, 29 Integer: 1, 12, 22, 28	**ВАРИАНТ 24** Begin: 7, 9, 20, 28, 37 Integer: 2, 16, 23, 28

1.3.1. Указания к заданиям группы Begin

Begin1–9. Простейшие задания на ввод–вывод данных. Для решения этих заданий не требуется использовать вспомогательные переменные и операцию присваивания. Решение задания Begin3 приводится в п. 1.2.1.

Begin10–11. Для хранения квадратов (Begin10) и модулей (Begin11) исходных чисел используйте вспомогательные переменные. Для нахождения модуля числа предназначена стандартная математическая функция *abs* (см. п. 5.3).

Begin12–15. В этих заданиях некоторые из результирующих данных используются при последующих вычислениях, поэтому перед выводом таких данных их следует сохранить во вспомогательных переменных. Например, при выполнении Begin12 следует сохранить найденную гипотенузу (ис-

пользуемую далее при вычислении периметра треугольника). Для вычисления квадратного корня используйте стандартную математическую функцию *sqrt*.

Begin16–19. При вычислении длин отрезков используйте функцию *abs* (заметим, что в Begin18 правильный ответ можно получить и без использования этого метода).

Begin22–24. Решение задания Begin22 приводится в п. 1.2.1. Задания Begin23–24 решаются аналогично.

Begin25–26. В Begin25 при вычислении x^6 используйте ранее найденное значение x^2; аналогично, в Begin26 при вычислении $(x-3)^6$ используйте ранее найденное значение $(x-3)^3$. При вычислении $(x-3)^3$ удобно предварительно сохранить значение выражения $x-3$ во вспомогательной переменной.

Begin29–40. Эти задания можно рассматривать как дополнительные. В большинстве из них требуется самостоятельно получить расчетные формулы, которые необходимо использовать при нахождении результирующих данных.

1.3.2. Указания к заданиям группы Integer

Integer1–5. Простейшие задания на применение операций целочисленного деления (Integer1–4) и нахождения остатка от деления (Integer5).

Integer6–10. Решение Integer8 приводится в п. 1.2.3, прочие задания решаются аналогично.

Integer11–16. Решение Integer11, выделяющее из трехзначного числа все цифры, приводится в п. 1.2.3. Используя полученные цифры, нетрудно сформировать числа, требуемые в заданиях Integer12–16 (ср. с решением Integer8).

Integer19–23. Для нахождения количества полных минут (задание Integer19) достаточно выполнить целочисленное деление *N* на 60; прочие задания решаются аналогично: в Integer20 используйте операцию целочисленного деления, в Integer21–22 — операцию взятия остатка от деления, в Integer23 — обе эти операции.

Integer24–28. В Integer24 номер дня недели для *K*-го дня года получается в результате нахождения остатка от целочисленного деления *K* на 7; это легко определить, составив таблицу зависимости номера дня недели от *K* для нескольких начальных значений *K* (например, для *K* = 1, 2, …, 10). В остальных заданиях для нахождения требуемых формул также удобно составить таблицу зависимости (заметим, что в этих формулах, помимо операции взятия остатка от деления, будут использоваться и другие арифметические операции).

Integer29. В ходе решения необходимо определить, сколько сторон квадрата *c* умещается на сторонах прямоугольника *a* и *b* (заметим, что сравнение *площадей* квадрата и прямоугольника не позволит получить правильный ответ).

1.4. Тест рубежного контроля

Тест содержит 6 заданий, на выполнение которых отводится 3 минуты. Выберите правильный, по вашему мнению, вариант ответа и отметьте его любым значком в бланке ответов.

1. Переменная *a* имеет значение 3, переменная *b* — значение 2; обе переменные имеют тип *double*. Укажите значения, которые будут содержать эти переменные после выполнения следующих операторов:
`a = a + b;`
`b = a - b;`			

(1)	*a* = 5, *b* = 1	(2)	*a* = 5, *b* = 3

(3)	$a = 3, b = 5$	(4)	$a = 3, b = 1$

2. Укажите оператор, позволяющий приостановить выполнение консольного приложения до нажатия любой клавиши.

(1)	`Pause;`	(2)	`system("pause");`
(3)	`system.pause;`	(4)	`System("Pause");`

3. Укажите значение, которое *не может* содержаться в переменной *a* типа *double* после выполнения оператора

```
a = static_cast<double>(rand()) / RAND_MAX * 100 - 50;
```

(1)	0.0	(2)	20.0
(3)	–25.0	(4)	60.0

4. Укажите выражение, значение которого равно количеству нечетных чисел, лежащих в диапазоне от 1 до *n* (включая концы диапазона), если *n* — некоторое целое положительное число.

(1)	$n / 2$	(2)	$n / 2 + 1$
(3)	$n / 2 - 1$	(4)	$(n + 1) / 2$

5. Укажите значение, которое будет содержаться в переменной *b* типа *double* после выполнения оператора

```
b = 10 + 14 / 4 * 2;
```

(1)	3.0	(2)	11.75
(3)	16.0	(4)	17.0

6. Укажите значение, которое будет содержаться в переменной *k* типа *int* после выполнения оператора

```
k = 4632 / 100 % 10;
```

(1)	6	(2)	3
(3)	0	(4)	46

2. Модуль № 2. Логические выражения и условные операторы

2.1. Комплексная цель

Освоить операции сравнения и логические операции, научиться составлять с их помощью сложные логические выражения. Изучить условный оператор *if* и оператор выбора *switch*, а также варианты их применения, в том числе каскадные условные операторы. Научиться составлять простейшие разветвляющиеся алгоритмы, в том числе алгоритм определения минимального/максимального из двух и трех чисел, а также его номера, алгоритм проверки упорядоченности трех чисел, алгоритм вычисления функции, заданной различными формулами на различных участках своей области определения, и т. п.

2.2. Содержание

2.2.1. Логические выражения

Особым типом данных являются данные *логического типа*, описываемые с помощью ключевого слова *bool*. Эти данные могут принимать только два значения: *true* (истина) или *false* (ложь).

При составлении логических выражений обычно используются операции отношения и логические операции.

К *операциям отношения* относятся следующие: «равно» ==, «не равно» !=, «меньше» <, «больше» >, «меньше или равно» <=, «больше или равно» >=. Операндами этих операций обычно являются выражения числовых типов (хотя допустимо применять эти операции и к данным других типов, например, к указателям), а результатом операций является значение типа *bool*. Например, выражение 1 < 2 имеет значение *true*, а 2 == 3 — значение *false*. Выражение *a* != 4 принимает значение *false*, если значение переменной *a* равно 4, и *false* в противном случае.

Основными *логическими операциями* являются следующие: «отрицание» !, «логическое И» &&, «логическое ИЛИ» ||. Выражение !*a* равно *true*, если операнд *a* имеет значение *false*, и *false*, если операнд *a* имеет значение *true*. Выражение *a* && *b* равно *true* только в случае, если оба операнда (*a* И *b*) имеют значение *true*. Выражение *a* || *b* равно *true*, если хотя бы один операнд (*a* ИЛИ *b*) имеет значение *true*.

В языке C++ операции отношения имеют более высокий приоритет, чем логические операции, поэтому скобки в сложных логических выражениях, как правило, не требуются. Например, выражение 2 < 3 && 4 < 5 эквивалентно выражению (2 < 3) && (4 < 5).

Среди логических операций наивысший приоритет имеет операция отрицания, за ней следует операция && («И»), затем операция || («ИЛИ»). Таким образом, если предположить, что *a*, *b* и *c* являются переменными логического типа, то в выражении *a* || !*b* && *c* скобки по умолчанию расставляются следующим образом: *a* || ((!*b*) && *c*).

☑ В C++ логические операции && и || выполняются *по сокращенной схеме*. Это означает, что данные операции могут возвращать результат, не вычисляя второй операнд, если этот результат однозначно определяется по первому операнду. Для операции && второй операнд не вычисляется, если первый операнд равен *false* (в качестве результата возвращается *false*); для операции || второй операнд не вычисляется, если первый равен *true* (в качестве результата возвращается *true*).

Рассмотрим две простые задачи на логические выражения.

Boolean2. Дано целое число *A*. Проверить истинность высказывания: «Число *A* является нечетным».

```
task("Boolean2");
int a;
pt >> a;
pt << (a % 2 != 0);
```

Требуемое логическое выражение можно указать непосредственно в виде операнда операции вывода <<, поэтому в программе не требуется описывать переменную логического типа. Использование скобок в последнем операторе является обязательным, так как операция << имеет более высокий приоритет, чем операции сравнения (в частности, операция !=).

При решении задания мы воспользовались тем фактом, что нечетные числа при деления на 2 дают ненулевой остаток.

☑ В C++ числовые выражения можно преобразовывать в логические; при этом нулевое выражение преобразуется в значение *false*, а ненулевое выражение — в значение *true*. Для подобного преобразования можно использовать *операцию приведения типа* *static_cast<тип>*(*выражение*). Используя этот факт, последний оператор решения задачи Boolean2 можно записать следующим образом:

```
pt << static_cast<bool>(a % 2);
```

С другой стороны, логические константы *true* и *false* можно преобразовать к целочисленному типу; при этом *static_cast<int>*(*false*) будет равно 0, а *static_cast<int>*(*true*) будет равно 1.

Заметим, однако, что злоупотреблять операциями приведения типа не следует, поскольку они приводят к усложнению кода и тем самым затрудняют его восприятие.

Boolean7. Даны три целых числа: *A*, *B*, *C*. Проверить истинность высказывания: «Число *B* находится между числами *A* и *C*».

```
task("Boolean7");
int a, b, c;
pt >> a >> b >> c;
pt << (a < b && b < c || a > b && b > c);
```

Заметим, что для решения задачи можно использовать более простое логическое выражение, поскольку достаточно проверить, что выражения $a - b$ и $b - c$ имеют один и тот же знак. Приведем последний оператор этого варианта решения:

```
pt << ((a - b) * (b - c) > 0);
```

2.2.2. Условный оператор

Условный оператор позволяет выполнить одно действие из двух в зависимости от истинности некоторого условия. Приведем общий вид условного оператора для языка C++, заключая в квадратные скобки те его элементы, которые могут отсутствовать:

```
if (условие)
    оператор_1
[else
    оператор_2]
```

Оператор_1 выполняется, если *условие* является истинным. Раздел *else* является необязательным; при его наличии *оператор_2* выполняется, если *условие* является ложным. Если в условном операторе требуется выполнить не один, а несколько операторов, то их надо превратить в один *составной оператор*, заключив требуемые операторы в операторные скобки {}. Следует обратить внимание на то, что после составного оператора {} (в отличие от любых других видов операторов) точка с запятой *не ставится*.

☑ В C++ условие в операторе *if* может представлять собой не только логическое, но и *числовое* выражение (а также выражение типа указателя). В этом случае оно неявно преобразуется к логическому типу по правилам, приведенным в последнем примечании предыдущего пункта, а именно — нулевое выражение преобразуется к логическому значению *false*, а ненулевое — к логическому значению *true*.

Оператор *if*, содержащий раздел *else*, называется *полным условным оператором* (или условным оператором *в полной форме*), а оператор *if* без раздела *else* — *сокращенным условным оператором* (или условным оператором *в краткой форме*).

Использование условного оператора проиллюстрируем на примере решения задания If6.

If6. Даны два числа. Вывести большее из них.

```
task("If6");
double a, b, max;
pt >> a >> b;
if (a > b)
    max = a;
else
    max = b;
pt << max;
```

Приведем другой, более короткий вариант решения, в котором используется *условная операция* ?:.

```
task("If6");
double a, b;
pt >> a >> b;
pt << (a > b ? a : b);
```

Если первый операнд условной операции ?: равен *true*, то возвращается второй операнд, иначе возвращается третий операнд. Следует заметить, что если второй и/или третий операнды являются выражениями, то вычисляется только то выражение, значение которого будет возвращено данной операцией. Как и в случае условия оператора *if* (см. последнее примечание), первый операнд условной операции при необходимости автоматически преобразуется к логическому типу.

Рассмотрим еще две задачи на условный оператор.

If12. Даны три числа. Найти наименьшее из них.

```
task("If12");
double a, b, c;
pt >> a >> b >> c;
double min = a;
if (b < min)
    min = b;
```

```
if (c < min)
    min = c;
pt << min;
```

В данном решении используется *краткая форма* условного оператора (без раздела *else*). Заметим, что использованный алгоритм нахождения минимального элемента можно обобщить на любое количество исходных данных (см. решение задания Minmax1 в п. 4.2.2).

If26. Для данного вещественного x найти значение следующей функции f, принимающей вещественные значения:

$$f(x) = \begin{cases} -x, & \text{если } x \le 0, \\ x^2, & \text{если } 0 < x < 2, \\ 4, & \text{если } x \ge 2. \end{cases}$$

При решении этой задачи возникает *каскад* условных операторов:

```
task("If26");
double x, y;
pt >> x;
if (x <= 0)
    y = -x;
else if (x < 2)
    y = x * x;
else
    y = 4;
pt << y;
```

Раздел *else* первого условного оператора содержит второй условный оператор (обратите внимание на то, что в каскаде условных операторов слово *if* и условие очередного условного оператора располагаются на той же строке, что и слово *else* предыдущего, а дополнительные отступы не используются). Поскольку перейти на второй условный оператор можно только в случае, когда значение x является положительным, для проверки того, что x лежит в интерва-

ле (0, 2), достаточно выполнить единственное сравнение: $x < 2$. Если и эта проверка возвращает *false*, значит, число x удовлетворяет условию $x >= 2$, и поэтому переменной y следует присвоить значение 4.

☑ Другая ситуация возникает, когда второй условный оператор нужно написать в разделе *if* первого. При этом надо учитывать следующее правило языка C++: *раздел else полного условного оператора соответствует ближайшему разделу if, у которого раздел else отсутствует*. Так, в следующем фрагменте программы раздел *else* относится ко второму (то есть внутреннему) условному оператору, что желательно подчеркнуть с помощью соответствующих отступов:

```
c = 0;
if (a < 2)
    if (b < 3)
        c = 4;
    else
        c = 5;
```

Если же раздел *else* в приведенном фрагменте следует связать в внешним условным оператором, то необходимо заключить внутренний оператор *if* в операторные скобки {}:

```
c = 0;
if (a < 2)
{
    if (b < 3)
        c = 4;
}
else
    c = 5;
```

Ясно, что приведенные фрагменты будут выполняться по-разному. Так, если *a* равно 4, а *b* равно 5, то после выполнения первого фрагмента переменная *c* сохранит начальное значение, равное 0, а после выполнения второго — изменит свое значение на 5.

2.2.3. Оператор выбора

Оператор выбора позволяет выполнить одно действие из нескольких возможных в зависимости от значения некоторого выражения, называемого *переключателем*. Приведем вид оператора выбора для языка C++:

```
switch (переключатель)
{
    case знач_1:
        операторы_1
        [break;]
    case знач_2:
        операторы_2
        [break;]
    ...
    [default:
        операторы_0]
}
```

Если значение *переключателя* равно *знач_1*, то выполняются операторы группы 1, если равно *знач_2* — выполняются операторы группы 2, и т. д. Если в операторе выбора указан *вариант по умолчанию default*, то он выполняется, если значение переключателя не равно ни одному из явно указанных значений. В качестве переключателя можно использовать выражение целого типа; возможные значения переключателя, указываемые в разделах *case*, могут быть только *константными* выражениями. Можно объединять значения, соответствующие одной и той же группе операторов; для этого следует указывать подряд несколько *case*-выражений (например, *case* 3: *case* 5: *case* 6:).

☑ Вариант *default* не обязательно указывать *в конце* оператора *switch*, хотя эта позиция для данного варианта представляется наиболее естественной.

После завершения каждой группы операторов, входящей в оператор выбора обычно указывается оператор *break*, позволяющий немедленно выйти из оператора выбора. Если после какой-либо группы не указать оператор *break*, то произойдет переход на следующую группу, указанную в этом же операторе выбора (подобное явление называется «проваливанием на следующую метку»).

Использование оператора выбора проиллюстрируем на примере решения задания Case4.

Case4. Дан номер месяца — целое число в диапазоне 1–12 (1 — январь, 2 — февраль и т. д.). Определить количество дней в этом месяце для невисокосного года.

```
task("Case4");
int n, k;
pt >> n;
switch (n)
{
    case 2:
        k = 28; break;
    case 4: case 6:
    case 9: case 11:
        k = 30; break;
    default:
        k = 31;
}
pt << k;
```

В данном решении используется оператор выбора, содержащий список *case*-выражений (для месяцев, насчитывающих 30 дней), а также вариант по умолчанию (для месяцев, насчитывающих 31 день).

Рассмотрим также задание группы Case, при решении которого приходится использовать и оператор выбора, и условный оператор.

Case9. Даны два целых числа: *D* (день) и *M* (месяц), определяющие правильную дату невисокосного года. Вывести значения *D* и *M* для даты, следующей за указанной.

```
task("Case9");
int d, m, k;
pt >> d >> m;
switch (m)
{
    case 2:
        k = 28; break;
    case 4: case 6:
    case 9: case 11:
        k = 30; break;
    default:
        k = 31;
}
if (d < k)
    ++d;
else
{
    d = 1;
    ++m;
    if (m == 13)
        m = 1;
}
pt << d << m;
```

Первая часть программы определяет число дней в месяце *m*, поэтому она почти дословно повторяет текст предыдущей программы.

Для увеличения значения переменной *d* на единицу используется операция *инкремента* ++. Дополнительные особенности операции инкремента ++ и об-

ратной к ней операции *декремента* -- обсуждаются в п. 3.3.2 (см. решение задания While11).

☑ При решении последней задачи можно допустить весьма опасную (хотя и типичную) ошибку, связанную с условным оператором, а именно, указать в его условии вместо знака равенства == знак присваивания =:

```
if (m = 13)
    m = 1;
```

Подобная конструкция не будет считаться ошибочной. Действительно, операция присваивания *m* = 13 не только записывает значение 13 в переменную *m*, но и *возвращает* это значение. Таким образом, условие принимает целочисленное значение 13, которое автоматически преобразуется к логическому значению *true*. Поэтому указанный фрагмент не будет приводить к ошибке компиляции или выполнения, однако при его выполнении в переменную *m всегда* будет записываться значение 1 (так как условие оператора *if* всегда будет иметь значение *true*).

Некоторые компиляторы (например, Borland C++Builder) в подобной ситуации выводят предупреждающее сообщение «Possibly incorrect assignment» («Возможно, неверное присваивание»). Для того чтобы аналогичное предупреждения выводил компилятор системы Visual Studio, необходимо вызвать окно свойств проекта, используя последнюю команду меню «Project», перейти в этом окне в раздел «Configuration Properties | C/C++ | General» и установить для свойства «Warning Level» значение «Level 4». В этом случае при компиляции приведенного выше ошибочного фрагмента будет выведено предупреждающее сообщение «Assignment within conditional expression» («Присваивание в условном выражении»).

Еще один способ избежать подобной ошибки — указывать в левой части операции сравнения на равенство не переменную, а *константу* или *константное выражение*:

```
if (13 == m)
    m = 1;
```

При этом в случае пропуска в операции сравнения одного символа = гарантированно возникнет ошибка компиляции.

2.3. Проектное задание

Выполните задачи групп Boolean, If и Case, указанные в вашем варианте проектного задания (формулировки задач приводятся в [1]). Если вы не получили вариант проектного задания, то выполните задачи из какого-либо варианта.

ВАРИАНТ 1 Boolean: 5, 19, 26, 40 If: 4, 11, 21, 30 Case: 3, 8, 19	**ВАРИАНТ 2** Boolean: 17, 18, 29, 35 If: 1, 8, 20, 25 Case: 7, 10, 17	**ВАРИАНТ 3** Boolean: 10, 23, 28, 37 If: 5, 9, 18, 25 Case: 2, 14, 20
ВАРИАНТ 4 Boolean: 11, 22, 26, 38 If: 5, 8, 19, 24 Case: 2, 11, 20	**ВАРИАНТ 5** Boolean: 1, 22, 27, 38 If: 1, 7, 17, 29 Case: 6, 13, 18	**ВАРИАНТ 6** Boolean: 4, 20, 25, 35 If: 2, 14, 18, 27 Case: 1, 14, 15
ВАРИАНТ 7 Boolean: 8, 20, 29, 40 If: 4, 7, 19, 28 Case: 3, 10, 17	**ВАРИАНТ 8** Boolean: 6, 21, 25, 39 If: 5, 9, 21, 30 Case: 6, 8, 19	**ВАРИАНТ 9** Boolean: 6, 18, 27, 37 If: 3, 14, 17, 24 Case: 7, 11, 15
ВАРИАНТ 10 Boolean: 3, 21, 29, 39 If: 5, 10, 16, 29 Case: 5, 13, 18	**ВАРИАНТ 11** Boolean: 16, 23, 29, 36 If: 3, 11, 16, 28 Case: 5, 12, 16	**ВАРИАНТ 12** Boolean: 9, 19, 28, 36 If: 2, 10, 20, 27 Case: 1, 12, 16
ВАРИАНТ 13 Boolean: 6, 19, 26, 38	**ВАРИАНТ 14** Boolean: 1, 20, 25, 37	**ВАРИАНТ 15** Boolean: 5, 18, 29, 36

If: 5, 14, 20, 28 Case: 1, 8, 20	If: 1, 7, 20, 30 Case: 2, 14, 15	If: 2, 9, 17, 27 Case: 3, 13, 18
ВАРИАНТ 16 Boolean: 10, 23, 29, 36 If: 3, 10, 21, 27 Case: 5, 14, 15	**ВАРИАНТ 17** Boolean: 17, 21, 26, 39 If: 3, 8, 17, 24 Case: 1, 11, 19	**ВАРИАНТ 18** Boolean: 6, 19, 27, 35 If: 4, 10, 21, 25 Case: 7, 11, 19
ВАРИАНТ 19 Boolean: 4, 18, 25, 40 If: 1, 9, 18, 25 Case: 6, 8, 16	**ВАРИАНТ 20** Boolean: 8, 21, 27, 40 If: 2, 11, 19, 30 Case: 5, 12, 18	**ВАРИАНТ 21** Boolean: 3, 22, 28, 37 If: 4, 11, 16, 28 Case: 6, 10, 17
ВАРИАНТ 22 Boolean: 9, 22, 28, 38 If: 5, 14, 18, 29 Case: 7, 12, 17	**ВАРИАНТ 23** Boolean: 11, 20, 29, 35 If: 5, 7, 16, 24 Case: 2, 10, 16	**ВАРИАНТ 24** Boolean: 16, 23, 29, 39 If: 5, 8, 19, 29 Case: 3, 13, 20

2.3.1. Указания к заданиям группы Boolean

Boolean1. Если исходное число содержится в переменной *a*, то достаточно вывести значение логического выражения $a > 0$.

Boolean2–3. Решение Boolean2 приводится в п. 2.2.1; задание Boolean3 решается аналогично, с заменой операции сравнения != «не равно» на операцию == «равно».

Boolean4–5. В Boolean4 используйте операцию && («И»), в Boolean5 используйте операцию || («ИЛИ»).

Boolean6. Двойное неравенство $A < B < C$ означает, что $A < B$ и $B < C$. Данное условие на языке C++ имеет вид $A < B$ && $B < C$.

Boolean7. Решение этого задания приводится в п. 2.2.1.

Boolean8–11. По поводу проверки четности см. решение Boolean2. В Boolean8 используйте операцию &&, в Boolean9 — операцию ||. В Boolean10–11 достаточно проанализировать четность *суммы* исходных чисел.

Boolean12–15. В Boolean12 используйте операцию &&, в Boolean13 — операцию ||. В Boolean14–15 используйте логические выражения, содержащие обе эти операции.

Boolean16–17. По поводу проверки четности см. решение Boolean2. При определении количества знаков достаточно учесть, что двузначные числа лежат в диапазоне 10–99, а трехзначные — в диапазоне 100–999.

Boolean18–19. При выполнении этих заданий можно обойтись без логических операций. Например, в Boolean18 достаточно проанализировать выражение $(A - B)\cdot(B - C)\cdot(C - A)$. В Boolean19 надо перемножать не разности, а суммы исходных чисел.

Boolean20–23. По поводу нахождения цифр, входящих в целое число, см. решения заданий Integer8 и Integer11 в п. 1.2.3. Ср. Boolean20 с Boolean18, Boolean21 с Boolean6, Boolean22 с Boolean7.

Boolean25–28. В Boolean25–26 используйте логическую операцию &&; в Boolean27–28 можно обойтись без использования логических операций.

Boolean29. Достаточно проверить, что одновременно выполняются два двойных неравенства: $x_1 < x < x_2$ и $y_2 < y < y_1$ (по поводу проверки двойных неравенств см. указание к Boolean6).

Boolean31–32. Ср. Boolean31 с Boolean18. В Boolean32 достаточно проверить, что квадрат какой-либо стороны равен сумме квадратов двух других сторон.

Boolean34–35. Цвет клетки с координатами (x, y) зависит от *четности* выражения $x + y$. По поводу проверки четности см. решение Boolean2.

Boolean36–40. При перемещении *ладьи* одна из ее координат не изменяется. При перемещении *короля* его координаты изменяются (по модулю) не более чем на 1: $|x_1 - x_2| \le 1$ *и* $|y_1 - y_2| \le 1$. *Слон* ходит по диагонали, поэтому каждая его координата должна изменяться на одинаковое (по модулю) число. *Ферзь* ходит как ладья и как слон. При перемещении *коня* одна из его координат изменяется (по модулю) на 1, а другая — на 2.

2.3.2. Указания к заданиям группы If

If1–3. Простейшие задания на применение условного оператора. В If1 используйте *краткую форму* условного оператора (без раздела *else*), в If2 — *полную форму*, в If3 — *каскад условных операторов*.

If4–5. Задание If4 решается с использованием одной переменной-счетчика, а задание If5 — с использованием двух.

If6–8. Решение If6 приводится в п. 2.2.2; прочие задания решаются аналогично.

If9. Ср. с Begin22.

If12–15. Решение If12 приводится в п. 2.2.2, аналогично находится наибольшее число в If14. Для нахождения среднего числа (If13) или двух наибольших чисел (If15) можно использовать либо сложные условия с логическими операциями, либо вложенные условные операторы.

If16–17. В If16 достаточно проверить, что исходные числа A, B, C удовлетворяют двойному неравенству $A < B < C$ (см. указание к Boolean6), в If17 достаточно проверить, что B находится между A и C (см. решение Boolean7 в п. 2.2.1).

If18–19. Если использовать вложенные условные операторы, то в каждом из этих заданий для определения требуемого номера будет достаточно вы-

полнить *две* операции сравнения (в If18 в некоторых случаях ответ можно получить после *одной* операции).

If20. По поводу нахождения расстояния между точками на числовой оси см. задание Begin16 и указание к нему.

If21–22. См. указание к If18–19.

If23. Ср. с If18. В данном случае необходимо проанализировать *по отдельности* абсциссы и ординаты трех данных точек и найти абсциссу и ординату, отличную от двух других (равных между собой).

If24–25. В решении достаточно использовать один полный условный оператор. В If25 можно обойтись без логической операции ||, если воспользоваться функцией *abs*(*x*), возвращающей модуль числа *x*.

If26. Решение этого задания приводится в п. 2.2.2.

If27. В случае неотрицательного *x* достаточно найти его *целую часть* и проанализировать ее *четность*. Для нахождения целой части неотрицательного числа достаточно преобразовать его к целому типу (см. п. 5.2). По поводу проверки четности см. решение Boolean2 в п. 2.2.1.

If28. В условии, отбирающем високосные годы, необходимо использовать операцию взятия остатка от деления (на 4, на 100 и на 400).

If29–30. В программе следует отдельно сформировать первую и вторую половины строки-описания, сохранив их в переменных типа *string*, после чего объединить их операцией сцепления +. Например, в If29 надо по отдельности проанализировать *четность* и *знак* данного числа (по поводу проверки четности см. решение Boolean2 в п. 2.2.1). Завершающая часть строки-описания не зависит от значения исходного числа, поэтому ее можно добавлять к результирующей строке после выхода из условного оператора. Между словами в строке-описании должно располагаться по *одному* пробелу.

2.3.3. Указания к заданиям группы Case

Case2. Используйте оператор выбора с *вариантом по умолчанию default*.

Case3. Используйте несколько case-выражений, соответствующих одному и тому же времени года, например, *case* 3: *case* 4: *case* 5: (весна).

Case4. Решение этого задания приводится в п. 2.2.3.

Case8–9. Решение Case9 приводится в п. 2.2.3. Задание Case8 решается аналогично; в нем также следует использовать вспомогательную переменную *k*, но уже для определения числа дней в *предыдущем* месяце (необходимость в этой переменной возникает при $D = 1$).

Case10–11. При выполнении этих заданий удобно перейти (используя оператор выбора) от символьных обозначений сторон света к числовым: 0 — север, 1 — запад, 2 — юг, 3 — восток. Теперь для получения нового направления достаточно *прибавить* значение команды (или обеих команд для Case11) к исходному значению стороны света и вернуться к символьному значению для найденной стороны света (еще раз используя оператор выбора). Необходимо также учитывать, что полученное число может лежать вне диапазона 0–3; например, результирующему северному направлению может соответствовать не только число 0, но и число 4, восточному направлению — не только 3, но и –1 и т. д.

Case15–19. Ср. с If29–30. Для выделения цифр из данного числа (в Case16–18) используйте операции целочисленного деления и взятия остатка — см. решения заданий Integer8 и Integer11 в п. 1.2.3. Эти же операции надо использовать в Case19 для определения номера подцикла и порядкового номера года в пределах подцикла. Во всех этих заданиях надо использовать *несколько* операторов выбора; например, в Case16 требуются три оператора выбора: для определения названий десятков, единиц и заключительного

слова (в зависимости от исходного числа это будет либо «год», либо «года», либо «лет»).

2.4. Тест рубежного контроля

Тест содержит 6 заданий, на выполнение которых отводится 3 минуты. Выберите правильный, по вашему мнению, вариант ответа и отметьте его любым значком в бланке ответов.

1. Даны три логические переменные: *a* = *false*, *b* = *false*, *c* = *true*. Укажите значения логических переменных *d1* и *d2* после выполнения следующих операторов:

```
d1 = b || !c && a;
d2 = !a || b && c;
```

(1)	*d1* = *true*, *d2* = *true*	(2)	*d1* = *true*, *d2* = *false*
(3)	*d1* = *false*, *d2* = *true*	(4)	*d1* = *false*, *d2* = *false*

2. Даны целые числа *a* и *b*. Укажите, какое условие проверяется следующим выражением:

```
(a % 2 == 0) && !(b % 2 == 0)
```

(1)	Число *a* является нечетным, *b* — четным	(2)	Число *b* является нечетным, *a* — четным
(3)	Хотя бы одно из чисел является четным	(4)	Оба числа являются четными

3. Даны три переменные типа *int*: *a* = 4, *b* = 5, *c* = 6. Укажите, какое значение получит переменная *c* после выполнения следующего фрагмента программы:

```
if (a < 5)
    if (b < 5)
      ++c;
else --c;
```

(1)	4	(2)	5
(3)	6	(4)	7

4. Дан фрагмент программы:

```
int a = 3, b = 2;
if (a / b - 1 < 0)
    a = b + 2;
    b = a + 1;
else
    b = a + 1;
    a = b + 2;
cout << a << " " << b;
```

Укажите, что произойдет при обработке этого фрагмента.

(1)	Возникнет ошибка компиляции	(2)	Возникнет ошибка времени выполнения
(3)	Будут напечатаны числа 4 и 5	(4)	Будут напечатаны числа 6 и 4

5. Укажите неверное утверждение.

(1)	Раздел *default* в операторе *switch* можно не указывать	(2)	Раздел *default* в операторе *switch* должен располагаться после всех разделов *case*
(3)	В разделе *default* оператора *switch* допустимо не указывать оператор *break*	(4)	В разделах *case* оператора *switch* допустимо не указывать оператор *break*

6. Дан фрагмент программы:

```
int a = 3;
switch (a)
{
    case 1:
        ++a;
    case 3: case 7:
        a += 2;
    default:
        a += 2;
        break;
}
cout << a;
```

Укажите, что произойдет при обработке этого фрагмента.

(1)	Произойдет ошибка компиляции	(2)	Будет напечатано число 3
(3)	Будет напечатано число 5	(4)	Будет напечатано число 7

3. Модуль № 3. Операторы цикла

3.1. Комплексная цель

Изучить различные варианты операторов цикла в C++ (*for*, *while*, *do while*); ознакомиться с особенностями каждого варианта. Освоить базовые итерационные алгоритмы, в том числе нахождение суммы и произведения, применение переменных-счетчиков, вычисление по рекуррентным формулам. Научиться использовать вложенные циклы.

3.2. Содержание

3.2.1. Цикл с параметром

Цикл с параметром используется в ситуации, когда число повторений (*итераций*) заранее известно. В языке C++ для организации цикла с параметром используется цикл *for*, хотя его возможности гораздо богаче, чем возможности «обычного» цикла с параметром. Приведем наиболее распространенный вариант цикла *for*, соответствующий циклу с возрастающим параметром:

```
for ([тип] парам = знач_1; парам <= знач_2; ++парам)
    оператор
```

Здесь *парам* обозначает переменную — *параметр цикла*, *тип* — тип данной переменной. Обычно в качестве типа параметра цикла используется целый тип; допустимым является и вещественный тип, хотя его использовать не рекомендуется (см. далее решение задания For5). *Знач_1* и *знач_2* определяют соответственно начальное и конечное значение параметра цикла; шаг изменения параметра равен 1.

Тело цикла (один оператор) выполняется для каждого значения параметра цикла. Если начальное значение больше конечного, то тело цикла ни разу не выполняется. Если требуется повторить в цикле несколько операторов, то их надо объединить в один составной оператор с помощью скобок {}.

Явное указание типа для параметра цикла означает, что этот параметр будет существовать *только при выполнении данного цикла*; при этом требуется, чтобы к моменту начала цикла переменная с таким именем не существовала. Рекомендуется всегда описывать параметр цикла в самом цикле, поскольку это исключает случайное использование данного параметра после выхода из цикла, и в настоящем пособии мы *всегда* будем следовать этой рекомендации.

Хотя параметр цикла не запрещено изменять в теле цикла, делать это не рекомендуется, так как подобное изменение параметра цикла, как правило, де-

лает алгоритм сложным для восприятия и может приводить к появлению трудно выявляемых ошибок.

Следует учитывать важную особенность, отличающую приведенный вариант цикла *for* для языка C++ от циклов с параметром, реализованных в других распространенных языках программирования (например, в языке Pascal или Visual Basic): в C++ выражение *знач_2* вычисляется *заново* на каждой итерации перед сравнением с текущим значением параметра. Для того чтобы эта особенность не привела к ошибочной работе цикла, желательно не изменять в теле цикла переменные, входящие в выражение *знач_2*.

Иногда бывает удобно организовать перебор значений параметра цикла *по убыванию* (при этом начальное значение должно быть *больше* конечного значения). Приведем вариант подобного цикла, предполагая, что шаг изменения параметра цикла равен –1.

```
for ([тип] парам = знач_1; парам >= знач_2; --парам)
    оператор
```

Разумеется, можно указать и другие варианты шага изменения параметра цикла; для этого надо указать соответствующий оператор в последнем разделе заголовка цикла (например, *парам* += 2).

Для досрочного выхода из цикла предусмотрен оператор *break*, а для досрочного прекращения *текущей итерации* цикла — оператор *continue*.

Проиллюстрируем использование цикла с параметром на примере решения нескольких заданий группы For.

For5. Дано вещественное число — цена 1 кг конфет. Вывести стоимость 0.1, 0.2, …, 1 кг конфет.

При решении данного задания можно использовать цикл с *вещественным* параметром и явно указанным шагом, равным 0.1.

```
task("For5");
double a;
```

```
pt >> a;
for (double i = 0.1; i <= 1; i += 0.1)
    pt << i * a;
```

Заметим, однако, что использование вещественного параметра цикла может приводить к неправильному определению числа итераций из-за погрешностей округления. Безопаснее использовать *целочисленный* параметр цикла, по которому в теле цикла *вычислять* требуемое вещественное значение:

```
task("For5");
double a;
pt >> a;
for (int i = 1; i <= 10; ++i)
    pt << 0.1 * i * a;
```

For12. Дано целое число N (> 0). Найти произведение $1.1 \cdot 1.2 \cdot 1.3 \cdot \ldots$ (N сомножителей).

Здесь, как и во втором варианте решения задания For5, будем использовать цикл с целочисленным параметром, вычисляя на основе текущего значения параметра очередной сомножитель требуемого произведения.

```
task("For12");
int n;
pt >> n;
double a = 1;
for (int i = 1; i <= n; ++i)
    a *= 1 + 0.1 * i;
pt << a;
```

Так как в переменной *a* будет вычисляться *произведение* чисел, данная переменная инициализируется числом 1. Обратите внимание на оператор *=, используемый для накопления произведения.

Аналогично решается и задание For13.

For13. Дано целое число N (> 0). Найти значение выражения 1.1 – 1.2 + 1.3 – … (N слагаемых, знаки чередуются). Условный оператор не использовать.

```
task("For13");
int n, k = 1;
pt >> n;
double a = 0;
for (int i = 1; i <= n; ++i)
{
    a += k * (1 + 0.1 * i);
    k = -k;
}
pt << a;
```

Здесь вычисляется *сумма* элементов, поэтому переменная суммирования *a* инициализируется значением 0. Смену знака у слагаемых обеспечивает вспомогательный множитель *k*, принимающий попеременно значения 1 и –1. Поскольку тело цикла состоит из двух операторов, приходится заключать их в операторные скобки {}.

For16. Дано вещественное число A и целое число N (> 0). Используя один цикл, вывести все целые степени числа A от 1 до N.

```
task("For16");
double a, p = 1;
int n;
pt >> a >> n;
for (int i = 1; i <= n; ++i)
{
    p *= a;
    pt << p;
}
```

В данном решении в цикле последовательно вычисляются все степени числа *a*, поэтому и вывод степеней выполняется в том же цикле.

☑ В заголовочном файле *cmath* стандартной библиотеки языка C++ имеется функция *pow*(*a*, *b*), обеспечивающая возведение вещественного числа *a* в целую или вещественную степень *b* (если *b* не является целым числом, то *a* должно быть положительным). Однако, поскольку в заданиях группы For требуется продемонстрировать умение работать с циклами, возведение в степень в этих заданиях надо проводить с помощью последовательного умножения в цикле. Данная рекомендация сохраняет силу и для «обычных» (не учебных) программ, если показателем степени является небольшое целое число, а к быстродействию кода предъявляются повышенные требования. Заметим, что для подключения к программе заголовочного файла *cmath* в ее начало надо добавить следующую директиву:

```
#include <cmath>
```

В системе Borland C++Builder данная директива должна располагаться *после* директивы

```
#include <vcl.h>
```

For20. Дано целое число N (> 0). Используя один цикл, найти сумму

$$1! + 2! + 3! + \ldots + N!$$

(выражение $N!$ — *N–факториал* — обозначает произведение всех целых чисел от 1 до N: $N! = 1 \cdot 2 \cdot \ldots \cdot N$). Чтобы избежать целочисленного переполнения, проводить вычисления с помощью вещественных переменных и вывести результат как вещественное число.

```
task("For20");
int n;
pt >> n;
double p = 1, s = 0;
for (int i = 1; i <= n; ++i)
{
    p *= i;
    s += p;
}
```

```
pt << s;
```

При выполнении цикла в переменной *p* последовательно вычисляются (и добавляются к сумме *s*) все требуемые факториалы.

For33. Дано целое число N (> 1). Последовательность *чисел Фибоначчи* F_K (целого типа) определяется следующим образом:

$$F_1 = 1, \qquad F_2 = 1, \qquad F_K = F_{K-2} + F_{K-1}, \quad K = 3, 4, \ldots .$$

Вывести элементы $F_1, F_2, ..., F_N$.

```
task("For33");
int n;
pt >> n;
int a, b;
a = b = 1;
pt << a << b;
for (int i = 3; i <= n; ++i)
{
    int c = a + b;
    pt << c;
    a = b;
    b = c;
}
```

При выполнении этого задания приходится использовать две вспомогательные переменные *a* и *b*, в которых хранятся два предыдущих элемента последовательности Фибоначчи. С их помощью вычисляется очередной элемент (*c*), после чего происходит корректировка значений вспомогательных переменных.

Обратите внимание на оператор *a* = *b* = 1. Данный оператор позволяет *одновременно* инициализировать переменные *a* и *b* значением 1. Это оказывается возможным благодаря тому, что операция присваивания *b* = 1 сама *возвращает значение*, равное значению ее правой части; поэтому это значение можно ис-

пользовать в качестве правой части для другой операции присваивания. Заметим, что операция присваивания является *правоассоциативной*, то есть скобки в ней по умолчанию расставляются справа налево: $a = (b = 1)$.

☑ Использование возвращаемых значений для операции присваивания часто приводит к сложному для восприятия программному коду, поэтому рекомендуется применять эту возможность *только для одновременного присваивания одного и того же значения нескольким переменным*.

Следующее задание демонстрирует использование *вложенных* циклов.

For36. Даны целые положительные числа N и K. Найти сумму $1^K + 2^K + \ldots + N^K$. Чтобы избежать целочисленного переполнения, вычислять слагаемые этой суммы с помощью вещественной переменной и выводить результат как вещественное число.

```
task("For36");
int n, k;
pt >> n >> k;
double s = 0;
for (int i = 1; i <= n; ++i)
{
    double p = 1;
    for (int j = 1; j <= k; ++j)
        p *= i;
    s += p;
}
pt << s;
```

Во внутреннем цикле (по параметру j) в переменной p вычисляется k-я степень числа i, которая затем добавляется к сумме s. Обратите внимание на то, что переменная внутреннего цикла (в нашем случае j) обязательно должна отличаться от переменной внешнего цикла (в нашем случае i).

3.2.2. Цикл с условием

Цикл с условием обычно используется в ситуациях, когда заранее неизвестно требуемое количество итераций. Различают два варианта цикла с условием: *цикл с предварительным условием* и *цикл с последующим условием*. Вначале приведем общий вид цикла с предварительным условием:

```
while (условие)
    оператор
```

Цикл работает, пока *условие* остается истинным. Заметим, что если уже при первой проверке *условие* оказывается ложным, то не выполняется ни одной итерации цикла.

Цикл с последующим условием имеет вид

```
do
    оператор
while (условие);
```

Цикл работает, пока *условие* остается истинным, причем первая проверка условия выполняется уже *после* выполнения оператора, указанного в теле цикла, поэтому при использовании данного варианта цикла обязательно выполняется *хотя бы одна* его итерация (в этом состоит основное отличие цикла с последующим условием от цикла с предварительным условием).

Как и в случае цикла *for*, тело любого варианта цикла с условием должно состоять из одного оператора. Поэтому если требуется повторить в цикле несколько операторов, их надо объединить в один составной оператор в помощью скобок {}.

Для досрочного выхода из цикла с условием, как и для выхода из цикла *for*, надо использовать оператор *break*.

☑ Условие в операторе цикла, как и в операторе *if*, может представлять собой не только логическое, но и *числовое* выражение (которое неявно преобразуется к логическому типу).

Приведем решения нескольких заданий группы While, посвященной циклу с условием.

While1. Даны положительные числа A и B ($A > B$). На отрезке длины A размещено максимально возможное количество отрезков длины B (без наложений). Не используя операции умножения и деления, найти длину незанятой части отрезка A.

```
task("While1");
double a, b;
pt >> a >> b;
while (a > b)
    a -= b;
pt << a;
```

Приведенный алгоритм фактически является алгоритмом нахождения остатка при делении вещественного числа *a* на вещественное число *b*. Напомним, что операция % нахождения остатка от деления в данном случае неприменима, так как ее операндами могут быть только целые числа.

While4. Дано целое число N (> 0). Если оно является степенью числа 3, то вывести True, если не является — вывести False.

```
task("While4");
int n, p = 1;
pt >> n;
while (p < n)
    p *= 3;
pt << (p == n);
```

В цикле *while* последовательно вычисляются степени числа 3 до тех пор, пока найденная степень не станет большей или равной числу *n*. После этого в случае равенства выводится *true*, а в случае неравенства — *false*.

Приведем другой способ решения задания While4, основанный на последовательном *делении* исходного числа на 3 и анализе полученного остатка.

```
task("While4");
int n;
pt >> n;
while (n % 3 == 0)
    n /= 3;
pt << (n == 1);
```

При обработке чисел, *не являющихся* степенями 3, во втором варианте решения потребуется меньше итераций цикла, чем в первом варианте. Кроме того, в этом варианте решения не требуется использовать вспомогательную переменную *p*.

While11. Дано целое число *N* (> 1). Вывести наименьшее из целых чисел *K*, для которых сумма 1 + 2 + … + *K* будет больше или равна *N*, и саму эту сумму.

```
task("While11");
int n, k = 0, s = 0;
pt >> n;
while (s < n)
{
    ++k;
    s += k;
}
pt << k << s;
```

В данной программе переменная *k* играет роль *счетчика* (подобного параметру цикла *for*), однако, поскольку используется цикл *while*, увеличение счетчика (с помощью операции инкремента ++) приходится выполнять на каждой итерации в теле цикла.

☑ Тело цикла *while* можно было бы оформить в виде *единственного* оператора:

```
s += ++k;
```

В данном операторе вначале выполняется увеличение значения переменной *k* на 1 (с помощью операции ++), а затем увеличенное значение *k* добавляется к переменной *s*. Заметим, что кроме *префиксного* варианта операции ++ (использованного в нашей программе) существует *постфиксный* вариант операции инкремента: *k*++, при котором вначале *прежнее* значение *k* используется в выражении, содержащем данную операцию, а затем значение *k* увеличивается на 1. Ясно, что использование постфиксной операции в операторе *s* += *k*++ в нашем случае приведет к ошибочному решению.

Использование операции инкремента в виде отдельного оператора (как в первом варианте решения) является и более наглядным, и более «безопасным», так как он менее подвержен ошибкам. В настоящем пособии при оформлении операции инкремента в виде отдельного оператора всегда используется ее префиксный вариант.

While22. Дано целое число *N* (> 1). Если оно является *простым*, то есть не имеет положительных делителей, кроме 1 и самого себя, то вывести True, иначе вывести False.

```
task("While22");
int n;
pt >> n;
int k = n - 1;
while (n % k != 0)
    k--;
pt << (k == 1);
```

В цикле последовательно перебираются по убыванию все возможные делители числа *n* (от *n* – 1 до 1), пока не будет обнаружен делитель, не дающий остатка. Если число составное, то найденный делитель *k* будет больше 1, если простое, то делитель *k* будет равен 1. Количество итераций можно уменьшить, если заметить, что числа от *n* / 2 + 1 до *n* – 1 не могут быть делителями *n*, и с учетом этого инициализировать переменную *k* значением *n* / 2.

☑ На самом деле достаточно анализировать числа от $\sqrt{n}$ до 1, поскольку легко доказать, что если у числа *n* найдется делитель, больший $\sqrt{n}$, то у него обязательно будет и делитель, меньший этой величины. При этом, однако, возникают две проблемы.

Первая заключается в том, что в стандартной библиотеке C++ функция *sqrt*(*x*) из заголовочного файла *cmath*, возвращающая квадратный корень из *x*, реализована только для *вещественных* типов, и при попытке вызова ее для параметра *n* типа *int* возникает неоднозначность, поскольку неясно, к какому из вещественных типов следует привести целочисленный параметр. Для исправления данной проблемы следует воспользоваться, например, операцией явного приведения типа: *sqrt*(*static_cast*<*double*>(*n*)) (можно также просто умножить *n* на вещественную единицу: *sqrt*(1.0 * *n*)).

Вторая проблема состоит в том, что функция *sqrt* возвращает *вещественное* число, поэтому для присваивания ее значения целочисленной переменной *k* необходимо преобразовать это значение к типу *int* (см. п. 5.2). Подобное преобразование приводит к потере информации (отбрасывается дробная часть), поэтому его желательно выполнить явным образом, чтобы избежать предупреждения компилятора. Поскольку нас интересует *округленное* значение квадратного корня, к нему надо предварительно прибавить 0.5:

```
k = static_cast<int>(sqrt(static_cast<double>(n)) + 0.5);
```

Напомним также, что для возможности использования функций из заголовочного файла *cmath* в начало программы надо добавить директиву #*include* <*cmath*>.

☑ Дополнительного уменьшения числа итераций можно добиться, если не анализировать четные значения *k* (за исключением *k* = 2).

3.3. Проектное задание

Выполните задачи групп For и While, указанные в вашем варианте проектного задания (формулировки задач приводятся в [1]). Если вы не получили вариант проектного задания, то выполните задачи из какого-либо варианта.

ВАРИАНТ 1	**ВАРИАНТ 2**	**ВАРИАНТ 3**
For: 1, 19, 27, 31, 39	For: 10, 21, 24, 32, 38	For: 3, 19, 25, 32, 39
While: 5, 7, 17, 26	While: 6, 13, 20, 27	While: 2, 12, 20, 29

ВАРИАНТ 4 For: 14, 17, 22, 35, 38 While: 2, 12, 21, 25	**ВАРИАНТ 5** For: 7, 18, 27, 29, 37 While: 6, 10, 21, 26	**ВАРИАНТ 6** For: 11, 17, 23, 30, 40 While: 6, 8, 21, 24
ВАРИАНТ 7 For: 4, 21, 22, 29, 40 While: 2, 10, 19, 25	**ВАРИАНТ 8** For: 15, 18, 26, 31, 39 While: 3, 7, 17, 28	**ВАРИАНТ 9** For: 9, 19, 23, 30, 38 While: 5, 9, 18, 24
ВАРИАНТ 10 For: 8, 18, 25, 34, 40 While: 3, 13, 19, 28	**ВАРИАНТ 11** For: 2, 21, 24, 34, 37 While: 5, 9, 21, 29	**ВАРИАНТ 12** For: 6, 17, 26, 35, 37 While: 3, 8, 18, 27
ВАРИАНТ 13 For: 1, 18, 23, 34, 37 While: 6, 12, 19, 26	**ВАРИАНТ 14** For: 15, 17, 26, 29, 40 While: 3, 8, 21, 27	**ВАРИАНТ 15** For: 4, 19, 27, 34, 40 While: 2, 7, 21, 24
ВАРИАНТ 16 For: 11, 17, 25, 32, 39 While: 5, 12, 18, 25	**ВАРИАНТ 17** For: 7, 21, 23, 30, 39 While: 5, 9, 17, 28	**ВАРИАНТ 18** For: 8, 19, 26, 32, 39 While: 3, 10, 18, 28
ВАРИАНТ 19 For: 2, 19, 25, 30, 38 While: 5, 8, 20, 25	**ВАРИАНТ 20** For: 3, 18, 24, 35, 40 While: 2, 10, 19, 29	**ВАРИАНТ 21** For: 6, 18, 22, 31, 38 While: 3, 9, 17, 27
ВАРИАНТ 22 For: 9, 21, 22, 31, 38 While: 6, 13, 20, 29	**ВАРИАНТ 23** For: 10, 17, 24, 29, 37 While: 2, 7, 21, 24	**ВАРИАНТ 24** For: 14, 21, 27, 35, 37 While: 6, 13, 21, 26

3.3.1. Указания к заданиям группы For

For2–3. Для нахождения количества чисел, выведенных в цикле, можно использовать либо явную формулу, либо вспомогательную переменную-счетчик, которая увеличивается на 1 на каждой итерации цикла. В For3 для перебора чисел по убыванию проще всего использовать цикл с шагом, равным –1.

For4–6. В For4 достаточно умножать цену 1 килограмма на текущее значение целочисленного параметра цикла, меняющегося от 1 до 10 с шагом 1. Варианты решения For5 приводятся в п. 3.2.1; задание For6 решается аналогично.

For7–11. Простейшие задания на применение стандартных алгоритмов нахождения суммы и произведения. В For7–8 в качестве очередного обрабатываемого числа (слагаемого или сомножителя) достаточно брать параметр цикла, изменяя его от A до B с шагом 1. В For9–11 очередное слагаемое необходимо *вычислять*, используя текущее значение параметра цикла; при этом начальное и конечное значение параметра цикла удобно выбирать следующим образом: от A до B в For9, от 1 до N в For10, от N до $2N$ в For11 (шаг всюду равен 1).

For12–13. Решения этих заданий приводятся в п. 3.2.1.

For14. Для перебора в цикле всех нечетных чисел можно использовать либо «обычный» цикл по i от 1 до N с шагом 1 (при этом нечетные числа k находятся по формуле $k = 2i - 1$), либо, что в данном случае проще, цикл по k от 1 до $2N - 1$ *с шагом 2*. В этом задании, как и в заданиях For1–6, вывод результатов осуществляется в самом цикле.

For15–18. Решение For16 приводится в п. 3.2.1; если в этом решении указать оператор вывода $pt << p$ не в цикле, а после цикла, то получим решение задания For15. Задания For17–18 решаются аналогично; в них совместно ис-

пользуются алгоритмы накопления суммы и произведения (ср. с решением For20). По поводу чередования знаков в For18 см. решение For13 (в данном случае нет необходимости в использовании особой переменной, отвечающей за знак элемента выражения; достаточно умножать ранее найденное слагаемое на число $-A$).

For19–21. Решение For20 приводится в п. 3.2.1; задания For19 и For21 решаются аналогично.

For22–28. В этих заданиях при вычислении очередного слагаемого A_K *не следует* заново пересчитывать степенную функцию и факториал (ср. с For17 и For21). Воспользуйтесь *рекуррентным соотношением* $A_K = C_K \cdot A_{K-1}$, подобрав подходящим образом множитель C_K. Например, для For22 множитель C_K равен X/K. В случае знакопеременных выражений (For23–26 и For28) множитель C_K будет *отрицательным* (ср. с For18).

For31–35. Задания на вычисление элементов последовательности, определенной *рекуррентным соотношением*. В For31–32 очередной элемент последовательности определяется по *единственному* предыдущему элементу, поэтому в этих заданиях для последовательного нахождения элементов достаточно использовать *одну* переменную. В For33–34 для вычисления очередного элемента необходимо учитывать значения двух предыдущих элементов, поэтому приходится использовать *три* переменные (см. решение For33 в п. 3.2.1; задание For34 решается аналогично). В For35 надо использовать *четыре* переменные.

For36–38. Решение For36 приводится в п. 3.2.1. Задания For37–38 решаются аналогично, с учетом того, что в данном случае число итераций внутреннего цикла зависит от номера итерации внешнего цикла (например, в For37 параметр j внутреннего цикла следует менять от 1 до i, где i — параметр внешнего цикла).

For39–40. Внутренний цикл в этих заданиях используйте для вывода очередного числа (требуемое количество раз), а внешний — для перебора выводимых чисел. В обоих заданиях число итераций внутреннего цикла зависит от номера итерации внешнего цикла (ср. с For37–38).

3.3.2. Указания к заданиям группы While

While1–2. Решение While1 приводится в п. 3.2.2. Задание While2 решается аналогично, с дополнительным подсчетом количества итераций цикла.

While3. Ср. с While1–2.

While4–5. Решение While4 приводится в п. 3.2.2. Для решения While5 достаточно модифицировать первый вариант решения While4, заменив множитель 3 на 2 и добавив подсчет числа итераций цикла.

While6. Каждый следующий сомножитель в формуле двойного факториала меньше предыдущего на 2. Поэтому заведем переменную *i*, до цикла инициализируем ее значением *N*, а в цикле будем накапливать произведение (умножая его на *i*) и уменьшать *i* на 2 (используя оператор *i* –= 2), пока выполняется неравенство $i > 1$.

While7–8. Для решения While7 следует завести счетчик *K* (инициализировав его нулем) и увеличивать его в цикле, пока выполняется неравенство $K^2 \le N$. Для решения While8 достаточно использовать алгоритм решения While7 и уменьшить полученное значение счетчика *K* на 1.

While9–14. Ср. While9–10 с While7–8. Решение While11 приводится в п. 3.2.2; задания While12–14 решаются аналогично.

While17–19. Последовательно выделяйте, обрабатывайте и удаляйте цифры из исходного числа *N*, пока выполняется условие $N > 0$. Для выделения и последующего удаления цифр используйте операции целочисленного деления и взятия остатка — см. решения Integer8 и Integer11 в п. 1.2.3. В While19 для добавления очередной *правой* цифры к формируемому числу

достаточно умножить прежнее значение числа на 10 и прибавить к нему эту цифру.

While20–21. В этих заданиях, в отличие от While17–19, цикл надо продолжать, пока выполняются *два* условия: основное условие $N > 0$ и дополнительное условие, указанное в задании («очередная цифра D не равна 2» для While20 или «очередная цифра D является четной» для While21). Таким образом в условии цикла необходимо использовать логическую операцию &&. После выхода из цикла достаточно проанализировать последнюю найденную цифру D (например, в While20 надо вывести значение логического выражения $D == 2$). По поводу проверки четности в задании While21 см. решение Boolean2 в п. 2.2.1.

While22. Решение этого задания приводится в п. 3.2.2.

While24–27. Последовательно находите числа Фибоначчи F_K, пока выполняется условие $F_K < N$ (или $F_K \le N$ в While25). В While27 дополнительно вычисляйте номер текущего числа Фибоначчи. По поводу нахождения чисел Фибоначчи см. решение For33 в п. 3.2.1.

While28–29. Ср. While28 с For31, While29 с For34.

While30. Ср. с While1–2.

3.4. Тест рубежного контроля

Тест содержит 6 заданий, на выполнение которых отводится 3 минуты. Выберите правильный, по вашему мнению, вариант ответа и отметьте его любым значком в бланке ответов.

1. Укажите неверное утверждение.			
(1)	Тело цикла *for* выполняется хотя бы один раз	(2)	Параметр цикла *for* может иметь вещественный тип

(3)	В заголовке цикла *for* можно задавать любой шаг параметра цикла (как положительный, так и отрицательный)	(4)	Параметр цикла *for* необходимо описывать в заголовке цикла

2. Дан фрагмент программы:

```
int a = 0;
for (int i = 0; i < 5; ++i);
    ++a;
cout << a;
```

Укажите, что произойдет при обработке этого фрагмента.

(1)	Возникнет ошибка компиляции	(2)	Будет напечатано число 0
(3)	Будет напечатано число 1	(4)	Будет напечатано число 5

3. Дан фрагмент программы:

```
int a = 2;
for (int i = a; i <= 2*a; ++i)
    a += 2;
```

Укажите, сколько итераций будет выполнено в цикле *for*.

(1)	1	(2)	2
(3)	3	(4)	более 3

4. Дан фрагмент программы:

```
double a = 2, i = a;
while (i <= 2 * a)
    ++i;
    ++a;
cout << a;
```

Укажите, что произойдет при обработке этого фрагмента.

(1)	Возникнет ошибка компиляции	(2)	Программа зациклится
(3)	Будет напечатано число 3	(4)	Будет напечатано число 5

5. Дан фрагмент программы:

```
for (int i = 0; i <= 4; ++i)
{
    if (i == 2)
        continue;
    for (int j = 0; j < 3; ++j)
    {
        cout << 1;
        if (j == 1)
            break;
    }
}
```

Укажите, сколько раз будет напечатано число 1.

(1)	10 раз	(2)	8 раз
(3)	6 раз	(4)	4 раза

6. Укажите неверное утверждение.

(1)	Цикл *while* с предварительным условием выполняется хотя бы один раз	(2)	Цикл *while* с последующим условием выполняется хотя бы один раз
(3)	Цикл *while* с предварительным условием завершает работу, когда условие становится ложным	(4)	Цикл *while* с последующим условием завершает работу, когда условие становится ложным

4. Модуль № 4. Описание и использование функций. Обработка числовых последовательностей

4.1. Комплексная цель

Изучить правила описания и использования функций в C++, изучить варианты передачи параметров в функции и особенности локальных и глобальных переменных программы. Научиться оформлять ранее изученные алгоритмы в виде функций и использовать их для решения заданий. Освоить базовые алгоритмы обработки числовых последовательностей, в том числе алгоритмы нахождения минимумов и максимумов.

4.2. Содержание

4.2.1. Функции, их описание и использование

При выполнении заданий группы Proc, посвященной разработке процедур и функций, следует учитывать, что в языке C++ нет понятия «процедура»: все подпрограммы считаются *функциями*, даже если они не возвращают значения. Однако в формулировках учебных заданий, в целях краткости и единообразия, понятие «процедура» используется, и в применении к языку C++ оно означает *функцию, не возвращающую значения*.

В формулировках заданий явно указывается имя функции или процедуры, которую требуется реализовать. Следует однако учитывать, что в языке C++ (как и в языке C) имена функций традиционно начинаются с маленькой буквы, а для разделения слов в них используются знаки подчеркивания «_». Поэтому в следующих примерах, давая название функции, будем следовать именно этим традициям, несколько отступая от текста задания.

Приведем общий вид описания функции в языке C++, опустив некоторые необязательные элементы, которые в дальнейшем не будут использоваться:

тип имя([*параметры*])

```
{
    тело функции
}
```

В качестве *типа* можно указывать ключевое слово *void*, означающее, что функция не возвращает значения, или тип возвращаемого значения (кроме того, *тип* может содержать дополнительные спецификаторы, которые здесь не рассматриваются). *Тело функции* состоит из операторов языка C++, в том числе операторов описания. Внутри функции описываются *локальные переменные*, то есть переменные, необходимые только на время ее выполнения. Эти переменные, как правило, каждый раз создаются заново при начале работы функции, а после ее завершения уничтожаются.

Если функция возвращает значение, то среди ее операторов *обязательно* должен присутствовать хотя бы один оператор, определяющий ее возвращаемое значение и имеющий следующий вид:

```
return выражение;
```

Данный оператор полагает возвращаемое значение функции равным значению указанного *выражения* и обеспечивает немедленный выход из функции.

Если функция не возвращает значение, то в ее теле можно использовать оператор *return*, который обеспечивает немедленный выход из функции; если же подобные операторы отсутствуют, то выход из функции происходит при выполнении всех операторов, указанных в теле функции.

Обратимся к *параметрам* функций. Параметры указываются в списке параметров через запятую, вид описания параметра зависит от способа его передачи в функцию и особенностей его использования.

Простейшее описание параметра имеет вид

```
тип имя
```

В этом случае соответствующий параметр передается *по значению*, то есть в функции создается *локальная копия* параметра с указанным значением. Таким

образом передаются в функцию *входные параметры*. Заметим, что в качестве входного параметра можно использовать выражение.

Если параметр является *выходным*, то есть после выполнения функции его значение должно измениться, то его необходимо передавать *по ссылке*, что оформляется при описании параметра следующим образом:

```
тип& имя
```

В этом случае изменение параметра при выполнении функции приводит к изменению соответствующего фактического параметра. В качестве выходного параметра можно использовать только *переменную*. Если параметр является одновременно входным и выходным, то он тоже должен передаваться по ссылке.

Иногда в языке C++ в качестве параметров функций передаются *указатели*, однако в данном пункте мы не будем обсуждать эту возможность.

Для *вызова* функции достаточно указать ее имя и (в скобках) — список фактических параметров, разделенных запятыми. Функции, возвращающие значения, обычно вызываются в выражениях. Если функция не возвращает значение, то ее вызов оформляется в виде отдельного оператора. Функцию, возвращающую значение, также можно вызывать в виде отдельного оператора, но при этом становится недоступным возвращаемое ею значение.

При вызове функции круглые скобки обязательны даже в том случае, когда функция не имеет параметров.

Проиллюстрируем особенности описания и использования функций на примерах решения заданий группы Proc. В примерах программ будем полностью приводить как описания вспомогательных функций, так и текст функции *solve*, включая ее заголовок. Еще раз напомним, что в формулировках заданий под процедурой понимается функция, не возвращающая значение, а также что при реализации требуемых функций мы будем давать им имена, удовлетворяющие традиционным правилам языка C++ (имя функции начинается с маленькой буквы, слова в имени разделяются символами подчеркивания).

Proc4. Описать процедуру TrianglePS(*a*, *P*, *S*), вычисляющую по стороне *a* равностороннего треугольника его периметр $P = 3a$ и площадь $S = a^2\sqrt{3}/4$ (*a* — входной, *P* и *S* — выходные параметры; все параметры являются вещественными). С помощью этой процедуры найти периметры и площади трех равносторонних треугольников с данными сторонами.

```
#include <cmath>
void triangle_ps(double a, double& p, double& s)
{
    p = 3 * a;
    s = a * a * sqrt(3.0) / 4;
}
void solve()
{
    task("Proc4");
    for (int i = 1; i <= 3; ++i)
    {
        double a, p, s;
        pt >> a;
        triangle_ps(a, p, s);
        pt << p << s;
    }
}
```

Заметим, что в использовании специальных переменных для хранения сторон трех исходных треугольников (например, *a1*, *a2*, *a3*) нет необходимости, так как треугольники обрабатываются *последовательно*, и после обработки очередного треугольника связанные с ним переменные (в нашем случае *a*, *p*, *s*) можно использовать при обработке следующего треугольника.

Обратите внимание на то, что в качестве выходных параметров передаются *неинициализированные* переменные *p* и *s*. Это допустимо, так как после выпол-

нения функции эти параметры получат требуемые значения. Кроме того, данные переменные, как и переменная *a*, описаны в теле цикла *for*, поскольку их значения используются только в пределах этого цикла (подобные переменные, описанные внутри блока операторов и, следовательно, имеющие область действия только в пределах этого блока, называются *блочными*).

Напомним, что для возможности использования функции *sqrt* в начале программы необходимо указать директиву

```
#include <cmath>
```

Следующее простое задание демонстрирует использование параметра, одновременно являющегося входным и выходным.

Proc8. Описать процедуру AddRightDigit(*D*, *K*), добавляющую к целому положительному числу *K* справа цифру *D* (*D* — входной параметр целого типа, лежащий в диапазоне 0–9, *K* — параметр целого типа, являющийся одновременно входным и выходным). С помощью этой процедуры последовательно добавить к данному числу *K* справа данные цифры D_1 и D_2, выводя результат каждого добавления.

```
void add_right_digit(int d, int& k)
{
    k = k * 10 + d;
}
void solve()
{
    task("Proc8");
    int k;
    pt >> k;
    for (int i = 1; i <= 2; ++i)
    {
        int d;
        pt >> d;
```

```
            add_right_digit(d, k);
            pt << k;
        }
    }
```

В заключение рассмотрим два задания, которые связаны с функциями, возвращающими значения. Все параметры подобных функций обычно являются входными.

Proc21. Описать функцию SumRange(*A*, *B*) целого типа, находящую сумму всех целых чисел от *A* до *B* включительно (*A* и *B* — целые). Если *A* > *B*, то функция возвращает 0. С помощью этой функции найти суммы чисел от *A* до *B* и от *B* до *C*, если даны числа *A*, *B*, *C*.

```
int sum_range(int a, int b)
{
    int s = 0;
    for (int i = a; i <= b; ++i)
        s += i;
    return s;
}
void solve()
{
    task("Proc21");
    int a, b, c;
    pt >> a >> b >> c;
    pt << sum_range(a, b) << sum_range(b, c);
}
```

Следует заметить, что особая ситуация *a* > *b* в функции *sum_range* не требует специальной обработки, поскольку в этом случае в цикле не выполняется ни одной итерации. Обратите внимание на то, что в разделе операторов функции *solve* вызов функции *sum_range* совмещен с выводом значения, возвращае-

мого этой функцией. Таким образом, раздел операторов содержит только операторы описания переменных, ввода исходных данных и вывода результатов

Proc25. Описать функцию IsSquare(*K*) логического типа, возвращающую True, если целый параметр *K* (> 0) является квадратом некоторого целого числа, и False в противном случае. С ее помощью найти количество квадратов в наборе из 10 целых положительных чисел.

```
bool is_square(int k)
{
    int n = 1;
    while (n * n < k)
        ++n;
    return n * n == k;
}
void solve()
{
    task("Proc25");
    int n = 0;
    for (int i = 1; i <= 10; ++i)
    {
        int k;
        pt >> k;
        if (is_square(k))
            ++n;
    }
    pt << n;
}
```

Благодаря оформлению алгоритма, распознающего квадраты чисел, в виде отдельной функции, соответствующая проверка введенных чисел в разделе операторов основной программы становится простой и наглядной.

Для того чтобы разработанные функции можно было использовать в различных программах, их объединяют в *cpp-файлы* (называемые также *файлами с реализацией*) которые затем подключают к требуемым проектам. Это избавляет программиста от необходимости копировать код нужной функции во все программы, в которых эта функция может потребоваться.

При выполнении учебных заданий возможность сохранения функций в отдельном cpp-файле также оказывается полезной, так как в некоторых заданиях требуется использовать ранее созданные функции.

Рассмотрим задание Proc10.

Proc10. Описать процедуру Swap(*X*, *Y*), меняющую содержимое переменных *X* и *Y* (*X* и *Y* — вещественные параметры, являющиеся одновременно входными и выходными). С ее помощью для данных переменных *A*, *B*, *C*, *D* последовательно поменять содержимое следующих пар: *A* и *B*, *C* и *D*, *B* и *C* и вывести новые значения *A*, *B*, *C*, *D*.

Функция *swap*, которую требуется реализовать в этом задании, пригодится и при выполнении других заданий (например, Proc11, Proc12 и Proc13). Поэтому целесообразно описать ее в отдельном cpp-файле, который можно назвать *MyFunc*.

Процесс создания нового cpp-файла опишем для среды Visual Studio 2005; в других средах cpp-файлы создаются аналогично.

Создавать новый cpp-файл удобно после создания проекта-заготовки для данного задания. Для создания файла выполните команду «Project | Add New Item…» из меню Visual Studio. В появившемся окне в списке категорий шаблонов (Categories), расположенном слева, выберите вариант «Code»; в появившемся справа списке шаблонов данной категории выберите вариант «C++ File (.cpp)». Имя, указанное в нижней части окна, замените на MyFunc; таким образом, создаваемый файл будет иметь имя MyFunc.cpp. После нажатия кнопки

«Add» будет создан пустой файл, который сразу загрузится в редактор. Заметим, что этот файл будет автоматически подключен к нашему проекту, в чем можно убедиться, посмотрев на окно «Solution Explorer», расположенное в правой части окна среды Visual Studio (в нем в списке «Source Files» будет указан файл с именем MyFunc.cpp).

В созданный файл следует ввести описание требуемой функции:

[MyFunc.cpp]

```
void swap(double& a, double& b)
{
    double temp = a;
    a = b;
    b = temp;
}
```

Однако наличие нового cpp-файла в проекте еще не означает, что описанные в нем функции будут доступны в других файлах приложения. Для того чтобы они стали доступны, необходимо *объявить* в файле нужную функцию.

☑ Разница между *объявлением* и *описанием* (или определением) состоит в том, что объявление говорит об имени сущности (например, функции) и ее «внешних» характеристиках (например, о списке параметров), в то время как описание (определение) представляет собой внутреннее устройство сущности (например, тело функции).

Общепринятым способом размещения объявлений функций является расположение их в *заголовочном файле*, имя которого совпадает с именем того cpp-файла, в котором определены данные функции, а в качестве расширения используется .h (от англ. header «заголовок»). После создания такого заголовочного файла (h-файла) он подключается директивой *#include* к тем файлам приложения, в которых требуется использовать данные функции.

Создадим файл MyFunc.h, выполнив те же действия, что и при создании файла MyFunc.cpp, за исключением того, что в качестве шаблона теперь надо выбрать вариант «Header File (.h)». Этот файл сразу будет подключен к нашему

проекту (в окне «Solution Explorer» он будет располагаться в группе «Header Files»).

Поскольку для объявления функции достаточно указать ее заголовок (без следующего за ним тела функции), в файле MyFunc.h должна содержаться единственная строка кода (которую можно скопировать из файла MyFunc.cpp, дополнив точкой с запятой):

[MyFunc.h]

```
void swap(double& a, double& b);
```

Осталось перейти в редакторе Visual Studio на вкладку с файлом Proc10.cpp и дополнить этой файл следующим образом (полужирным шрифтом выделены те строки, которые требуется добавить к существующей заготовке; завершающая часть данного файла не приводится, так как в ее корректировке нет необходимости):

[Proc10.cpp]

```
#include <windows.h>
#pragma hdrstop
#include "pt4.h"
#include "MyFunc.h"

void solve()
{
    task("Proc10");
    double a, b, c, d;
    pt >> a >> b >> c >> d;
    swap(a, b);
    swap(c, d);
    swap(b, c);
    pt << a << b << c << d;
}
.  .  .
```

Если в дальнейшем потребуется подключить файл MyFunc.cpp к другому проекту, то для этого достаточно будет после создания заготовки проекта выполнить команду «Project | Add Existing Item…» и в появившемся окне выбрать два файла: MyFunc.cpp и MyFunc.h (щелкнув на их именах мышью при нажатой клавише [Ctrl]). В том, что указанные файлы будут подключены к проекту, можно убедиться с помощью окна «Solution Explorer». Если потребуется откорректировать данные файлы, то для их загрузки в редактор Visual Studio будет достаточно выполнить двойной щелчок мышью на именах этих файлов в окне «Solution Explorer».

4.2.2. Обработка последовательностей, нахождение минимумов и максимумов

Группа заданий Series содержит задания на обработку последовательностей чисел. В алгоритмах решения этих заданий требуется совместно использовать и условные операторы, и операторы цикла (с параметром или условием).

Поскольку при выполнении заданий из этой группы не требуется применять новые языковые конструкции, сразу перейдем к примерам решения заданий.

Series15. Дано целое число *K* и набор ненулевых целых чисел; признак его завершения — число 0. Вывести номер первого числа в наборе, большего *K*. Если таких чисел нет, то вывести 0.

```
task("Series15");
int k;
pt >> k;
int i = 0, n = 0;
while (true)
{
    int a;
    pt >> a;
```

```
        if (a == 0)
            break;
        ++i;
        if (a > k)
        {
            n = i;
            break;
        }
    }
    pt << n;
```

В этом решении применен так называемый «цикл с выходом из середины», в котором условие продолжения цикла всегда является истинным (равно логической константе *true*). В начале тела цикла вводится очередное число последовательности и, если оно оказывается нулевым, происходит немедленный выход из цикла с помощью оператора *break*. Переменная *i* используется в качестве счетчика прочитанных элементов последовательности (то есть *ненулевых* чисел). При обнаружении первого элемента, большего *k*, в переменной *n* запоминается его номер, после чего также происходит выход из цикла в помощью оператора *break*. Если элемент, больший *k*, в последовательности отсутствует, то переменная *n* сохраняет свое начальное значение, равное 0.

Приведем другой, более традиционный, вариант решения этой же задачи.

```
task("Series15");
int k, a;
pt >> k >> a;
int i = 0, n = 0;
while (a != 0)
{
    ++i;
    if (a > k)
    {
```

```
            n = i;
            break;
        }
        pt >> a;
    }
    pt << n;
```

Следует обратить внимание на то, что теперь первый элемент последовательности считывается до цикла, а остальные — в цикле, причем сразу после считывания очередного элемента он сравнивается с 0, и при обнаружении равенства цикл завершает работу (тем самым нулевое значение в цикле не обрабатывается, что является правильным, поскольку 0 считается не очередным элементом последовательности, а лишь *признаком ее завершения*). В качестве недостатка такого решения можно указать то, что приходится дублировать код для ввода очередного числа последовательности (перед циклом и в теле цикла), а кроме того расширяется область видимости переменной *a*, используемой для хранения этого числа.

Series17. Дано вещественное число *B*, целое число *N* и набор из *N* вещественных чисел, упорядоченных по возрастанию. Вывести элементы набора вместе с числом *B*, сохраняя упорядоченность выводимых чисел.

```
task("Series17");
bool f = true;
double b;
int n;
pt >> b >> n;
for (int i = 1; i <= n; ++i)
{
    double a;
    pt >> a;
    if (a > b && f)
```

```
        {
            pt << b;
            f = false;
        }
        pt << a;
    }
    if (f)
        pt << b;
```

При решении данного задания удобно использовать переменную-*флаг f* логического типа. Если значение флага равно *true*, значит, число *b* еще не было выведено. При обнаружении элемента последовательности, большего числа *b*, выполняется вывод числа *b* и «сброс» флага *f* в состояние *false*. Важно также проверять состояние флага после завершения цикла; это позволит правильно обработать ситуацию, когда число *b* превосходит *все* числа из исходного набора и, следовательно, не было выведено ни на одной из итераций цикла.

Series19. Дано целое число N (> 1) и набор из N целых чисел. Вывести те элементы в наборе, которые меньше своего левого соседа, и количество K таких элементов.

```
task("Series19");
int n, a, k = 0;
pt >> n >> a;
for (int i = 2; i <= n; ++i)
{
    int b;
    pt >> b;
    if (a > b)
    {
        pt << b;
        ++k;
```

```
        }
        a = b;
    }
    pt << k;
```

На каждой итерации цикла сравниваются две переменные: *b* — *текущий* элемент и *a* — *предыдущий* элемент последовательности (то есть *левый сосед* текущего элемента). В конце каждой итерации текущий элемент сохраняется в переменной *a* и в дальнейшем используется при анализе следующего элемента.

Series21. Дано целое число *N* (> 1) и набор из *N* вещественных чисел. Проверить, образует ли данный набор возрастающую последовательность. Если образует, то вывести True, если нет — вывести False.

```
task("Series21");
bool f = true;
int n;
double a;
pt >> n >> a;
for (int i = 2; i <= n; ++i)
{
    double b;
    pt >> b;
    if (a >= b)
    {
        f = false;
        break;
    }
    a = b;
}
pt << f;
```

В данном задании, как и в ранее рассмотренном задании Series19, на каждой итерации анализируются соседние элементы последовательности: *a* и *b*. При обнаружении пары элементов, нарушающих упорядоченность по возрастанию, происходит немедленный выход из цикла.

Группа Minmax содержит различные задания, связанные с минимальными и максимальными элементами числовых последовательностей, поэтому она может считаться естественным продолжением группы Series. Приведем решение первого задания из этой группы (сравните его с решением задания If12 в п. 2.2.2).

Minmax1. Дано целое число *N* и набор из *N* чисел. Найти минимальный и максимальный из элементов данного набора и вывести их в указанном порядке.

```
task("Minmax1");
int n;
double a, min, max;
pt >> n >> a;
min = max = a;
for (int i = 2; i <= n; ++i)
{
    pt >> a;
    if (a < min)
        min = a;
    else if (a > max)
        max = a;
}
pt << min << max;
```

В данном решении мы использовали *каскад* условных операторов (см. решение задания If26 в п. 2.2.2).

4.3. Проектное задание

Выполните задачи, указанные в вашем варианте проектного задания (формулировки задач групп Proc и Series приводятся в [1], а группы Minmax — в [2]). Если вы не получили вариант проектного задания, то выполните задачи из какого-либо варианта.

ВАРИАНТ 1 Proc: 1, 26, 29, 42, 53 Series: 2, 12, 29 Minmax: 2, 14, 18	**ВАРИАНТ 2** Proc: 6, 22, 30, 50, 52 Series: 1, 18, 26 Minmax: 11, 12, 21	**ВАРИАНТ 3** Proc: 15, 28, 31, 44, 52 Series: 7, 22, 39 Minmax: 6, 13, 29
ВАРИАНТ 4 Proc: 7, 24, 34, 40, 55 Series: 6, 16, 30 Minmax: 10, 15, 25	**ВАРИАНТ 5** Proc: 11, 19, 36, 46, 58 Series: 3, 14, 28 Minmax: 7, 15, 27	**ВАРИАНТ 6** Proc: 13, 22, 35, 41, 53 Series: 11, 24, 27 Minmax: 5, 12, 20
ВАРИАНТ 7 Proc: 2, 20, 35, 47, 59 Series: 5, 13, 38 Minmax: 4, 13, 17	**ВАРИАНТ 8** Proc: 14, 17, 34, 51, 57 Series: 11, 20, 36 Minmax: 6, 13, 24	**ВАРИАНТ 9** Proc: 9, 16, 31, 45, 60 Series: 8, 23, 35 Minmax: 3, 15, 28
ВАРИАНТ 10 Proc: 3, 23, 29, 49, 56 Series: 10, 13, 37 Minmax: 8, 12, 16	**ВАРИАНТ 11** Proc: 12, 18, 36, 48, 54 Series: 9, 25, 31 Minmax: 7, 14, 26	**ВАРИАНТ 12** Proc: 5, 27, 30, 43, 56 Series: 4, 14, 40 Minmax: 9, 14, 19

ВАРИАНТ 13 Proc: 6, 27, 34, 49, 52 Series: 11, 16, 37 Minmax: 9, 12, 17	**ВАРИАНТ 14** Proc: 14, 17, 29, 42, 59 Series: 2, 25, 35 Minmax: 4, 13, 29	**ВАРИАНТ 15** Proc: 7, 20, 35, 47, 52 Series: 4, 23, 36 Minmax: 2, 14, 24
ВАРИАНТ 16 Proc: 11, 22, 31, 45, 53 Series: 10, 13, 26 Minmax: 5, 14, 19	**ВАРИАНТ 17** Proc: 9, 18, 30, 41, 54 Series: 7, 14, 39 Minmax: 11, 14, 16	**ВАРИАНТ 18** Proc: 2, 22, 36, 44, 53 Series: 3, 12, 28 Minmax: 6, 15, 18
ВАРИАНТ 19 Proc: 12, 16, 35, 40, 55 Series: 11, 24, 27 Minmax: 7, 12, 27	**ВАРИАНТ 20** Proc: 1, 24, 29, 48, 58 Series: 5, 14, 29 Minmax: 6, 12, 21	**ВАРИАНТ 21** Proc: 5, 19, 34, 43, 56 Series: 1, 13, 38 Minmax: 7, 13, 25
ВАРИАНТ 22 Proc: 13, 23, 36, 46, 57 Series: 6, 18, 40 Minmax: 8, 15, 26	**ВАРИАНТ 23** Proc: 3, 26, 30, 50, 56 Series: 8, 20, 30 Minmax: 3, 13, 20	**ВАРИАНТ 24** Proc: 15, 28, 31, 51, 60 Series: 9, 22, 31 Minmax: 10, 15, 28

4.3.1. Указания к заданиям группы Proc

Proc1–2. Простейшие задания на разработку функций, не возвращающих значения (подобные функции во многих языках программирования называются *процедурами*; это же название используется в формулировках учебных заданий). Исходные числа удобно обрабатывать *в цикле*, последовательно считывая их в одну и ту же переменную (как в решении Proc4). Аналогич-

ный прием можно использовать и в других заданиях данной группы (например, Proc4–7, Proc12–15 и т. д.).

Proc3. Ср. с Begin8–9.

Proc4. Решение этого задания приводится в п. 4.2.1.

Proc5. Ср. с Begin19.

Proc6. Ср. с While18.

Proc7. Ср. с While19.

Proc8–9. В Proc8 достаточно умножить исходное число *K* на 10 и прибавить к нему цифру *D*; решение этого задания приводится в п. 4.2.1. В Proc9 надо к числу *K* прибавить цифру *D*, умноженную на 10 в степени, равной *количеству цифр* числа *K* (требуемую степень числа 10 вычисляйте в цикле; для определения количества цифр числа *K* можно воспользоваться функцией *DigitCountSum* из задания Proc6).

Proc10. Решение этого задания приводится в п. 4.2.1; ср. его с Begin22. Полезно сохранить созданную функцию *Swap* в отдельном файле, чтобы в дальнейшем использовать при решении других заданий (например, Proc11–13).

Proc11. Ср. с If9. Для перемены местами содержимого *X* и *Y* можно использовать функцию *Swap* из задания Proc10.

Proc12–13. Для сортировки трех значений *A*, *B*, *C* *по возрастанию* достаточно выполнить три последовательных вызова функции *Minmax* из задания Proc11 с параметрами (*A*, *B*), (*B*, *C*) и (*A*, *B*). Для сортировки этих значений *по убыванию* достаточно отсортировать их по возрастанию и поменять местами содержимое *A* и *C* (для этого можно использовать функцию *Swap* из задания Proc10).

Proc14–15. Ср. с Begin23–24.

Proc16–17. Простейшие задания на разработку функций, возвращающих значения. По поводу ввода исходных данных см. указание к Proc1–2. В этих и

последующих заданиях вызов требуемых функций можно *совместить с выводом результатов*; например, в Proc16 требуемое выражение *Sign*(*a*) + *Sign*(*b*) можно поместить непосредственно в оператор вывода.

Proc18–19. Ср. с Begin13.

Proc21. Решение этого задания приводится в п. 4.2.1.

Proc22. Ср. с Case5. В данном случае удобно использовать оператор выбора с *вариантом по умолчанию default*.

Proc23. Ср. с If22.

Proc24–28. Задания на разработку функций, возвращающих логические значения. Ср. Proc24 с Boolean3, Proc25–27 с While4–5, Proc28 с While22.

Proc29–30. Ср. с While17–18.

Proc32–33. Ср. с Begin31–32.

Proc34. Ср. с For19.

Proc35. Ср. с While6.

Proc36. Ср. с For33.

Proc37–39. При описании функции *Power1* в Proc37 используйте стандартные математические функции *exp* и *log* (см. п. 5.3); ср. Proc38 с For15. Созданные в Proc37–38 функции *Power1* и *Power2* удобно сохранить в отдельном файле (см. решение Proc10), чтобы упростить их использование при решении Proc39.

Proc40–45. См. указание к For22–28. Поскольку количество слагаемых в сумме заранее не известно, используйте цикл с условием.

Proc46–49. Ср. Proc46 с While23. Для хранения созданных процедур и функций используйте отдельный файл.

Proc50–55. Ср. Proc50 с Integer19–23, Proc52 с If28, Proc53 с Case4, Proc54–55 с Case8–9. В Proc51 переведите исходное время в количество секунд, прибавьте к нему число *T* и выполните обратное преобразование времени в ча-

сы, минуты и секунды (для обратного преобразования удобно воспользоваться процедурой *TimeToHMS* из задания Proc50). Для хранения созданных процедур и функций используйте отдельный файл.

Proc56–60. Ср. Proc56–58 с Begin20–21. Для хранения созданных процедур и функций используйте отдельный файл.

4.3.2. Указания к заданиям группы Series

Series1–7. Простейшие задачи на обработку последовательностей, использующие стандартные алгоритмы нахождения суммы и произведения (ср. с For7–11). В Series5 для выделения *целой части* вещественного числа используйте стандартную функцию *floor* (см. п. 5.3). В Series6 для выделения *дробной части* числа достаточно вычесть из данного числа его целую часть. Для *округления* числа в Series7 достаточно прибавить к нему 0.5, после чего преобразовать полученное выражение к типу *int* (см. п. 5.2).

Series8–9. Задания на обработку элементов, удовлетворяющих некоторому условию. По поводу проверки на четность/нечетность см. решение Boolean2 в п. 2.2.1. В Series9 для определения номера текущего элемента набора воспользуйтесь параметром цикла.

Series10–11. Задания на проверку наличия в наборе требуемых элементов. При обнаружении требуемого элемента цикл, в котором обрабатываются элементы, следует прервать, используя оператор *break*.

Series12–16. Особенность этих заданий состоит в том, что перед началом ввода элементов набора *не известно их количество*: признаком завершения набора является число 0, расположенное после последнего элемента набора (подчеркнем, что число 0 в набор *не входит*). Для решения этих заданий следует использовать цикл с условием. Решение Series15 приводится в п. 4.2.2. Для получения решения Series16 достаточно в решении Series15 убрать из цикла *while* оператор *break*.

Series17. Решение этого задания приводится в п. 4.2.2.

Series18–22. Для выполнения этих заданий необходимо на каждом этапе обработки исходного набора иметь доступ не только к *текущему* (только что введенному) элементу, но и к *предыдущему* элементу, поскольку их необходимо сравнивать между собой. Решения Series19 и Series21 приводятся в п. 4.2.2, прочие задания решаются аналогично.

Series23. Ср. с Series18–22. Удобно на каждом этапе обработки исходного набора иметь доступ к *трем* последовательным элементам *a*, *b*, *c*; в этом случае легко проверить, является ли *средний* элемент зубцом (достаточно сравнить *знаки* выражений $b - a$ и $b - c$ — ср. со вторым вариантом решения Boolean7 в п. 2.2.1). После обработки текущей тройки элементов значение *b* необходимо переместить в переменную *a*, значение *c* — в переменную *b*, а очередной элемент набора ввести в переменную *c*.

Series24–25. Сложность этих заданий состоит в том, что при обработке элементов, расположенных в наборе после первого нуля, заранее не известно, надо ли включать их в требуемую сумму (поскольку мы еще не знаем, сколько нулей будет обнаружено в наборе). Опишем алгоритм решения Series24. Используем две переменные *s* и *t*, инициализируя их нулевыми значениями. Считываем начальные элементы исходного набора, пока не обнаружим первый нуль. После этого в цикле перебираем остальные элементы набора, причем если очередной элемент отличен от нуля, то он добавляется к сумме *t*, а если очередной элемент равен нулю, то текущее значение *t* записывается в *s*, а переменная *t* обнуляется. В результате после перебора всех элементов исходного набора в переменной *s* будет содержаться требуемая сумма. Алгоритм для Series25 имеет единственное отличие: при обнаружении очередного нуля текущее значение *t* должно *добавляться* к прежнему содержимому *s*. Для считывания начальных элементов набора (до первого нуля) можно использовать вспомогательный цикл с условием.

Можно также использовать единственный цикл для считывания всех элементов набора, но в этом случае необходимо по-разному обрабатывать элементы, прочитанные до и после обнаружения первого нуля.

Series26–28. Ср. с For36–38.

Series29–34. Задания на обработку нескольких числовых наборов одинакового размера. Используйте внешний цикл (с параметром, меняющимся от 1 до *K*) для перебора различных исходных наборов. Для обработки элементов очередного набора потребуется вложенный цикл, параметр которого будет меняться от 1 до *N*. В этих заданиях важно правильно определить место инициализации используемых переменных. Например, в Series29 переменная для хранения суммы элементов всех наборов должна инициализироваться нулем *один раз* перед началом выполнения внешнего цикла, а в Series30 переменная для хранения суммы элементов текущего набора должна принимать нулевое значение в начале *каждой* итерации внешнего цикла (перед выполнением вложенного цикла).

Series35–40. Задания на обработку нескольких числовых наборов *различного* размера. Перед началом ввода элементов очередного набора *не известно их количество*, поэтому в данном случае, в отличие от Series29–34, в качестве вложенного цикла следует использовать *цикл с условием* (см. указание к Series12–16). По поводу инициализации вспомогательных переменных см. указание к Series29–34. Ср. Series36 с Series21, Series39–40 с Series23. Алгоритм, описанный в указании к Series23, можно использовать и в Series37–38, если заметить, что ни один из внутренних элементов возрастающего или убывающего набора *не может быть зубцом* (элемент называется *внутренним*, если он имеет *двух* соседей: слева и справа). Таким образом, для решения Series37 достаточно подсчитать количество наборов, не имеющих внутренних зубцов. При решении Series38 для упорядоченных наборов требуется дополнительно выяснить «способ упорядоченно-

сти»: по возрастанию или убыванию. Для этого достаточно сравнить *два последних* элемента набора (эти элементы сохранятся во вспомогательных переменных после выхода из вложенного цикла).

4.3.3. Указания к заданиям группы Minmax

Minmax1–3. Решение Minmax1 приводится в п. 4.2.2. В Minmax2–3 элементы набора, в котором ищется минимум/максимум, необходимо *вычислять* по имеющимся исходным данным.

Minmax4–11. В программе необходимо использовать не только переменные *min/max* для хранения самих минимальных/максимальных элементов (см. решение Minmax1), но и дополнительные переменные для хранения их *номеров* (например, *nMin* и *nMax*). В случае, когда имеется несколько минимальных/максимальных элементов и требуется найти номер первого или последнего из них, важно выбрать правильную операцию отношения (<, > или <=, >=) при сравнении исходных данных с переменной *min/max*.

Minmax12–15. Задания на нахождение *условного* минимума/максимума. В данном случае переменные *min/max* (предназначенные для хранения результата) нельзя инициализировать значением первого элемента исходного набора (как в решении Minmax1, приведенном в п. 4.2.2), так как этот элемент может не удовлетворять дополнительным условиям. Проще всего инициализировать переменную *min* значением, которое заведомо *больше* всех элементов исходного набора данных, а переменную *max* — значением, которое заведомо *меньше* этих элементов (в качестве таких значений удобно использовать определения из заголовочного файла *limit*s, возвращающие минимальные или максимальные значения требуемых числовых типов, например, *numeric_limits<int>::min()* или *numeric_limits<double>::max()* — см. п. 5.1). Еще одна особенность условных минимумов/максимумов заключается в том, что для некоторых наборов данных они могут отсутство-

вать. Выявить такую ситуацию несложно: в этом случае *начальное значение* переменной *min/max не изменится* после просмотра исходного набора данных.

Minmax16–18. Решение сводится к нахождению *номера* требуемого минимального/максимального элемента (см. также указание к Minmax4–11).

Minmax19–20. Используйте переменную-счетчик для хранения количества найденных минимумов/максимумов. Этот счетчик необходимо корректировать в двух ситуациях: при изменении значения переменной *min/max* (содержащей значение наименьшего/наибольшего элемента *из уже просмотренных* — см. решение Minmax1) и при появлении элемента, значение которого *совпадает* с текущим значением *min/max*.

Minmax21. Поскольку заранее неизвестно, какой элемент набора окажется минимальным (максимальным), следует просуммировать *все* исходные элементы, одновременно определяя минимальный и максимальный из них. После обработки всех исходных данных достаточно вычесть из полученной суммы минимальный и максимальный элементы и разделить результат на $N - 2$.

Minmax22–23. Для решения Minmax22 каждый элемент исходного набора надо сравнивать не только с переменной *min1*, в которой хранится значение наименьшего *из уже просмотренных* элементов, но и с переменной *min2*, содержащей значение, следующее за наименьшим (из уже просмотренных). Более точно: следует проверять, в каком из промежутков $(-\infty, min1)$, $(min1, min2)$, $(min2, +\infty)$ содержится очередной элемент исходных данных, и в зависимости от результата корректировать значения *min1* и *min2*. Важно также позаботиться о правильной *инициализации* этих переменных (например, можно инициализировать обе переменные одним и тем же значением, которое заведомо *больше* всех элементов исходного набора). В Min-

max23 следует использовать три вспомогательные переменные: *max1*, *max2*, *max3*.

Minmax24–28. В этих заданиях основная трудность состоит не в реализации собственно алгоритма нахождения минимума/максимума, а в нахождении величин, из которых надо выбрать минимальное/максимальное значение. Если при решении задания не использовать массив, то текущий (анализируемый) элемент исходных данных необходимо сохранять во вспомогательной переменной, которую в дальнейшем использовать при анализе следующего элемента.

Minmax29–30. В этих заданиях алгоритм нахождения минимума/максимума используется дважды: для отбора минимальных/максимальных элементов из исходного набора данных и для определения самой длинной/короткой последовательности таких элементов. Напомним, что требуемый результат надо получить после *однократного* просмотра исходного набора данных.

4.4. Тест рубежного контроля

Тест содержит 6 заданий, на выполнение которых отводится 3 минуты. Выберите правильный, по вашему мнению, вариант ответа и отметьте его любым значком в бланке ответов.

1. Укажите неверное утверждение.			
(1)	Входные параметры можно передавать по значению	(2)	Входные параметры можно передавать по ссылке
(3)	Выходные параметры можно передавать по значению	(4)	Выходные параметры можно передавать по ссылке

2. Учащийся разработал функцию *swap*:

```
void swap(int& a, int b)
{
    int t = a;
    a = b;
    b = t;
}
```

Для тестирования функции *swap* используется следующий фрагмент:

```
int m = 0, n = 1, k = 2;
swap(m, n);
swap(n, k);
cout << m << ' ' << n << ' ' << k << endl;
```

Укажите, что произойдет при обработке данного фрагмента.

(1)	Возникнет ошибка компиляции	(2)	Будут напечатаны числа 0, 1, 2
(3)	Будут напечатаны числа 1, 2, 2	(4)	Будут напечатаны числа 1, 2, 0

3. Укажите неверное утверждение.

(1)	Объявление функции всегда содержит тип ее возвращаемого значения	(2)	Объявление функции всегда содержит ее тело
(3)	Объявление функции обычно указывается в заголовочном файле	(4)	Программа может содержать несколько объявлений одной и той же функции

4. Дан фрагмент программы, обрабатывающий последовательность из *n* вещественных чисел:

```
double b = 0;
int k = 0;
for (int i = 0; i < n; ++i)
{
    double a;
    cin >> a;
    if (a > 0)
        ++k;
        b += a;
}
cout << b;
```

Укажите, какая характеристика исходной последовательности будет выведена на экран.

(1)	Сумма всех элементов	(2)	Сумма всех положительных элементов
(3)	Среднее арифметическое всех элементов	(4)	Среднее арифметическое всех положительных элементов

5. Дан фрагмент программы, обрабатывающий последовательность из *n* вещественных чисел:

```
double a;
cin >> a;
int k = 0;
for (int i = 2; i < n; ++i)
{
    double b;
    cin >> b;
    if (a > b)
        ++k;
    a = b;
}
cin >> a;
cout << k;
```

Укажите, какая характеристика исходной последовательности будет выведена на экран (*внутренними* элементами последовательности считаются все элементы, кроме начального и конечного).

(1)	Количество внутренних элементов, которые меньше своего левого соседа	(2)	Количество внутренних элементов, которые больше своего левого соседа
(3)	Количество внутренних элементов, которые меньше своего правого соседа	(4)	Количество внутренних элементов, которые больше своего правого соседа

6. Дан фрагмент программы, обрабатывающий последовательность из *n* целых чисел:

```
int a;
cin >> a;
int k = 0;
for (int i = 1; i < n; ++i)
{
    int b;
    cin >> b;
    if (a >= b)
    {
        k = i;
        a = b;
    }
}
cout << k;
```

Укажите, какая характеристика исходной последовательности будет выведена на экран.

(1)	Количество элементов, находящихся перед первым минимальным элементом	(2)	Количество элементов, находящихся после первого минимального элемента
(3)	Количество элементов, находящихся перед последним минимальным элементом	(4)	Количество элементов, находящихся после последнего минимального элементом

5. Приложение А.
Язык C++: справочные сведения

5.1. Числовые типы

В пособии используются наиболее распространенные числовые типы языка C++, а именно, целочисленный тип *int* и вещественный тип *double*.

В C++ имеется много различных *целочисленных типов*, отличающихся размером и отсутствием или наличием знака. В таблице перечислены все стандартные целочисленные типы и их размеры для рассматриваемых реализаций C++ (включение в таблицу типов *char* и *unsigned char*, обычно используемых для хранения символов, объясняется тем, что в арифметических выражениях данные этих типов автоматически приводятся к числам из указанного в таблице диапазона).

Наиболее краткое имя	**Синонимы**	**Размер (в байтах)**	**Диапазон значений**
char	*signed char*	1	от –128 до 127
unsigned char	нет	1	от 0 до 256
short	*short int*, *signed short int*	2	от –32 768 до 32 767
unsigned short	*unsigned short int*	2	от 0 до 65 535
int	*signed*, *signed int*	4	от –2 147 483 648 до 2 147 483 647
unsigned	*unsigned int*	4	от 0 до 4 294 967 295

long	*long int*, *signed long*, *signed long int*	4	от –2 147 483 648 до 2 147 483 647
unsigned long	*unsigned long int*	4	от 0 до 4 294 967 295

В любой из рассматриваемых реализаций языка C++ (то есть Borland C++Builder и Microsoft Visual C++) переменная типа *double* имеет размер 8 байтов, может содержать до 15 десятичных знаков и принимать положительные и отрицательные значения в диапазоне от 10^{-308} до 10^{308}. Имеется также вещественный тип *float*, размер которого в любой реализации C++ не превосходит размера *double*, и тип *long double*, размер которого в любой реализации равен или превосходит размер *double*. В рассматриваемых реализациях языка C++ тип *float* имеет размер 4 байта, может содержать до 7 десятичных знаков и принимать положительные и отрицательные значения в диапазоне от 10^{-38} до 10^{38}, а тип *long double* в реализации Visual C++ совпадает с типом *double*, а в реализации Borland C++Builder имеет размер 10 байтов, может содержать до 18 десятичных знаков и принимать положительные и отрицательные значения в диапазоне от 10^{-4932} до 10^{4932}.

Имеются три заголовочных файла библиотеки C++, позволяющих определить диапазон значений и другие свойства числовых типов: это *climits* и *cfloat* (перенесенные из библиотеки C) и *limits* (библиотека C++, основанная на шаблонах). При подключении к программе заголовочного файла *limits* к ней автоматически подключаются и два других файла. В файле *climits* содержатся характеристики целочисленных типов, в файле *cfloat* — вещественных. Приведем некоторые макросы, реализованные в файлах *climits* и *cfloat*. Следует заметить, что имена макросов для других типов отличаются лишь своими префиксами. Так, для вещественных типов *float* и *long double* вместо *DBL* следует указывать *FLT* и *LDBL* соответственно.

`int INT_MIN` `<climits>`

`int INT_MAX`	`<climits>`

Минимальное и максимальное значение для типа *int* (значения для рассматриваемых реализаций C++ см. в приведенной выше таблице).

`double DBL_MIN`	`<cfloat>`
`double DBL_MAX`	`<cfloat>`

Минимальное положительное и максимальное значение для типа *double*. В рассматриваемых реализациях C++ значения примерно равны 2.22507e–308 и 1.79769e+308 соответственно.

`double DBL_DIG`	`<cfloat>`

Количество десятичных знаков, которые можно хранить в *double*. В рассматриваемых реализациях C++ значение равно 15.

`double DBL_EPSILON`	`<cfloat>`

Разность между 1 и наименьшим из чисел, превышающих 1, которые можно хранить в *double*. Иными словами, это предел точности, достижимой при вычислениях, основанных на типе *double*. В рассматриваемых реализациях C++ значение примерно равно 2.22045e–16.

С той же целью можно использовать определения из заголовочного файла *limits*, соответствующие приведенным выше макросам.

`numeric_limits<int>::min()`	`<limits>`
`numeric_limits<int>::max()`	`<limits>`
`numeric_limits<double>::min()`	`<limits>`
`numeric_limits<double>::max()`	`<limits>`
`numeric_limits<double>::digits()`	`<limits>`
`numeric_limits<double>::epsilon()`	`<limits>`

5.2. Преобразование числовых типов

Преобразование числовых типов может выполняться *неявно* (то есть автоматически) или *явно*. Для явного преобразования типов в языке C++ преду-

смотрен набор операций, самой «старой» из которых (пришедшей из языка C) является операция вида (*тип*)*выражение* (например, (*int*)*a*) В C++ добавлен вариант операции, отличающийся только синтаксисом, совпадающим с синтаксисом вызова функции: *тип*(*выражение*) (например, *int*(*a*); этот вариант возможен только для типов, описываемых *одним словом*). В настоящее время эти операции считаются устаревшими, и применять их не рекомендуется. Вместо этого определен набор шаблонных cast-функций (операций), каждая из которых должна использоваться в конкретных ситуациях. В частности, для преобразования числовых типов должна использоваться операция *static_cast*<*тип*>(*выражение*). Считается, что операции преобразования типа усложняют понимание программы, поэтому во всех случаях, где это возможно, их следует избегать. Достаточно громоздкий синтаксис операций преобразования типов способствует быстрому обнаружению в коде соответствующих конструкций. Заметим, что при явном преобразовании вещественных типов к целочисленным происходит *отбрасывание дробной части*.

Если требуемое преобразование не может привести к потере информации, то оно может быть выполнено неявно (например, тип *short* может быть неявно преобразован в *int*, тип *int* — в *long*, любой целочисленный тип — в *double*). Неявные преобразования, приводящие к потере информации, в языке C++ также разрешены, однако эта возможность является опасной, так как не позволяет вовремя выявить ошибки, связанные с преобразованием типов. По этой причине при обнаружении операторов, содержащих неявные преобразования, которые могут привести к потере информации, компилятор выводит *предупреждение* (warning), например: «conversion from … to …, possible loss of data» («преобразование из … в …, возможна потеря данных). Это предупреждение не препятствует дальнейшей компиляции программы и ее запуску на выполнение. Если известно, что требуемое преобразование не приведет к ошибке, то для по-

давления предупреждающего сообщения достаточно выполнить это преобразование явным образом.

5.3. Стандартные математические функции

Математические функции объявляются в нескольких стандартных заголовочных файлах, среди которых можно отметить *cstdlib* и *cmath* (перенесенные из библиотеки C), *complex* и *valarray* (входящие в библиотеку C++, основанную на шаблонах). Особенностью файла *complex* является возможность работы с комплексными числами, а реализации математических функций в файле *valarray* позволяют применить эти функции ко всем элементам некоторого массива. В настоящем пункте описаны наиболее распространенные функции, реализованные в заголовочных файлах *cstdlib* и *cmath* (приводятся только те перегруженные варианты, которые используют в качестве целочисленного типа тип *int*, а в качестве вещественного типа тип *double*).

Следует заметить, что никакие ошибки, возникающие при выполнении функций из заголовочного файла *cmath* (в частности, ошибки, связанные с неверным параметром или переполнением), не приводят к аварийному завершению программы. Функции в подобной ситуации возвращают особые значения (в частности, при переполнении возвращается значение *HUGE_VAL* типа *double*, определенное в том же заголовочном файле *cmath*). Для проверки успешности выполнения функции можно проанализировать значение системной переменной *errno* (из заголовочного файла *cerrno*), содержащей код последней ошибки. Если ошибок не было, то значение *errno* равно 0. Ненулевое значение *errno* можно сбрасывать в 0 с помощью обычного присваивания (*errno* = 0), при этом нулевое значение будет сохраняться в *errno* до возникновения новой ошибки.

```
int abs(int x)                                          <cstdlib>
double abs(double x)                                      <cmath>
```

Возвращает абсолютное значение (*модуль*) параметра *x*.

```
double ceil(double x)                                     <cmath>
double floor(double x)                                    <cmath>
```

Функции, обеспечивающие *округление* числа соответственно *с избытком* и *с недостатком*. Функция *ceil* возвращает ближайшее к *x* целое число, большее или равное *x*, а *floor* — меньшее или равное *x*). Подчеркнем, что возвращаемое значение имеет тип *double*.

```
double fmod(double y, double x)                           <cmath>
```

Возвращает *дробный остаток* от деления *x* на *y*, то есть величину, равную $x - k*y$, где k — некоторое целое число (параметр *y* не должен равняться нулю). Результат имеет тот же знак, что и *x*; по модулю он всегда меньше *y*.

```
double sqrt(double x)                                     <cmath>
```

Возвращает квадратный корень из неотрицательного параметра *x*: $\sqrt{x}$.

```
double pow(double x, double y)                            <cmath>
double pow(double x, int y)                               <cmath>
```

Две реализации степенной функции x^y. Если параметр *y* (типа *double*) имеет ненулевую дробную часть, то параметр *x* должен быть положительным.

```
double exp(double x)                                      <cmath>
```

Возвращает значение показательной функции e^x от параметра *x*.

```
double log(double x)                                      <cmath>
double log10(double x)                                    <cmath>
```

Логарифмические функции по основанию *e* и 10 от положительного параметра *x*: $\ln x$ и $\lg x$.

```
double sin(double x)                                    <cmath>
double cos(double x)                                    <cmath>
double tan(double x)                                    <cmath>
```

Тригонометрические функции (синус, косинус и тангенс соответственно) от параметра *x*, измеряемого в радианах.

```
double asin(double x)                                   <cmath>
double acos(double x)                                   <cmath>
double atan(double x)                                   <cmath>
```

Обратные тригонометрические функции. Функция *acos* возвращает арккосинус, функция *asin* — арксинус, функция *atan* — арктангенс параметра *x*. Возвращаемые значения измеряются в радианах и лежат в следующих промежутках: $[-\pi/2, \pi/2]$ для *asin* и *atan*, $[0, \pi]$ для *acos*. Для *asin* и *acos* параметр должен находиться в диапазоне [–1, 1].

```
double atan2(double y, double x)                        <cmath>
```

Возвращает *угол наклона* ненулевого радиус-вектора с координатами $\{x, y\}$ к положительной полуоси *OX*, измеряемый в радианах и лежащий в промежутке $(-\pi, \pi]$.

```
double sinh(double x)                                   <cmath>
double cosh(double x)                                   <cmath>
double tanh(double x)                                   <cmath>
```

Гиперболические функции (гиперболический синус, косинус и тангенс соответственно) от параметра *x*.

☑ В заголовочном файле *cmath* отсутствуют константы для чисел *e* и π. Для их получения достаточно воспользоваться стандартными функциями: вызов *exp*(1.0) возвращает число *e*, вызов *acos*(–1.0) возвращает число π.

5.4. Генерация случайных чисел

`void srand(unsigned seed)` `<cstdlib>`

Инициализирует *датчик равномерно распределенных псевдослучайных чисел* с помощью параметра *seed*. Функция должна вызываться в начале программы; если этого не сделать, то при каждом запуске программы функция *rand* будет генерировать одну и ту же последовательность случайных чисел (соответствующую параметру *seed*, равному 1).

Для того чтобы датчик при каждом запуске генерировал различные последовательности случайных чисел, в качестве параметра функции *srand* следует указывать текущее время по *системным часам компьютера* (соответствующий пример приводится в п. 1.2.2).

`int RAND_MAX` `<cstdlib>`

Возвращает максимальное целочисленное значение, которое может вернуть функция *rand*. В рассматриваемых реализациях C++ имеет значение 32 767.

`int rand()` `<cstdlib>`

Возвращает очередное случайное число, лежащее в диапазоне от 0 до *RAND_MAX* включительно.

6. Приложение В.
Средства задачника Programming Taskbook

Описанные в данном разделе функции и объекты будут доступны в программе, если к ней подключен файл pt4.cpp (данный файл автоматически подключается к любому проекту-заготовке, созданному для выполнения задания).

```
void task(char* name);
```

Функция инициализирует задание с именем *name*. Она должна вызываться в начале программы, выполняющей это задание (до вызова функций ввода-вывода *get–put*). Если в программе, выполняющей задание, не указана функция *task*, то при запуске программы будет выведено окно с сообщением «*Не вызвана процедура Task с именем задания*».

Если функция *task* вызывается в программе несколько раз, то все последующие ее вызовы игнорируются.

Имя задания *name* должно включать имя группы заданий и порядковый номер в пределах группы (например, *"Begin3"*). Регистр букв в имени группы может быть произвольным. Если указана неверная группа, то программа выведет сообщение об ошибке, в котором будут перечислены названия всех имеющихся групп. Если указан недопустимый номер задания, то программа выведет сообщение, в котором будет указан диапазон допустимых номеров для данной группы. Если после имени задания в параметре *name* указан символ «?» (например, *"Begin3?"*), то программа будет работать в *демонстрационном режиме*, имеющем следующие особенности:

- даже если программа содержит решение задания, это решение не анализируется и информация в файл результатов не заносится;

- после отображения на экране окна задачника в разделе результатов сразу становится активной вкладка «Пример верного решения»;
- при одном запуске программы можно просмотреть несколько вариантов исходных и контрольных данных; для смены набора данных требуется нажать кнопку «Новые данные» или клавишу пробела;
- при одном запуске программы можно последовательно просмотреть все задания данной группы; для перехода к заданию с большим номером требуется нажать кнопку «Следующее задание» или клавишу [Enter], а для перехода к заданию с меньшим номером требуется нажать кнопку «Предыдущее задание» или клавишу [Backspace]. Задания перебираются циклически.

```
void getB(bool& a);
void getN(int& a);
void getD(double& a);
void getC(char& a);
void getS(char* a);
void getS(string& a);
```

Функции обеспечивают ввод исходных данных в программу, выполняющую учебное задание. Они должны вызываться после вызова функции *task*; в случае их вызова до вызова функции *task* при запуске программы будет выведено сообщение об ошибке «*В начале программы не вызвана процедура Task с именем задания*».

Используемая функция ввода должна соответствовать типу очередного элемента исходных данных; в противном случае выводится сообщение об ошибке «*Неверно указан тип при вводе исходных данных*» (такое сообщение будет выведено, например, если очередной элемент данных является символом, а для его ввода используется функция *getN*).

При попытке ввести больше исходных данных, чем это предусмотрено в задании, выводится сообщение об ошибке «*Попытка ввести лишние исходные данные*». Если исходные данные, необходимые для решения задания, введены не полностью, то выводится сообщение «*Введены не все требуемые исходные данные*».

Следует обратить внимание на то, что строковые данные могут считываться либо в переменную типа *char**, либо в объект *string*.

```
void putB(bool a);
void putN(int a);
void putD(double a);
void putC(char a);
void putS(char* a);
void putS(string a);
```

Функции обеспечивают вывод на экран результирующих данных, найденных программой, и их сравнение с *контрольными данными* (то есть с правильным решением). Как и функции группы *get*, эти функции должны вызываться после вызова функции *task*; в противном случае при запуске программы будет выведено сообщение об ошибке «*В начале программы не вызвана процедура Task с именем задания*».

В отличие от функций группы *get*, в качестве параметра функций группы *put* можно указывать не только переменные, но и *выражения* (в частности, *константы* соответствующего типа). Используемая функция должна соответствовать типу очередного элемента результирующих данных, в противном случае выводится сообщение об ошибке «*Неверно указан тип при выводе результатов*». Как и в случае функций группы *get*, при вызовах функций группы *put* программа осуществляет контроль за соответствием количества требуемых и выведенных результирующих данных. Если программа выведет недостаточное или избыточное количество результирующих данных, то после проверки этих

данных появится сообщение «*Выведены не все результирующие данные*» или, соответственно, «*Попытка вывести лишние результирующие данные*».

Следует обратить внимание на то, что в функции *putS* можно указывать параметры двух типов: *char** и *string*.

`pt (поток ввода-вывода)`

Поток ввода-вывода *pt* может применяться вместо функций групп *get–put* (именно он используется во всех примерах решений, приведенных в настоящем пособии). С его использованием операторы ввода-вывода могут быть оформлены более компактно. Например, вместо последовательности вызовов функций *getN*(*a*); *getD*(*b*); *getS*(*s*); достаточно указать один оператор чтения из потока:

```
pt >> a >> b >> s;
```

Литература

1. *Абрамян М. Э.* 1000 задач по программированию. Часть I: Скалярные типы данных, управляющие операторы, процедуры и функции. — Ростов н/Д.: УПЛ РГУ, 2004. — 43 с.
2. *Абрамян М. Э.* 1000 задач по программированию. Часть II: Минимумы и максимумы, одномерные и двумерные массивы, символы и строки, двоичные файлы. — Ростов н/Д.: УПЛ РГУ, 2004. — 42 с.
3. *Липпман С.*, *Лажойе Ж., Му Б.* Язык программирования C++. Вводный курс. — М.: Вильямс, 2007. — 896 с.
4. *Мейерс С.* Эффективное использование STL. — СПб.: Питер, 2002. — 224 с.
5. *Саттер Г.*, *Александреску А.* Стандарты программирования на C++. — М.: Вильямс, 2005. — 224 с.
6. *Сафонцев С. А.* Кредитно-модульная рейтинговая технология: Учебно-методическое пособие. — Ростов н/Д., 2008. — 16 с.
7. *Страуструп Б.* Язык программирования C++. — М.: «Издательство Бином»; СПб.: «Невский диалект», 1999. — 991 с.
8. *Эккель Б.*, *Эллисон Ч.* Философия C++: Практическое программирование. — СПб.: Питер, 2004. — 608 с.

Содержание