Министерство образования и науки
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Я. М. РУСАНОВА, М. И. ЧЕРДЫНЦЕВА

C++ КАК ВТОРОЙ ЯЗЫК В ОБУЧЕНИИ ПРИЕМАМ И ТЕХНОЛОГИЯМ ПРОГРАММИРОВАНИЯ

Учебное пособие

Ростов-на-Дону
Издательство Южного федерального университета
2010

УДК 004.43
ББК 32.973.2
Р 88

Рецензенты:
ст. преподаватель ЮФУ **В. Н. Брагилевский**,
к.ф.-м.н., доцент ЮФУ **М. Э. Абрамян**,
д.т.н., профессор РГУПС **М. А. Бутакова**

Русанова Я. М.

Р 88 С++ как второй язык в обучении приемам и технологиям программирования / Я. М. Русанова, М. И. Чердынцева. – Ростов н/Д : Изд-во ЮФУ, 2010. – 200 с.

ISBN 978-5-9275-0749-8

В данном учебном пособии внимание уделяется языку С++ и использованию объектно-ориентированного подхода. Оно состоит из трех модулей и проектных заданий к ним. Для закрепления знаний к каждому модулю даны вопросы для рубежного контроля, а также задания для самостоятельной работы. Адресовано студентам, обучающимся по бакалаврской программе по направлению «Прикладная математика и информатика».

ISBN 978-5-9275-0749-8

УДК 004.43
ББК 32.973.2

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Эта книга написана в качестве учебного пособия по курсу «Языки программирования». В данном издании мы стремились достичь двух целей: во-первых, облегчить изучение языка C++ студентам, имеющим базовые знания по основам программирования на языке Паскаль; во-вторых, продемонстрировать приемы решения задач с учетом особенностей языка C++.

Выражаем благодарность ст. преподавателю ЮФУ В. Н. Брагилевскому и доценту ЮФУ М. Э. Абрамяну, высказавшим замечания, которые помогли улучшить качество и ценность книги. Мы признательны доценту ЮФУ С. С. Михалковичу за возможность воспользоваться некоторыми интересными примерами, а также многим нашим коллегам, преподавателям и студентам, с которыми нас объединяют общие интересы.

ВВЕДЕНИЕ

В данной книге содержится материал, важный для понимания языка программирования. Материал не ориентирован на определенную версию компилятора, внимание уделяется именно языку C++ и использованию объектно-ориентированного подхода, а не особенностям конкретной реализации.

В модуле 1 дается неформальное введение в синтаксис языка C++. Изложение материала сопровождается разбором примеров решенных задач. При этом акцент делается не на рассмотрение алгоритмов и методов решения, а на особенности синтаксиса языка C++. Такой подход позволяет студентам, уже знакомым с базовыми алгоритмами, структурами данных и языком программирования Паскаль, достаточно легко перейти к использованию нового языка программирования.

В модуле 2 рассматриваются особенности представления и использования таких базовых структур данных языка C++, как массивы, строки и файлы; делается акцент на работу с указателями и адресной арифметикой и функциями в языке C++. Раскрываются возможности использования функций.

Модуль 3 содержит материал, который позволит студентам освоить объектно-ориентированные аспекты языка C++. Изложение материала иллюстрируется подробным рассмотрением примеров решенных задач. Такой подход позволит студентам, уже знакомым с базовыми понятиями «класс», «объект», «инкапсуляция», «наследование», «полиморфизм», научиться применять теоретические знания при решении конкретных задач.

Представленный в учебном пособии материал используется в курсе «Языки программирования и методы трансляции» бакалаврской программы по направлению «Прикладная математика и информатика».

Для облегчения работы с текстом в книге приняты следующие соглашения:

- ✓ Коды программ, фрагменты примеров, операторы, классы, объекты, методы обозначены специальным шрифтом (`Courier`), что позволяет легко найти их в тексте.
- ✓ Для указания имен файлов и каталогов используется шрифт Arial.
- ✓ Важные термины, встречающиеся впервые, выделены *курсивом*.
- ✓ Определения, термины, объяснения для запоминания предваряются специальным символом 📖.
- ✓ Более подробные объяснения отмечены специальным символом 📚.
- ✓ Знак ➢ означает, что проводятся сравнения языка C++ с языками Паскаль или C.
- ✓ Специальный символ ☺ означает рекомендации по стилю написания программ.
- ✓ Предостережения от ошибок начинаются со знака ☠.
- ✓ Упражнения, которые необходимо выполнить по ходу изучения, обозначены специальным символом 🔔.
- ✓ В конце каждого модуля определены требования к знаниям 👍 и умениям 🖐, которые должен получить студент после изучения модуля.

МОДУЛЬ 1. НАЧАЛЬНЫЕ СВЕДЕНИЯ О ПРОГРАММИРОВАНИИ НА ЯЗЫКЕ C++

Задачи и цели: Ознакомиться с организацией простейшей программы на языке C++, способами описания и использования простых функций, операциями ввода-вывода. Рассмотреть принципы организации программ на C++ в виде нескольких исходных модулей. Изучить структуру и назначение заголовочных файлов. Ознакомиться с базовыми типами в языке C++, включая тип указатель. Рассмотреть особенности некоторых операций, особенности правил построения и использования выражений на языке C++. Ознакомиться с возможностями структурных операторов языка.

Требуемый начальный уровень подготовки. Владение навыками программирования средствами языка Паскаль. Знакомство с процедурными приемами программирования. Знакомство с модульной структурой проектов.

1.1. Используемые термины

Термины «операция», «выражение», «оператор» вводились формально при изучении языка программирования Паскаль, поэтому их использование позволит легко провести аналогии и отметить различия в синтаксисе этих двух языков программирования.

Операция с точки зрения компилятора языка C++ – это команда, которая может иметь один, два или три операнда, возвращающая результат.

Операциями являются, например: +, –, ++, =, >, ==, `new`, &, `sizeof`, « , » – операция запятая и пр. Для классов имеется возможность перегрузки операций.

Из литералов, идентификаторов, операций и специальных символов строятся выражения. *Выражения* предназначены для того, чтобы вычислить значение либо достичь каких-либо побочных эффектов.

Операторы определяют и контролируют то, что и как делает программа. Операторы языка C++ делятся на: операторы-выражения, объявления, составные операторы (блоки), операторы выбора, циклы, операторы перехода и операторы обработки исключений.

 Потребность остановиться на используемой в книге терминологии вызвана прежде всего тем, что в имеющейся литературе по языку C++ встречается противоречивое употребление основных терминов. В большей степени это связано с тем, что в разных контекстах оригинальные термины даются в разных переводах. При этом из-за вольностей перевода многие особенности синтаксиса языка стираются.

Основные терминологические проблемы возникают вокруг трех понятий:

- `operator` (в переводе встречаются варианты: «оператор» и «операция»);
- `expression` («выражение»);
- `statement` (в переводе встречаются варианты: «инструкция» и «оператор»).

В русскоязычной литературе строгое определение понятия «оператор» отсутствует. В силу особенностей перевода в русскоязычной литературе, особенно переводной, именно для именования инструкций применяется термин «оператор». Но при этом оператором называют и знаки операций +, – и т. д. Некоторые авторы даже оправдывают это тем, что трудно перепутать оператор + с оператором цикла. С этим можно поспорить хотя бы в случае с оператором присваивания, поскольку с точки зрения синтаксиса инструкция присваивания – это оператор присваивания, завершающийся точкой с запятой. Следует также заметить, что при этом некоторые авторы используют термин «перегрузка операции», который непонятно, что обозначает, если не вводится понятие «операция».

Можно согласиться, что в каждом конкретном контексте легко однозначно определить, о чем идет речь. Но если изучаемый язык программирования интересует нас не только как инструмент создания программ, но и как объект для понимания принципов и методов компиляции, необходимо внести формализм в используемую терминологию.

Размышляя над проблемами терминологии, авторы решили, что проще изменить перевод англоязычных терминов, чтобы использовать привычные понятия.

1.2. Первые шаги

Прежде всего отметим первое правило.

 Язык C++ (как и язык C) *чувствителен к регистру*. В нем различаются символы верхнего и нижнего регистров.

Пример 1. Реализовать программу, выводящую на экран фразу «Hello, world!».

Программа «Hello, world!» в виде консольного приложения – типичный пример, упоминаемый практически во всех учебниках. Текст этой программы на языке C++ выглядит следующим образом:

```
#include <iostream>
int main(){
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

Первая строка содержит директиву препроцессора:

```
#include <iostream>
```

В языке C++ используются специальные файлы заголовков (header), которые содержат информацию, необходимую компилятору. Директива препроцессора `#include` сообщает о том, какой заголовочный файл должен быть включен в исходный текст программы.

В старых (до стандарта 1998 г.) версиях компиляторов языка C++ заголовочные файлы стандартной библиотеки имеют расширение .h, которое употреблялось ранее в языке C. В этом случае директива препроцессора будет выглядеть так:

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

Подключение файла <iostream> необходимо для того, чтобы можно было использовать для вывода объект `cout`. Наиболее просто организовать ввод-вывод в программах на C++, используя стандартные потоковые объекты `cin` и `cout`. Эти объекты связаны со стандартными устройствами консольного ввода и вывода – клавиатурой и дисплеем.

Поскольку имена `cin` и `cout` в приводимой программе не описаны, необходимо указать пространство имен, в котором они определены. Все стандартные идентификаторы C++ определены в пространстве имен `std`.

При обращении к объектам, объявленным в этом пространстве, имя пространства имен может быть указано явно. Для этого используется операция разрешения области видимости `::`. Такой способ удобно использовать, если к какому-то пространству имен потребуется одно-два обращения. В противном случае, чтобы не загромождать текст программы, лучше выносить объявление необходимого пространства имен, используя оператор `using namespace`.

Тогда текст программы «Hello, world!» будет выглядеть так:

```
#include <iostream>
using namespace std;
int main() {
  cout << "Hello, world!" << endl;
  return 0;
}
```

Использование пространств имен позволяет не заботиться о возможности употребления одинаковых имен для разных объектов при организации больших программ.

Для вывода используются объект `cout` и операция `<<`. Операция `<<` может использоваться последовательно многократно. Символьные строки при выводе должны заключаться в двойные кавычки. Манипулятор `endl` (определен в пространстве имен `std`) очищает буфер вывода и добавляет в поток `cout` символ новой строки.

📖 Выполнение программы на C++ всегда начинается с вызова функции с именем `main()`.

Функция `main()` должна вернуть управление операционной системе после своего завершения и сообщить код возврата. В случае аварийного завершения операционной системе будет сообщено ненулевое целочисленное значение – код ошибки (код возврата). В случае безошибочной работы нужно возвратить признак нормального завершения `0`. Именно поэтому в

заголовке функции `main()` указано, что возвращаемый ею результат должен быть целого типа, а последним оператором тела функции `main()`является оператор возврата:

```
return 0;
```

Пример 2. Найти максимум из последовательности *n* вводимых целых чисел.

Рассмотрим текст программы FindMax.cpp. Например, он будет выглядеть следующим образом:

```
// FindMax.cpp
#include <iostream>
using namespace std;
int main() {
  int n, max;
  cout<<"Input n"<<endl;
  cin>>n;
  cout<<"Input "<<n<<" elements"<<endl;
  cin>>max;
  for ( int i=1;i<n;i++) {
    int x;
    cin>>x;
    if (x>max)
      max=x;
  }
  cout<<"Maximum = "<<max<<endl;
  return 0;
}
```

В операторах ввода-вывода указываются имена стандартных потоковых объектов `cin` и `cout`.

```
  cout<<"Input n"<<endl;
  cin>>n;
  cout<<"Input "<<n<<" elements"<<endl;
  cin>>max;
```

Эти объекты связаны со стандартными устройствами консольного ввода и вывода – клавиатурой и дисплеем.

Для ввода данных объект `cin` использует операцию `>>`. Справа от этого знака находится переменная, принимающая вводимую информацию.

В процессе ввода последовательность символов, вводимых с клавиатуры, преобразуется к типу, соответствующему переменной, принимающей информацию. Если это невозможно, генерируется ошибочная ситуация.

Использование объектов `cin` и `cout` возможно, потому что в программе подключен файл <iostream> и объявлено пространство имен `std`, где определены эти объекты. Напомним также, что операции >> и << могут быть применены последовательно многократно.

Далее рассмотрим фрагмент программы:

```
for (int i=1;i<n;i++) {
  int x;
  cin>>x;
  if (x>max)
    max=x;
}
```

В нем задействованы управляющие конструкции `if` и `for`.

Обычно рекомендуют объявлять переменные непосредственно перед их использованием. Переменная `x`, к примеру, нужна только в теле цикла, то же самое относится к счетчику цикла `i`. При таком объявлении время жизни переменных `i`, `x` заканчивается за пределами данного цикла.

Переменные могут определяться в управляющих выражениях циклов `for` и `while`, в условиях оператора `if` и критериях выбора оператора `switch`.

Оператор `if-else` существует в двух вариантах: с секцией `else` и без нее.

Первый вариант:

```
if (выражение)
  оператор
```

Второй вариант:

```
if (выражение)
  оператор
else
  оператор
```

В программе представлен первый вариант:

```
if (x>max)
  max=x;
```

📖 В C++ выражение в условном операторе может быть любого типа.

Результат выражения, равный нулю, интерпретируется как «ложь», а любое ненулевое значение как «истина». В примере использовано логическое выражение.

➢ В отличие от языка Паскаль «выражение» в условном операторе всегда заключается в скобки.

Под оператором в языке C++ понимается либо одиночный оператор, либо блок операторов в фигурных скобках (аналог составного оператора в языке Паскаль).

➢ В отличие от языка Паскаль любой оператор, кроме блока, завершается символом точки с запятой (;). Поэтому перед `else` может стоять символ точки с запятой (;).

Общая форма цикла `for` выглядит так:

```
for (инициализация; условие; изменение)
  оператор
```

В данном примере можно провести аналогию оператора `for` в языке C++ и цикла `for` в языке Паскаль. В действительности возможности цикла `for` в C++ гораздо шире.

1.3. Функции как строительные блоки программы

📖 Программа на языке C++ состоит из одной или нескольких функций, объявлений и определений глобальных объектов, подключения пространства имен и заголовочных файлов.

Выполнение C++ программы начинается выполнением функции `main()`. Среди функций должна быть одна и только одна с именем `main()`.

В частном случае программа может состоять только из функции `main()`, подключения пространства имен и заголовочных файлов.

Пример 3. Описать две функции для обмена значениями двух переменных, соответственно, целого и вещественного типа.

```
#include <iostream>
using namespace std; //использовать пространство имен std
/*объявления функций
  сами функции описаны после функции main()
*/
void my_swap(double &a, double &b);
void my_swap(int &a, int &b);

int main() {
  int k,m;
  cout<<"enter two integer values:";
  cin>>k>>m;
  cout<<k<<"  "<<m<<endl;
  my_swap(k, m);
  cout<<k<<"  "<<m<<endl;
  double a,b;
  cout<<"enter two double values:";
  cin>>a>>b;
  cout<<a<<"  "<<b<<endl;
  my_swap(a,b);
  cout<<a<<"  "<<b<<endl;
  return 0;
}
void my_swap(double &a, double &b) {
  double r = a;
  a=b;
  b=r;
}

void my_swap(int &a, int &b) {
  int r=a;
  a=b;
  b=r;
}
```

Разберем этот пример подробно. В программе продемонстрировано использование двух типов комментариев: первый является однострочным, он начинается с пары символов // и заканчивается концом строки; далее в тексте программы помещен многострочный комментарий, который начинается с пары символов /* и заканчивается парой символов */. Многострочный комментарий может занимать одну или несколько строк, а также только часть строки.

☺ Правила хорошего стиля рекомендуют помещать символы, завершающие комментарий, в начале новой строки под соответствующими им символами начала комментария, а сам комментарий располагать с отступом.

Комментарии не являются синтаксическими единицами, но играют важную роль в оформлении текста программы.

В следующих двух строках программы содержатся объявления заголовков двух пользовательских функций.

```
void my_swap(double &a, double &b);
void my_swap(int &a, int &b);
```

Необходимость помещения их в тексте программы обусловлена двумя правилами: правилом синтаксиса языка C++ и правилом стиля написания программ на C++.

📖 Всякое имя, прежде чем будет использовано, должно быть описано.

Это правило относится и к именам функций.

В C и C++ вместо одного понятия – описание объектов программы – вводятся два понятия – объявления и определения. Нужно хорошо понимать разницу между объявлениями и определениями.

📖 *Объявление* сообщает компилятору некоторое имя (идентификатор).

Фактически объявление означает: «Эта функция или переменная где-то существует и выглядит так». Определение означает: «Создай здесь эту переменную» или «Создай здесь эту функцию».

📖 *Определение* сообщает компилятору о необходимости создания в памяти объекта с указанным именем. Это относится как к переменным, так и к функциям.

В случае переменной объявление и определение часто совпадают.

Расположение определений самих функций в программе может быть любым. Однако…

☺ Правило стиля рекомендует всегда в программе располагать определения всех функций либо до функции `main()`, либо после функции `main()`.

Если определение функций располагается после функции `main()`, используется предварительное описание заголовка функции (т. е. ее объявление). Объявления нужны и в том случае, когда программа имеет многомодульную структуру, а описание функции располагается не в том файле, где она используется.

В объявлении функций достаточно указать типы параметров, а имена можно опускать, так как компиляторы их игнорируют. Хотя чтобы сделать программу понятнее, имена переменных включают в объявления функций.

После объявления функций `my_swap` располагается определение функции `main()`.

В общем случае определение функции состоит из заголовка функции и тела функции. Тело функции представляет собой блок или составной оператор, поэтому должно быть ограничено символами { и }.

Тело функции `main()` в приведенном примере представляет собой линейную программу, содержащую операторы следующих видов:

✓ операторы объявления (или описания)

```
int k,m;
double a,b;
```

✓ операторы ввода

```
cin>>k>>m;
cin>>a>>b;
```

✓ операторы вывода

```
cout<<"enter two integer values:";
cout<<k<<"  "<<m<<endl;
cout<<"enter two double values:";
cout<<a<<"  "<<b<<endl;
```

✓ операторы вызова функции

```
my_swap(k,m);
my_swap(a,b);
```

✓ оператор возврата результата выполнения функции

```
return 0;
```

Каждый оператор в языке C++ должен завершаться точкой с запятой – этим он отличается от выражения или операции.

Поскольку в языке C++ отсутствует явно выделенный раздел описаний, описания локальных переменных могут встречаться в любом месте блока. Но при этом не следует забывать о правиле: всякое имя, прежде чем может быть использовано, должно быть описано.

📖 В языке C++ определение одной функции нельзя вкладывать в определение другой функции.

Рассмотрим операторы вызова функции.

```
my_swap(k,m);
my_swap(a,b);
```

Основные действия по обработке данных в языке C++, как и в других процедурных языках, реализуются в виде подпрограмм.

В C++ единственным видом подпрограммы является функция. Функция может быть вызвана как операция, т. е. использована в выражении соответствующего типа. Кроме этого функция может быть вызвана оператором вызова функции.

Оператор вызова функции (в отличие от операции вызова функции) используется в одном из следующих случаев:

- ✓ если результат функции не будет использован, а функция является функцией с побочным эффектом;
- ✓ когда функция является функцией с побочным эффектом, но не возвращает результата. В последнем случае функция семантически эквивалентна процедуре.

В рассматриваемом примере имеет место как раз второй вариант. Обе функции не возвращают результат. Это видно из их объявлений

```
void my_swap(double &a, double &b);
void my_swap(int &a, int &b);
```

Ключевое слово `void` означает, что функция не возвращает никакого значения. Следовательно, такая функция может быть использована только в операторе вызова функции.

Обе функции реализуют один и тот же алгоритм – поменять местами значения двух переменных. Но одна из функций предназначена для перестановки значений целых переменных, а другая – для перестановки вещественных.

Объявление двух функций с одинаковыми именами означает, что имеет место *перегрузка* имени функции.

Перегруженные функции обязательно должны иметь разные *сигнатуры*, т. е. должны отличаться своими списками параметров.

Вернемся к операторам вызова функции.

```
my_swap(k,m);
my_swap(a,b);
```

Именно при их анализе компилятор принимает решение о том, какой из вариантов перегруженной функции будет вызван.

В операторе вызова `my_swap(k,m)` в качестве фактических параметров используются две целые переменные `k` и `m`. Значит, будет вызвана та версия перегруженной функции, у которой параметрами являются целые переменные. Оператор вызова `my_swap(a,b)` использует два фактических параметра вещественного типа. Для него будет использован вызов функции с двумя вещественными параметрами.

Рассмотрим определения этих двух перегруженных функций.

```
void my_swap(double &a, double &b) {
  double r = a;
  a=b; b=r;
}

void my_swap(int &a, int &b) {
  int r=a;
  a=b; b=r;
}
```

Как известно, для того чтобы результат изменения значений формальных параметров был заметен в точке их вызова, необходимо использовать передачу параметров по ссылке.

> ➢ Возможность передавать параметры по ссылке появилась только в C++. В языке C из-за отсутствия такой возможности требовалось передавать указатели.

При этом параметрами, переданными по ссылке, в теле функции мы оперируем как обычными переменными.

☺ Стиль программирования на языке C++ рекомендует использовать передачу параметров по ссылке, вместо передачи указателей.

1.4. Многофайловый проект, включение заголовочных файлов

Напомним требования к структуре программы. Программа на языке C++ состоит из одной или нескольких функций, объявлений и определений глобальных объектов, подключения пространства имен и заголовочных файлов. Среди функций должна быть одна и только одна с именем `main()`.

Все функции, составляющие программу, могут быть расположены в одном файле – исходном модуле. Реальные программы на C++, как правило, состоят из нескольких исходных модулей. Это возможно благодаря тому, что C++ и C поддерживают раздельную компиляцию модулей.

Пример 4. Дано целое число. Определить количество и сумму цифр в десятичной записи этого числа. Выдать само число и число, получающееся из него при вычеркивании первой и последней цифр.

Оформим действия определения количества цифр в десятичной записи числа, вычисления суммы цифр числа, преобразования числа путем вычеркивания из него первой и последней цифры в виде функций.

Рассмотрим сначала вариант, когда все пользовательские функции и функция `main()` располагаются в одном файле.

```
#include <iostream>
using namespace std;
/*   объявления функций
     сами функции описаны после функции main()
*/
int count_dig(int a);//параметр передается по значению
int sum_dig(int a);
void del_last_dig(int& a);//параметр передается по ссылке
int del_first_dig(int& a);
```

```
int main() {
  int x;
  cin>>x;
  cout<<count_dig(x)<<endl;
  cout<<sum_dig(x)<<endl;
/*следующие две функции изменяют значение параметра, поэтому
  сохраним исходное число во вспомогательной переменной*/
  int r=x;
  del_last_dig(r);
  cout<<x<<' '<<r<<endl;
  cout<<del_first_dig(r)<<' '<<r;
}

int count_dig(int a) {
  int k=0;
  while (a!=0) {
    k++;       //операция увеличения
    a/=10;     //составная операция присваивания
  }
  return k;
}

int sum_dig(int a) {
  int s=0;
  a=abs(a);
  while (a!=0) {
    s+=a%10;
    a/=10;
  }
  return s;
}

void del_last_dig(int& a) {
  a/=10;
}

int del_first_dig(int& a) {
  int k=count_dig(a);
  int r=1;
  for (int i=0; i<k-1; i++)
    r*=10;
  a%=r;
  return a;
}
```

Перед определением функции `main()` расположены *объявления* (заголовки) пользовательских функций.

➢ В отличие от языка С в С++ рекомендуется всегда объявлять функции.

Объявления необходимы в том случае, когда программа имеет многомодульную структуру и определение функции располагается не в том файле, где она используется. Поэтому объявление функций является хорошей подготовкой к организации многомодульной структуры программы.

Первые две функции – для вычисления количества и суммы цифр числа – используют *передачу параметров по значению*.

```
int count_dig(int a);//параметр - значение
int sum_dig(int a);
```

Если фактический параметр будет передаваться по значению, то при описании формального параметра указываются только его имя и тип.

В функциях, которые вычеркивают одну из цифр числа (первую или последнюю), параметр *передается по ссылке*.

```
void del_last_dig(int& a);
//параметр-ссылка будет изменен в функции
int del_first_dig(int& a);
```

Чтобы определить способ передачи параметра по ссылке, после указания типа формального параметра ставится модификатор ссылки &.

Различие в объявлениях двух последних функций не связано с различием в алгоритмах соответствующих функций. Оно призвано только продемонстрировать два стиля описания и использования функций. В первом случае описание более соответствует стилю описания процедур: функция не возвращает никакого результата. Второй вариант описания более соответствует стилю описания функций в языках С и С++. Если в результате выполнения функции изменяется один из параметров, то полученное после изменения значение возвращается как результат функции. В коде функции `main()` видно, что функцию `del_last_dig()` необходимо вызывать опе-

ратором вызова функции, а затем использовать полученное значение фактического параметра.

Функцию `del_first_dig()` можно вызвать аналогичным образом, но в программе показано, что возвращаемое функцией значение можно использовать сразу, например, поместив его в поток вывода. Для сравнения сразу за значением, возвращаемым функцией, выдается изменившееся значение фактического параметра.

При реализации функций, кроме уже рассмотренных операторов, используются операторы:

- ✓ цикла с предусловием

```
while (a!=0){
  k++;
  a/=10;
}
```

Оператор цикла с предусловием является универсальным оператором цикла:

```
while (условие)
  оператор;
```

- ✓ инкремента

```
k++;
```

Операция инкремента `k++` является сокращенной формой операции присваивания следующего вида: `k=k+1`. Однако компилятор языка при трансляции такой операции должен использовать более эффективную реализацию – заменить, по возможности, операцию сложения на машинную операцию увеличения (`inc`). Аналогичный смысл имеет операция декремента (`--`). Операция присваивания, завершающаяся точкой с запятой, представляет оператор присваивания.

- ✓ комбинированные или составные операторы присваивания

```
a/=10;
s+=a%10;
r*=10;
```

Составная операция присваивания вида `s+=a` является сокращенной записью операции присваивания вида `s=s+a`.

Для вычисления остатка от деления двух целых чисел используется операция %. Операция деления / для целых операндов определена как целочисленное деление.

Разместим теперь описание функций, работающих с цифрами десятичного представления числа в отдельном файле digit.cpp.

С точки зрения любого компилятора языка C++, файл является единицей компиляции – исходным модулем. В результате компиляции каждого исходного модуля (в случае отсутствия ошибок) получаются объектные модули. Каждому исходному модулю будет соответствовать свой объектный модуль. Такая структура программы называется многомодульной. Если при этом компилятор может компилировать каждый исходный модуль по отдельности, то говорят, что компилятор поддерживает принцип раздельной компиляции.

В этом случае содержимое файла digit.cpp будет выглядеть так:

```
int count_dig(int a) {
  int k=0;
  while (a!=0) {
    k++;       //операция увеличения
    a/=10;     //составная операция присваивания
  }
  return k;
}

int sum_dig(int a) {
  int s=0;
  a=abs(a);
  while (a!=0) {
    s+=a%10;
    a/=10;
  }
  return s;
}
```

```
void del_last_dig(int& a) {
  a/=10;
}

int del_first_dig(int& a) {
  int k=count_dig(a);
  int r=1;
  for (int i=0; i<k-1; i++)
    r*=10;
  a%=r;
  return a;
}
```

Содержимое файла довольно необычно для C++, поскольку не содержит ни одной команды подключения заголовочных файлов. Если попробовать откомпилировать такой файл отдельно, то выдается сообщение об ошибке компиляции. Ошибка означает, что имя функции `abs()` неизвестно, необходимо подключить заголовочный файл, в котором находится объявление функции `abs()` для целочисленного аргумента. Чтобы исправить эту ошибку, достаточно подключить заголовочный файл <cstdlib>:

```
#include <cstdlib>
```

Почему, когда программа была представлена одним файлом, не понадобилось подключать этот заголовочный файл? Это связано с тем, что многие заголовочные файлы C++ сами подключают другие заголовочные файлы. Такая ситуация имеет место и в случае использования заголовочного файла <iostream>.

Файл, содержащий функцию `main()`, будет выглядеть следующим образом:

```
#include <iostream>
using namespace std;
/*   объявления функций
     сами функции описаны после функции main()
*/
int count_dig(int a);//параметр передается по значению
int sum_dig(int a);
void del_last_dig(int& a);//параметр передается по ссылке
int del_first_dig(int& a);
```

```
int main() {
  int x;
  cin>>x;
  cout<<count_dig(x)<<endl;
  cout<<sum_dig(x)<<endl;
/*следующие две функции изменяют значение параметра, поэтому
  сохраним исходное число во вспомогательной переменной
*/
  int r=x;
  del_last_dig(r);
  cout<<x<<' '<<r<<endl;
  cout<<del_first_dig(r)<<' '<<r;
}
```

В полученном варианте программы присутствует еще один недостаток – явное объявление всех функций, вынесенных нами в отдельный файл, перед описанием функции `main()`. Однако убрать их нельзя, так как в этом случае все имена функций станут неизвестными и при компиляции будет получено сообщение об ошибках, аналогичное предыдущему.

Очевидно, что хорошим решением в этом случае было бы наличие заголовочного файла, содержащего объявления всех функций, вынесенных в отдельный файл. Это удобно тем, что не нужно размещать в основном файле, содержащем функцию `main()`, объявления всех функций. Достаточно подключить соответствующий h-файл.

В нашем случае это будет заголовочный файл digit.h с таким содержимым:

```
int count_dig(int a);
int sum_dig(int a);
void del_last_dig(int& a);
int del_first_dig(int& a);
```

Подключение заголовочного файла, используемого в проекте, выглядит так:

```
#include "digit.h"
```

☺ Правило хорошего стиля требует в случае вынесения функций в отдельный cpp-файл создавать соответствующий ему h-файл, содержащий объявления соответствующих функций.

Рекомендуется, чтобы имена cpp-файла и соответствующего ему h-файла совпадали.

Что же стоит включать в заголовочные файлы? Основное правило рекомендует ограничиться одними объявлениями. В заголовочном файле не должно быть определений функций.

1.5. Заголовочные файлы и библиотеки в C++

Многие библиотеки содержат большое количество функций и переменных. Чтобы избавить программиста от лишней работы и обеспечить логическую последовательность внешних объявлений, в C и C++ используется *механизм заголовочных файлов*.

Создатель библиотеки представляет заголовочный файл. Для того чтобы объявить в программе функции и внешние переменные этой библиотеки, пользователь просто включает в программу заголовочный файл препроцессорной директивой `#include`.

Заставляя правильно работать с заголовочными файлами, язык C++ гарантирует согласованность библиотек и сокращает количество ошибок благодаря всеобщему использованию одного и того же интерфейса.

Существует достаточно большое количество стандартных библиотечных и соответствующих им заголовочных файлов в языках C и C++.

Пример 5. Найти все простые числа в диапазоне от 2 до N.

Рассмотрим программу для решения задачи поиска простых чисел.

Листинг Prime.cpp – первый вариант

```
#include <cmath>
#include <iostream>
using namespace std;
bool IsPrime(int x) {
  for (int i=2;i<=int(sqrt((double)x));i++) {
    int y=x%i;
    if (y=0)
      return false;
    }
  return true;
}

int main() {
  int N;
  cout<<"N = ";
  cin>>N;
  for (int i=2;i<=N;i++)
    if (IsPrime(i))
      cout<<i<<' ';
  cout<<endl;
  return 0;
}
```

В этом примере обратим внимание прежде всего на необходимость подключения заголовочного файла cmath. В нем находятся объявления математических функций и некоторые константы.

После запуска программы на выполнение можно увидеть следующие результаты:

```
N=10
2 3 4 5 6 7 8 9 10
```

Легко заметить, что не все выведенные числа являются простыми. Следовательно, программа содержит логические ошибки.

Исходя из текста программы, можно сделать вывод, что ошибки следует искать в реализации функции `IsPrime`. Анализ результатов работы программы показывает, что функция `IsPrime` всегда возвращает значение `true` (истина).

Если ошибка сразу не видна, то можно вставить выводы промежуточных результатов или выполнить трассировку функции `IsPrime`, т. е. реализовать ее пошаговое выполнение.

Проконтролируем значение переменной `y` после выполнения оператора

```
int y=x%i;
```

Запустим программу на выполнение при N=10.

Можно заметить, что хотя `y` и принимает значение 0, досрочный выход из цикла (и соответственно, из функции) не наблюдается никогда.

Значит, условный оператор

```
if (y=0)
     return false;
```

содержит ошибку. Действительно, в выражении `y=0` вместо операции сравнения используется операция присваивания, т. е. переменной `y` всегда присваивается 0, а это значит, что выражение всегда будет принимать значение `false` (ложь).

В правильном варианте программа имеет следующий вид:

Листинг Prime.cpp – второй вариант

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

bool IsPrime (int x) {
  for (int i=2;i<=int(sqrt((double)x));i++){
    int y=x%i;
    if (y==0)
      return false;
    }
  return true;
}

int main() {
  int N;
  cout<<"N = ";
  cin>>N;
```

```
  for (int i=2;i<=N;i++)
    if (IsPrime (i))
      cout<<i<<' ';
  cout<<endl;
  return 0;
}
```

Директива `#include` указывает препроцессору, что он должен открыть файл с заданным именем и вставить его содержимое на место директивы.

В угловых скобках обычно подключается заголовочный файл стандартной библиотеки, в двойных кавычках – заголовочный файл, определяемый программистом.

При включении файла в угловых скобках препроцессором обычно используется некоторая разновидность «пути поиска», задаваемого в переменной окружения или в командной строке компилятора. Способ определения пути поиска зависит от компьютера, операционной системы и реализации C++.

Директива с именем файла, заключенным в кавычки, сообщает препроцессору, что поиск заданного файла начинается с текущего каталога. Если файл не обнаружен, то директива обрабатывается примерно так, как если бы вместо кавычек использовались угловые скобки.

Библиотеки, унаследованные из языка C, сохранили традиционное для заголовочных файлов расширение `.h`. Впрочем, их можно использовать и при включении в современном стиле C++, для чего имя заголовочного файла снабжается префиксом `c`.

Рассмотрим следующий фрагмент:

```
#include <stdio.h>
#include <stdlib.h>
```

В современном варианте он выглядит так:

```
#include <cstdio>
#include <cstdlib>
```

При просмотре кода программы сразу понятно, когда в программе используются библиотеки C, а когда – библиотеки C++.

☺ В одной программе две формы подключения стандартных заголовочных файлов библиотек (с префиксом `c` или с расширением `.h`), унаследованных из языка C, лучше не смешивать.

Пример 6. Составить функцию для вычисления `n!` Использовать ее для вычисления биномиального коэффициента для заданных натуральных чисел `n` и `m` по формуле $C_n^m = \frac{n!}{m!(n-m)!}$.

Разместим определение функции в отдельном cpp-файле и создадим для этого cpp-файла заголовочный h-файл.

Листинг файла functions.cpp

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

Листинг файла binom.cpp

```
#include <iostream>
#include "functions.h"
using namespace std;
int main() {
  int n,m;
  cout<<"Input n and m"<<endl;
  cin>>n>>m;
  cout<<"result = "<<fact(n)/(fact(m)*fact(n-m))<<endl;
  return 0;
}
```

Листинг файла functions.h.

```
double fact(int);
```

Если сравнить содержимое заголовочного файла functions.h с содержимым заголовочных файлов стандартных библиотек языка C++, таких как

iostream, то можно заметить, что в начале нашего файла отсутствуют директивы препроцессора. Рассмотрим, для чего они используются.

Может оказаться, что заголовочный файл многократно включается в сложную программу. В результате возникают ошибки, связанные с многократным определением.

📖 Принцип единственного определения в C++: объявлений может быть сколько угодно, но определение должно быть только одно.

Чтобы предотвратить ошибки при многократном включении заголовочного файла, необходимо использовать специальные директивы препроцессора: `ifndef`, `endif`, `define`.

В каждом заголовочном файле следует сначала проверить, не был ли этот заголовочный файл уже включен в многофайловый проект. Это делается при помощи препроцессорного флага. Если флаг не установлен, значит, флаг еще не включался; установите флаг и выполните объявления. Если флаг уже установлен, то код файла с объявлениями игнорируется.

Примерный формат заголовочного файла:

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
// необходимые объявления
#endif    // HEADER_FLAG
```

Имя HEADER_FLAG можно заменить любым уникальным именем, но существует надежная схема выбора имен.

☺ В качестве имени препроцессорного флага рекомендуется записать имя заголовочного файла символами верхнего регистра и заменить точку символом подчеркивания.

Например, для файла с именем simple.h можно, следуя данному правилу, получить имя SIMPLE_H.

```
#ifndef SIMPLE_H
#define SIMPLE_H
double fact(int);
#endif    //SIMPLE_H
```

Рекомендуется разбивать большой код на относительно независимые части и объединять в cpp-файлы группы взаимосвязанных функций.

1.6. Целочисленные типы данных

В языке C++ имеются две группы встроенных типов данных: базовые или фундаментальные (`fundamental`) и составные (`compound`).

> ➢ В C++ базовые типы являются эквивалентами машинных типов данных. В языке Паскаль встроенные типы данных (как простые – скалярные, так и составные – структурные) в большей степени являются отражением математических понятий, а не машинных типов данных.

Базовые типы в C++ служат для представления целых чисел и чисел с плавающей точкой. Существует несколько разновидностей встроенных типов для представления целочисленных значений и значений с плавающей точкой.

В порядке возрастания (а точнее, неубывания) размерности базовые целые типы – это: `char`, `short`, `int` и `long`. Каждый из этих типов подразделяется на две разновидности: со знаком (`signed`) и без знака (`unsigned`).

В языке C++, как и в его предшественнике языке C, действуют правила умолчания. По этим правилам типы `short`, `int` и `long`, если ничего не указано, являются знаковыми.

Данные типа `char` служат для хранения стандартных символов ASCII-кода. Если же необходимо использовать тип `char` для числовых

значений, то следует выбрать тип signed char для диапазона от –128 до +127, а тип unsigned char – для диапазона от 0 до 255.

Чтобы узнать размерность целых типов и диапазоны допустимых значений, можно воспользоваться средствами самого языка C++.

Прежде всего это операция sizeof, которая возвращает размер типа sizeof (имя типа) или переменной sizeof объект в байтах. Заметим, что скобки в операции sizeof требуются, если в качестве операнда используется имя типа, но не нужны в случае использования имени переменной. Строго говоря, sizeof возвращает беззнаковое целое, тип которого size_t определен в файле cstddef.

В заголовочном файле climits определены символические имена для представления предельных значений диапазонов типов. Например, для типа int диапазон значений задается символическими константами INT_MIN и INT_MAX.

В языке C++ появились целочисленные типы wchar_t и bool. Тип данных wchar_t служит для представления расширенного набора символов. Такой тип используется для представления двухбайтовых кодовых таблиц, например, UTF-16 или ISO.10646.

Ранее в языке C++, как и в C, данные логического типа отсутствовали. Вместо этого ненулевые значения интерпретировались как «истина» (true), а нулевые – как «ложь» (false). Тип bool позволяет использовать переменные логического типа и предопределенные константы true и false. Однако по-прежнему любое ненулевое значение представляет true, а нулевое – false, т. е. выполняется неявное приведение типа. Это неявное приведение типа осуществляется для любых числовых значений и для значений указателя. Допустимо и обратное неявное преобразование –

из типа `bool` в целый тип. Значение `true` представляется единицей, а `false` – нулем.

Пример 7. Найти наибольший общий делитель (НОД) целых чисел A и B.

Для вычисления НОД используем алгоритм, основанный на соотношении:

$$НОД(a,b) = \begin{cases} a, & a = b; \\ НОД(a-b,b), & a > b; \\ НОД(a,b-a), & a < b; \end{cases}$$

где $a > 0, b > 0.$

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

int gcd(int a, int b);
int main() {
  int a,b;
  cin>>a>>b;
  cout<< gcd(a,b);
  return 0;
}

int gcd(int a, int b) {
  a=abs(a);
  b=abs(b);
  while (a!=b) {
    if (a>b)
      a-=b;
    else
      b-=a;
  }
  return a;
}
```

Для того чтобы алгоритм можно было использовать и для отрицательных чисел, в начале функции `gcd()`избавляемся от знаков у аргументов.

Другой алгоритм, традиционно называемый алгоритмом Евклида, использует соотношение

$$НОД(a,b)=\begin{cases} a, & b=0; \\ НОД(b,\lfloor a\ /b \rfloor), & b\neq 0. \end{cases}$$

Рассмотрим только, как изменится функция, вычисляющая НОД.

```
int gcd (int a, int b) {
  while (b!=0) {
    int r;
    r=a%b;
    a=b;
    b=r;
  }
  return a;
}
```

Для вычисления остатка от деления двух целых чисел используется операция %.

Помимо привычных арифметических операций и операций сравнения для целочисленных типов (кроме типа `bool`), в C++ определены поразрядные операции. Эти операции предназначены для тестирования, установки или сдвига отдельных битов в байтах или машинных словах и чаще всего используются для решения задач программирования системного уровня.

Хотя бинарные поразрядные операции, за исключением операций сдвига, являются коммутативными, традиционно принято левый операнд считать проверяемым или подвергаемым изменению. При этом правый операнд задает принцип проверки или изменения левого операнда. В этом случае правый операнд называется *маской*. В табл. 1 приведено краткое описание семантики поразрядных операций в предположении, что правый операнд является маской.

Пример 8. Написать программу, демонстрирующую использование поразрядных операций для опроса содержимого байта.

```
#include <iostream>
using namespace std;
void dispBinary (unsigned char u);
int main() {
```

```
  unsigned char u;
  cout<<"input number between 0 and 255 :";
  cin >> u;
  cout <<"number in binary code:";
  dispBinary(u);
  cout << "addition to unity: ";
  dispBinary(~u);
  return 0;
}
void dispBinary (unsigned char u) {
  unsigned char t;
  for (t=128; t>0; t=t>>1)
    if (u&t)
      cout <<"1";
    else
      cout <<"0";
  cout<<"\n";
}
```

Таблица 1

Краткое описание семантики поразрядных операций

Операция	Значение	Комментарий
&	Поразрядное И (and)	0 в разряде маски гарантирует установку 0 в соответствующих битах результата; 1 в маске проверяет наличие установленной 1 в соответствующем разряде левого операнда.
|	Поразрядное ИЛИ (or)	1 в разряде маски гарантирует установку 1 в нужных битах; 0 в маске проверяет наличие 0 в левом операнде.
^	Поразрядное исключающее ИЛИ (xor)	Дважды примененная операция с одной и той же маской восстанавливает значение левого операнда.
>>	Поразрядный сдвиг вправо	Левый операнд определяет значение, подвергаемое сдвигу, правый операнд задает величину сдвига в битах. Для значений со знаком значение знакового разряда сохраняется. Для беззнаковых значений при сдвиге свободные разряды дополняются нулями.
<<	Поразрядный сдвиг влево	Левый операнд определяет значение, подвергаемое сдвигу, правый операнд задает величину сдвига в битах. В левый освобождающийся разряд при каждом сдвиге записывается ноль.
~	Поразрядное отрицание НЕ (not)	Унарная операция, инвертирует состояние всех битов своего операнда, результат – дополнение операнда до 1.

Для анализа двоичного представления числа используется однобайтовое число без знака (`unsigned char`). Данный тип допускает представление значений в диапазоне от 0 до 255. Функция `dispBinary()` в цикле сравнивает данное число с маской `t`.

```
unsigned char t;
for (t=128; t>0; t=t>>1)
  if (u&t)
    cout <<"1";
  else
    cout <<"0";
```

В маске в одном разряде устанавливается единица, в остальных – нули. Единица в маске последовательно сдвигается от самого левого разряда до самого правого. После последнего сдвига все разряды становятся нулевыми.

Начальное значение параметра цикла `t`, используемого как маска, число 128 соответствует двоичному 10 000 000. На каждом шаге итерации единица в записи числа `t` сдвигается на одну позицию вправо. Такие значения маски позволяют выделять двоичные разряды в числе слева направо. Если в разряде, определяемом маской `t`, стоит 1, поразрядное & даст в этом разряде ненулевой результат.

В данном примере видно, что в операторе цикла `for` выражение итерации может быть любым. Хотя обычно важно только, чтобы это выражение влияло на значение переменных, входящих в условие так, чтобы цикл когда-нибудь завершился.

Если вставить в тело цикла вспомогательный оператор, печатающий значение параметра цикла `t`, или воспользоваться средствами отладчика визуальной среды программирования, можно убедиться, что поразрядный сдвиг вправо соответствует операции деления на два.

Пример 9. Разработать функции для преобразования латинских букв в нижний или верхний регистр.

```
char low(char c) {
  return c | 32;
  //двоичное представление числа 32 – 00100000
}
char upp(char c) {
  return c & 223;
  //двоичное представление числа 223 – 11011111
}
```

Эти функции основаны на том, что в наборе символов ASCII коды прописных (заглавных) и строчных латинских букв отличаются только в пятом разряде (разряды нумеруются с нуля справа налево). У заглавных букв этот разряд равен нулю, а у прописных – единице.

Для того чтобы поставить в пятом разряде единицу, сохранив значение остальных разрядов, используется побитовая операция «|» и двоичная маска 00100000, которая соответствует десятичному числу 32. Чтобы поставить в пятом разряде ноль, сохранив значение остальных разрядов, используется побитовая операция «&» и двоичная маска 11011111, которая соответствует десятичному числу 223.

1.7. Типы данных для вещественных значений

Вещественные числа представляются с помощью типов данных с плавающей точкой. В языке C++ предусмотрены три типа данных с плавающей точкой: `float`, `double` и `long double`. Основным типом для операций с вещественными числами является тип `double`.

> ➢ В отличие от языка Паскаль, в языке C++ операция деления (/), как и все остальные арифметические операции, является полиморфной, т. е. тип ее результата зависит от типа операндов.

Если оба операнда целочисленные, то операция будет соответствовать делению нацело, а если хотя бы один операнд вещественный, то и результат будет вещественным. Это следует учитывать при программировании выражений. Например, в следующей программе при вычислении суммы, чтобы получить правильное значение очередного слагаемого, используется явная запись числителя в виде вещественной константы.

Пример 10. Вычислить сумму $\sum_{i=1}^{n}\frac{1}{i}$.

```
#include <iostream>
using namespace std;
double sum_1(int n);
double sum_2(int n);
int main() {
  int n;
  cout<<”input number of terms”<<endl;
  cin>>n;
  cout.precision(20);
  cout<<sum_1(n)<<endl;
  cout<<sum_2(n)<<endl;
  return 0;
}

double sum_1(int n) {
  double s=0;
  for (int i=1; i<n+1; i++)
    s+= 1.0/i;
  return s;
}
double sum_2(int n) {
  double s=0;
  for (int i=n; i; i--)
    s+= 1.0/i;
  return s;
}
```

В примере вычисление конечной суммы осуществляется с помощью двух функций, отличающихся только организацией цикла суммирования. Первая функция выполняет суммирование, начиная с первого слагаемого до последнего. Вторая функция выполняет суммирование в обратном по-

рядке – от последнего слагаемого до первого. Задавая большие значения (например, 2000) количества суммируемых слагаемых `n`, можно увидеть, что результаты суммирования начинают отличаться из-за накопления ошибки округления (рис. 1).

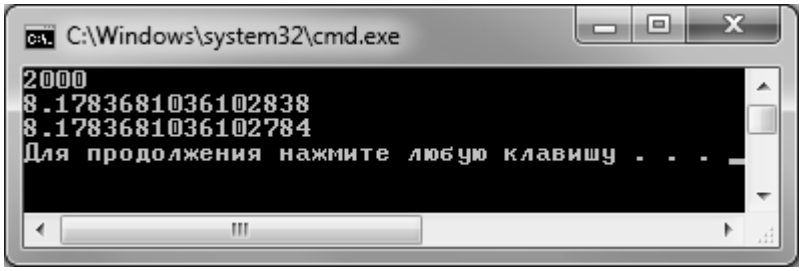


Рис. 1. Результат выполнения при *n* = 2000

В операторе объявления

```
double s=0;
```

продемонстрирована возможность использования инициализатора при объявлении.

Операторы цикла `for`

```
for (int i=1; i<n+1; i++) оператор
for (int i=n; i; i--) оператор
```

семантически эквивалентны операторам цикла `for` в языке Паскаль. Но в отличие от них оператор цикла `for` в языке С может содержать в части инициализации простое объявление. При этом областью видимости для объявленных в части инициализации объектов является оператор цикла, время жизни таких переменных также ограничено оператором цикла.

В качестве выражения итерации в вышеприведенных циклах используются выражения инкремента и декремента (`i++`, `i--`).

Во втором операторе цикла показано, что значение переменной может быть использовано в качестве выражения условия, поскольку любое ненулевое значение соответствует `true`, а нулевое – `false`.

Так как логическое выражение `i<n+1` проверяется на каждом шаге, то и `n+1` вычисляется на каждом шаге. Это неэффективно, поэтому не рекомендуется включать вычисляемые выражения в условия продолжения цикла.

Оператор в теле цикла представляет собой оператор присваивания. В выражении присваивания использована сокращенная операция присваивания:

```
s+= 1.0/i;
```

Делимое записано в виде вещественной константы. Если этого не сделать, операция деления будет распознана как целочисленное деление.

Наконец, необходимо обратить внимание на то, что прежде чем выводить результаты вычисления значений функций, следует воспользоваться вызовом функции для объекта `cout`:

```
cout.precision(20);
```

Функции позволяют влиять на способ представления данных в потоке вывода. В частности, функция `precision(n)` позволяет изменить количество цифр, отображаемых после десятичной точки. По умолчанию `n` равно 6. Вызов функции `precision` влияет на все операции ввода-вывода с вещественными значениями до следующего обращения к `precision`. Следует обратить внимание, что происходит округление, а не отбрасывание дробной части.

Пример 11. Используя разложение функции `sin(x)` в ряд $\sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{(2i+1)!}$, построить таблицу значений функции на отрезке [`a`, `b`] с шагом `h`.

```
#include <iostream>
#include <iomanip>
#include <cmath>
```

```
using namespace std;
double m_sin(double x);
int main() {
  double a,b,h;
  cout<<”input a, b, h”<<endl;
  cin>>a>>b>>h;
  double bg=b+h/3;
  for(double x=a; x<bg; x+=h)
    cout<<setw(6)<<x<<setw(10)<<setprecision(3)<<m_sin(x)
    <<setw(10)<<setprecision(3)<<sin(x)<<endl;
  return 0;
}

double m_sin(double x) {
  const double eps=1.0e-10;
  //while (abs(x)>2*M_PI) x = (x>0) ? x-M_PI : x+M_PI;
  double a=x,s=a,p=x*x;
  double i=0;
  while (abs(a)>eps) {
    i++;
    a=-a*p/(2*i)/(2*i+1);
    s+=a;
  }
  return s;
}
```

Для формирования таблицы значений используется оператор цикла `for` с параметром вещественного типа:

```
for(double x=a; x<bg; x+=h)
  out<<setw(6)<<x<<setw(10)<<setprecision(3)<<m_sin(x)
  <<setw(10)<<setprecision(3)<<sin(x)<<endl;
```

Здесь мы воспользовались манипулятором для объекта `cout`. Манипулятор `setw(6)` позволяет установить количество позиций для вывода значения переменной `x`, `setw(10)` – для вывода `m_sin(x)` и `sin(x)`. Для его использования требуется подключать заголовочный файл iomanip.

В таблице выводятся последовательно значения `x`, приближенное значение функции `sin(x)`, полученное в виде суммы бесконечного ряда, и значение функции `sin(x)`, полученное с использованием библиотеки cmath.

В объявлении константы `eps`

```
const double eps=1.0e-10;
```

использован квалификатор `const`, который указывает, что объявляемый объект не может быть модифицирован.

Обратите внимание на закомментированный оператор цикла

```
while (abs(x)>2*M_PI) x = (x>0) ? x - M_PI : x + M_PI;
```

 Прежде чем добавить этот оператор в код функции, следует провести вычислительные эксперименты, построив таблицу значений функции при больших значениях аргумента, например, 200 и более.

Этот оператор необходим для того, чтобы исключить влияние накопления ошибки округления при больших значениях модуля аргумента.

В теле цикла использован оператор присваивания, правая часть которого представляет собой условное выражение.

При записи условного выражения используется тернарная операция (операция, требующая трех операндов). Первый операнд приводится к типу `bool`. Если его значение `true`, то вычисляется второй операнд, в противном случае вычисляется третий операнд. Результатом условного выражения является результат вычисления второго или третьего операнда, в зависимости от того, какой именно из них был вычислен.

В отличие от традиционной условной конструкции `if-else`, тернарная условная операция возвращает результат.

1.8. Указатели

Указатель хранит адрес элемента данных и ему известен тип хранимых данных, т. е. способ интерпретации данных при доступе к памяти.

Объявление указателя

```
<тип> * <имя>;
```

Инициализация указателя возможна адресом существующей переменной соответствующего типа с помощью операции взятия адреса (&), значением другого указателя или адресом новой переменной, размещаемой в памяти с помощью операции new:

```
int *p, *q, *t;
int a=0;
p=&a;
t=p;
q=new int;
```

Доступ к данным через указатель осуществляется с помощью операции разыменования (*):

```
cout<<*p;
```

Если указатель содержит адрес составного объекта (структуры, экземпляра класса), то обратиться к полям и методам можно двумя способами:

- ✓ разыменовать указатель и использовать операцию "." (точка)

```
struct zap {
  int nom;
  double val;
};
zap* p = new zap;
(*p).nom=1;
```

- ✓ использовать операцию "->"

```
p->val=0.5;
```

Для обозначения «пустого» указателя в стандарте 1998 г. используется значение 0. Ранее для этих целей использовалась константа NULL.

```
p=NULL; q=0;
```

Удалить объект в динамической памяти через указатель можно с помощью операции delete:

```
delete q;
```

Для указателей определены операции адресной арифметики: +, –, ++, ––. Они увеличивают или уменьшают значение указателя на величину,

кратную размеру адресуемого элемента данных. Такие операции используются при работе с массивами, поскольку элементы массива расположены в памяти последовательно. Указатели тесно связаны с массивами. Указатели также использовались в С для передачи выходных параметров.

Рассмотрим использование указателей для организации динамических структур данных.

Пример 12. Создать список, содержащий N вводимых целых чисел. Распечатать полученный список в обратном порядке. Выполнить «сжатие» списка – из подряд стоящих одинаковых элементов оставить только один.

```
#include <iostream>
using namespace std;
struct list {
  int inf;
  list* next;
};

list* sp_create();
void print_sp(list* L);
void compress(list* L);
void erase(list*&L);

int main() {
  list* F;
  F=sp_create();
  print_sp(F);
  compress(F);
  print_sp(F);
  erase(F);
  return 0;
}

list* sp_create() {
  list* L,*p;
  int n;
  cout <<"size list->" ;
  cin >> n; L=NULL;
  for (int i=0; i<n; i++) {
    p = new list;
    cout << "list item ->";
    cin >>p->inf;
    p->next=L;
```

```
    L=p;
  }
  return L;
}

void print_sp(list* L) {
  list * p;
  p=L;
  while (p!=NULL) {
    cout << p->inf<< " ";
    p=p->next;
  }
  cout <<"\n";
}

void compress(list*L) {
  list* p, *q, *t, *d;
  p=L;
  while (p!=NULL) { q=p->next;
    while (q!= NULL && p->inf==q->inf)
      q=q->next;
    t=p->next;
    if (t!=q) {
      p->next=q;
      while (t!=q) {
        d=t;
        t=t->next;
        delete d;
      }
    }
    p=p->next;
  }
}

void erase(list*&L) {
  list* t;
  t=L;
  while(t!=NULL){
    L=t->next;
    delete t;
    t=L;
  }
}
```

В данном примере используется рекурсивно определенная структура данных, одно из полей которой является указателем на следующий элемент списка, если он существует.

```
struct list {
  int inf;
  list* next;
};
```

Поскольку в условии указано, что предполагается печать списка в обратном порядке, в функции создания списка `sp_create()` используется алгоритм добавления новых элементов в голову списка. Эта функция возвращает в качестве результата указатель на начало списка.

Функции печати и сжатия списка получают в качестве параметра указатель на голову списка.

Для размещения элементов списка в динамической памяти используется операция `new`, для удаления – операция `delete`.

В функцию удаления списка передается указатель на голову списка по ссылке, чтобы обеспечить изменение фактического параметра после выполнения функции. Спецификация формального параметра `list*&L` выглядит достаточно экзотично. Чтобы не прибегать к такой конструкции, можно описать тип указателя на список и изменить заголовки функций:

```
typedef list* sp_ptr;
sp_ptr sp_create();
void print_sp(sp_ptr L);
void compress(sp_ptr L);
void erase(sp_ptr &L);
```

1.9. Выражения и операции

Выражения делятся на именующие (`lvalue`) и значащие (`rvalue`).

> 🕮 *Именующее выражение* – это выражение, результатом которого является ссылка на объект.

Например, именующими являются имя переменной, массив, ссылка на элемент массива по индексу, разыменованный указатель. Именующее

выражение всегда связано с некоторой областью памяти, адрес которой известен.

📖 *Значащее выражение* – это выражение, не являющееся именующим.

В операциях присваивания левым операндом должно быть именующее выражение. Операции взятия адреса &, операции инкремента и декремента (++, - -) должны применяться к именующим выражениям. Во всех остальных операциях требуется использовать значащее выражение. В случае использования в них именующего выражения, оно будет преобразовано в значащее выражение.

Результатами работы встроенных операций индексации [ ], разыменования *, присваивания = += –= и т. д., инкремента и декремента (++, - -) являются именующие выражения. Все прочие встроенные операции производят значащие выражения.

Результатом приведения к ссылочному типу является именующее выражение, все прочие приведения имеют своим результатом значащие выражения.

Вызов функции, которая возвращает ссылку, – именующее выражение, в противном случае результат вызова функции – это значащее выражение.

Если необходимо, именующее выражение неявно преобразуется в значащее, но не наоборот.

На самом основном уровне выполнение программы на C++ сводится к последовательному вычислению выражений, проводимому под управлением инструкций (операторов). При этом некоторые выражения могут иметь один или несколько побочных эффектов. Иногда выражения включаются в программу только ради их побочных эффектов.

В языке C++ присутствуют общепринятые унарные операции, бинарные операции и одна тернарная операция. Доступ к элементам массива также производится с помощью операции [ ], а вызов функции – это n-арная операция. Это важно понимать, поскольку в C++ допустимо *для пользовательских классов* переопределять *почти все операции*, за редким исключением. Такое переопределение называется *перегрузкой операции*.

При разборе выражений следует знать приоритет и ассоциативность используемых операторов. Ассоциативность определяет принцип группировки в выражении нескольких одноприоритетных операций.

Рассмотрим только некоторые операции и выражения, отмечая их специфические особенности.

Операции *инкремента* и *декремента* имеют две формы – префиксную и постфиксную:

`++a` – префиксная форма;

`a++` – постфиксная форма.

Как было указано выше, операции инкремента и декремента в любой форме могут применяться только к именующим выражениям. Побочным эффектом выражения с такой операцией является увеличение или уменьшение именующего выражения. Результатом вычисления выражения, использующего префиксную форму, является значение именующего выражения *после его изменения*. Результатом вычисления выражения, использующего постфиксную форму, является значение именующего выражения *перед его изменением*.

В качестве примера рассмотрим два случая:

```
a=5;        a=5;
b=a++;      b=++a;
```

В первом случае после выполнения двух операторов `a=6` и `b=5`. Во втором же случае – `a=6` и `b=6`.

Операция *присваивания* используется в выражениях присваивания. В выражении присваивания значение правого операнда присваивается левому. Левый операнд выражения присваивания должен быть именующим выражением, которое можно изменять. Результат выражения присваивания также будет именующим выражением – это будет левый операнд, после того, как присваивание завершится. Это означает, что результат выражения присваивания может быть использован в другом выражении, в частности, в другом выражении присваивания:

```
a=b=0;
```

Операция присваивания имеет предпоследний приоритет, более низкий приоритет только у операции запятая. Операция присваивания право-ассоциативна (несколько подряд записанных выражений присваивания вычисляются справа налево). Помимо обычного присваивания, существует несколько других операций присваивания, каждая из которых является сокращением для операции и присваивания. Например +=, –=, *= и т. п.

Операция *запятая* позволяет использовать два выражения там, где синтаксис C++ допускает только одно выражение. Например, оператор цикла `for` предполагает использование трех выражений в заголовке – выражения инициализации, выражения условия продолжения и выражения итерации. Если при повторении цикла необходимо изменять значения сразу двух переменных, в выражении итерации можно использовать операцию запятая:

```
for (k=j=0; k<n; k++, j=k*2)
```

Иногда удобно использовать операцию запятая и в выражении инициализации.

Операция запятая гарантирует *сериализацию* вычисления выражения. Это означает, что левый операнд будет вычислен раньше правого. Поэтому результат вычисления левого операнда может быть использован при вы-

числении правого. Результатом выражения с операцией запятая является значение правого операнда. Операция запятая имеет самый низкий приоритет.

Для работы с динамической памятью в C++ используются операции `new` и `delete`. Использование операций `new` и `delete` уже было продемонстрировано в п. 1.8.

Операция выделения памяти `new` сначала резервирует необходимое для размещения объекта или массива объектов количество динамической памяти, а затем конструирует в выделенной памяти объект (выполняет его инициализацию). Результатом выражения с операцией `new` является указатель на выделенную память. Если выделить необходимое количество памяти не удалось, результатом будет или нулевой указатель, или генерация исключительной ситуации.

Операция `delete` служит для уничтожения динамического объекта или массива объектов и освобождения памяти. Значением выражения `delete` является `void`.

Рассмотрим еще один пример, где используется динамическая структура данных – дерево.

Пример 13. Создать и распечатать сбалансированное дерево, содержащее `n` целых чисел.

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

struct elem{
  int inf;
  elem* lt,*rt;
};
typedef elem* t_ptr;
t_ptr create(int n);
void erase(t_ptr t);
void printLKR(t_ptr t);
```

```
void printTREE(t_ptr t,int n);

int main(){
  int n, k=0;
  t_ptr tree;
  cout<<"Number of elements? (Cardinality) ";
  cin>>n;
  tree=create(n);
  cout<<endl<<"Infix bypass"<<endl;
  printLKR(tree);
  cout<<endl<<"Print Tree"<<endl;
  printTREE(tree,0);
  erase(tree);
  return 0;
}

void printTREE(t_ptr t, int n){
  if (t!=NULL){
    printTREE(t->rt,n+1);
    for (int i=0; i<n; i++)
      cout<<"  ";
    cout<<t->inf<<endl;
    printTREE(t->lt,n+1);
  }
}
t_ptr create(int n) {
  t_ptr p;
  int d;
  if (n>0) {
    p= new elem;
    cout<<"Another element?";
    cin>> p->inf;
    d= n/2;
    p->lt=create(d);
    p->rt=create(n-1-d);
    return p;
  }
  else
    return NULL;
}

void printLKR(t_ptr t) {
  if (t!=NULL) {
    printLKR(t->lt);
    cout<<t->inf<<" ";
    printLKR(t->rt);
  }
}
```

```
void erase(t_ptr t) {
  if (t!=NULL) {
    erase(t->lt);
    erase(t->rt);
    delete(t);
  }
}
```

В приведенной выше программе показано также использование рекурсивных функций.

На рис. 2 приведены результаты двух выводов дерева: при инфиксном обходе слева направо и в виде «перевернутого» дерева.

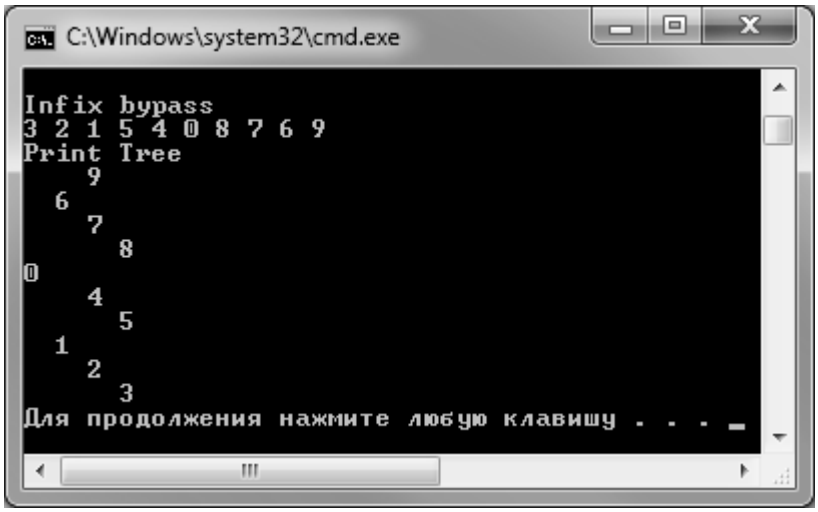


Рис. 2. Вывод сбалансированного дерева

1.10. Операторы (управляющие инструкции)

Кратко отметим особенности операторов языка C++: оператора выражения, оператора объявления, операторов цикла, операторов выбора, операторов перехода, оператора блока (составного оператора) и пустого оператора. Синтаксические правила для операторов применимы рекурсивно, поэтому везде, где правилами синтаксиса требуется оператор, может быть использован практически любой оператор.

По правилам синтаксиса любой оператор, кроме составного (блока), должен завершаться точкой с запятой.

Пустой оператор вводится для того, чтобы иметь возможность опускать при необходимости оператор там, где нет никакого действия. Пустой оператор должен завершаться точкой с запятой. Например, иногда удобно использовать оператор цикла, в котором отсутствует тело цикла:

```
for (k=s=0; k<n; k++, s+=1.0/k);
```

Оператор «выражение» представляет собой любое выражение, за которым следует точка с запятой. Поскольку результат оператора выражения не используется, этот оператор включается в программу ради его побочных эффектов. Некоторые примеры приведены в табл. 2.

Таблица 2

Примеры побочных эффектов

Оператор выражения	Побочный эффект оператора выражения
`b++;`	Увеличение значения переменной `b`
`a=0.5;`	Присваивание значения переменной `a`
`strcpy(s1,s2);`	Копирование одной строки в другую
`delete s;`	Освобождение памяти, занимаемой динамическим объектом
`a+b;`	Оператор выражения не имеет побочного эффекта

В языке C++ существуют три вида операторов цикла: `for`, `while` и `do while`.

Общий вид оператора цикла `for`:

```
for (инициализация; условие; итерации) оператор
```

Все три части, входящие в заголовок цикла (инициализация, условие и итерация), являются выражениями. Выражение итерации выполняется перед каждым повторением цикла, кроме первого. Условие проверяется перед каждым повторением цикла. Выражение инициализации выполняется перед первым выполнением оператора цикла. Важно, что все три выра-

жения в заголовке могут отсутствовать. При этом отсутствующее выражение условия считается всегда истинным.

Например, бесконечный цикл можно записать следующим образом:

```
for (;;) оператор
```

Циклы `while` и `do while` отличаются следующим: первый оператор проверяет условие перед каждой итерацией, второй – перед всеми, кроме первой:

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

> ➢ В языке C++ (C), в отличие от языка Паскаль, обе формы операторов цикла используют условие для продолжения повторений.

Ветвление в программе организуется операторами `if else` и `switch`:

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

```
switch (целочисленное выражение)
{
   case конст-выр1 : операторы
   case конст-выр2 : операторы
 . . .
   default     : операторы
}
```

Оператор `switch` существенно отличается от аналогичного оператора в языке Паскаль.

> ➢ В языке C++ каждое `конст-выр`N действует только как метка строки, а не как разграничитель вариантов.

Метка определяет, куда будет осуществлен переход при конкретном значении целочисленного выражения. После этого перехода будут по порядку выполняться все операторы, следующие за этой строкой, если только явно не указано иное. Чтобы воспрепятствовать выполнению следующего варианта, необходимо использовать оператор `break`.

Если выяснилось, что ни одна из констант не подходит, при наличии ветви, помеченной словом `default`, выполняется помеченная им ветвь. Ветвь, помеченная `default`, может отсутствовать. Чаще всего оператор `switch` используется для организации простых меню и проверок, в которых нужно разделить большое количество вариантов обработки.

Пример 14. Ввести последовательность символов, завершающуюся символом $. Посчитать, сколько раз среди вводимых символов встретились цифры 0, 1, 3, 7.

```
#include <iostream>
using namespace std;
int main() {
  char ch;
  int c0,c1,c3,c7, all;
  c0=c1=c3=c7=all=0;
  cout<<"Enter the string of characters, terminated by $";
  cin.get(ch);
  while (ch!='$') {
    all++;
    switch (ch) {
      case '0': c0++; break;
      case '1': c1++; break;
      case '3': c3++; break;
      case '7': c7++; break;
      default: break;
    }
    cin.get(ch);
  }
  cout<<"all characters-"<<all<<"\n";
  cout<<"digit 0-"<<c0<<" digit 1-"<<c1;
  cout<<" digit 3-"<<c3<<" digit 7-"<<c7<<"\n";
  return 0;
}
```

Функция, определенная в классе `istream` библиотеки <iostream>,

```
char istream::get();
```

извлекает из входного потока один символ и возвращает извлеченный символ.

В результате изучения модуля студент должен уметь:

- реализовывать алгоритмы решения несложных вычислительных задач на языке C++, с оформлением одной или нескольких функций;
- выделять при решении сложных задач логически связные функции и оформлять их в виде отдельных файлов;
- писать заголовочные файлы для подключения своих функций.

В результате изучения модуля студент должен знать:

- как описываются и используются переменные базовых типов;
- какие операции могут быть использованы при построении выражений;
- чем отличаются выражение и оператор «выражение»;
- как использовать управляющие инструкции (операции) при организации программ;
- как описывать и использовать динамические структуры данных.

Вопросы для рубежного контроля

1) Для чего нужна директива препроцессора:
   `#include <iostream>`
2) Для чего в программе на C++ нужна функция `main()`?
3) Чем различаются объявление функции и ее определение?
4) Как в C++ специфицируется передача параметра по ссылке?
5) Какие функции называются перегруженными?
6) Могут ли описания функций на C++ быть вложены друг в друга?
7) Для чего используются объекты `cin` и `cout`?
8) Перечислить все целочисленные типы данных.

9) Какие особенности записи выражений следуют из полиморфизма операции деления?

10) Для чего используются поразрядные логические операции?

11) Каково назначение каждого из выражений в заголовке оператора `for`?

12) В чем заключается различие между именующими и значащими выражениями?

13) Для чего нужна операция запятая? Что является ее результатом?

14) Каково назначение тернарной операции?

15) Описать использование операций `new` и `delete`.

16) Как используется указатель для доступа к полям структуры?

17) В чем заключается различие между циклом `while` и `do while`?

Задания для самостоятельной работы

1. Найти ошибки в каждом из следующих фрагментов программы и объяснить, как их можно исправить. Для проверки внесенных исправлений необходимо написать программу, вставить в нее фрагменты и откомпилировать.

a)
```
int g() {
  cout << "внутри функции g" << endl;
  int h() {
    cout << "внутри функции f" << endl;
  }
}
```

b)
```
int sum(int x, y) {
  int result;
  result=x+y;
}
```

c)
```
//рекурсивное вычисление суммы всех целых чисел от 1 до n
  int sum(int n) {
    if (n==0)
        return 0;
    else
        n=sum(n-1);
  }
```

```
d)  void f(float a);
     {
     float a;
     cout << a << endl;
    }
e)  void product() {
     int a,b,c,result;
     cout << "Input a,b,c: ";
     cin >> a >> b >> c >>;
     result=a*b*c;
     cout >> "Result = " >> Result;
     return result;
    }
f)  double max(double a,b) {
     if a>b
       return a
     else
       return b;
    }
```

2. Найти логические ошибки в каждом из следующих фрагментов программы и исправить их. Для выполнения задания необходимо написать программу и вставить в нее фрагменты. Для поиска ошибок можно воспользоваться отладчиком.

```
a)  int sum(int a, int b) {
     for(int i=a, int s=0;i<=b;i++)
       s+=i;
     return s;
    }
b)  int fact(int n) {
     int f;
     for (int i=1; i<n; i++)
       f*=i;
     return f;
    }
c)  void swap(int a,int b) {
     int c=a;
     a=b;
     b=c;
    }
d)  int max(int a, int b, int c) {
     int maxi;
     if (a>maxi)
       maxi=a;
     else if (b>maxi)
       maxi=b;
```

```
  else if (c>maxi)
     maxi=c;
  return maxi;
}
```

Проектные задания к модулю

1. Написать функцию, вычисляющую площадь треугольника по заданным длинам его сторон. Создать проект для решения треугольников по трем сторонам.

2. Написать функцию определения расстояния между двумя точками, заданными декартовыми координатами на плоскости. Создать проект для отладки данной функции.

3. Написать функцию, определяющую наибольший общий делитель двух целых чисел с использованием алгоритма Евклида. Написать функцию, преобразующую обыкновенную дробь, заданную в виде числителя и знаменателя, к несократимому виду. Написать функцию, печатающую обыкновенную дробь. Создать проект, в котором будут использованы все вышеперечисленные функции.

4. Описать функцию `Factors(R)`, находящую разложение натурального числа `R` на простые множители. Функция должна вернуть количество множителей, а сами множители выдать на экран в порядке неубывания. С помощью этой функции разложить на простые множители пять данных чисел.

5. Расширить список функций, работающих с цифрами десятичного представления, добавив, например, функцию замены цифры в числе, функцию перестановки цифр и др. Внести соответствующие изменения в файлы digit.cpp и digit.h. Создать новый проект, в котором в функции `main()` будут использованы новые функции. Включить в этот проект уже готовые файлы digit.cpp и digit.h.

6. Составить набор функций (b_ digit.cpp) для работы с цифрами двоичного представления целого числа. Создать проект, в котором используются функции для работы с цифрами десятичного (digit.cpp) и двоичного (b_ digit.cpp) представления целого числа.

Для вывода двоичного представления целого неотрицательного числа можно использовать рекурсивную функцию, выводящую число без последней цифры, а затем – последнюю цифру:

```
void print_b(int a) {
  if (a) {
    print_b(a/2);
    cout<<a%2;
  }
}
```

7. Написать набор функций (roots.cpp) для вычисления квадратного, кубического корня и арифметического корня степени k ($y = \sqrt[k]{x}$), используя итерационную формулу Ньютона:

$$\begin{cases} y_{n+1} = \dfrac{1}{k}\left[(k-1)y_n + \dfrac{x}{y_n^{k-1}}\right] \\ y_0 = x \end{cases}.$$

Итерационный процесс вычисления заканчивается, когда два последовательных приближения будут удовлетворять условию

$$\left|y_{n+1} - y_n\right| < \varepsilon .$$

Использовать написанные функции в проекте, который выдает таблицу значений арифметических корней для заданной последовательности целых положительных чисел.

8. Написать набор функций (elem.cpp) для вычисления элементарных функций *sin x*, *cos x*, e^x, используя формулы Маклорена. Составить проект, в котором выполняется табулирование указанных функций на про-

извольном отрезке с заданным шагом. Сравнить получаемые значения со значениями, вычисляемыми функциями из библиотеки cmath.

9. Написать набор функций (q_digit.cpp) для проверки, обладает ли десятичная запись целого положительного числа заданными свойствами. Например, в десятичной записи числа нет нулей, десятичная запись числа состоит точно из k цифр, в десятичной записи числа встречается заданная цифра, сумма цифр равна заданному значению и др. Составить программу, в которой вводимая последовательность целых положительных чисел проверяется на наличие/отсутствие указанных свойств.

10. Написать набор функций (compl.cpp) для выполнения операций с комплексными числами, заданными в тригонометрической форме. Составить программу, вычисляющую значение любого выражения, операндами которого являются комплексные числа.

МОДУЛЬ 2. СТРУКТУРЫ ДАННЫХ И ФУНКЦИИ

Задачи и цели: Рассмотреть возможности описания и использования массивов в программах на C++. Познакомиться с тем, как связаны массивы, указатели и адресная арифметика. Изучить способы передачи параметров-массивов в функции. Научиться использовать динамические массивы. Познакомиться с двумя способами представления и обработки строк в C++. Рассмотреть особенности описания и использования функций в языке C++. Изучить новые возможности использования функций по сравнению с языком C.

Требуемый начальный уровень подготовки. Умение выполнять реализацию алгоритмов решения несложных вычислительных задач на языке C++, с оформлением одной или нескольких функций. Умение выделять при решении сложных задач логически связные функции и оформлять их в виде отдельных файлов. Умение составлять заголовочные файлы для подключения собственных функций.

2.1. Одномерные массивы

Оператор объявления массива должен определять тип элементов в массиве, имя массива и количество элементов в массиве:

```
<тип> <имя>[<размерность>];
```

Размер массива фиксирован на все время существования в памяти и не может быть изменен.

Доступ к элементам массива производится с помощью операции индексации [ ].

Пример 15. Дан целочисленный массив, содержащий не более 10 элементов. Найти номер первого нулевого элемента в массиве.

```
#include <iostream>
using namespace std;
int check_null(int a[], int n);
int main() {
  int a[10], n,k;
  do {
    cout << " array size " << endl;
    cin>> n;
  }
  while (n<1 ||n>10);
  for(int i=0; i<n; i++) {
    cout<<i<<" array element ";
    cin>>a[i];
  }
  if ( (k=check_null(a,n))<0)
    cout<<" no zeros \n";
  else
    cout<<" number of the first zero = "<<k;
  return 0;
}
int check_null(int a[], int n) {
  int i=0;
  while (i<n && a[i]!=0)
    i++;
  return i<n ? i : -1;
}
```

Необходимо обратить внимание на потенциальный источник ошибок при работе с массивами.

☠ C++ не проверяет границ массивов, чтобы предупредить о ссылках на несуществующие элементы.

Из приведенного примера видно, что в функции передавать массив можно следующим образом:

```
int check_null(int a[], int n)
```

📚 В C++ при передаче массивов в качестве параметров функции передается адрес первого элемента массива. Функции могут изменять значения элементов массива.

Для предотвращения модификации исходного массива внутри тела функции можно в определении функции применять к параметру-массиву спецификатор типа `const`.

Функции не должны модифицировать массивы без крайней необходимости (принцип наименьших привилегий):

```
int check_null(const int a[], int n)
```

Другой способ передачи массивов в качестве параметров в функции связан с использованием указателей:

```
int check_null(const int *a, int n)
```

 Изменить в примере 15 заголовок функции так, чтобы для передачи массива использовался указатель.

➢ При работе с массивом поэлементно следует учитывать, что нумерация элементов массива в языке C++ всегда начинается с 0. Поэтому последний элемент массива имеет индекс на единицу меньше, чем размер массива.

Пример 16. Написать программу, подсчитывающую во вводимой последовательности символов по отдельности каждую цифру, пробельные литеры (пробелы, табуляции и новые строки) и все другие литеры (кроме «*»). Признаком окончания последовательности является символ «*».

Вот один из вариантов решения:

```
#include <iostream>
using namespace std;
void main () {
  int i, nwhite, nother;
  char c;
  int ndigit [10];
  nwhite = nother = 0;
  for (i = 0; i < 10; ++i)
    ndigit [i] = 0;
  cin.get(c);
```

```
  while (c != '*') {
    if ((c >= '0') && (c <= '9'))
      ++ndigit [c-'0'];
    else if ((c ==' ') || (c == '\n') || (c == '\t'))
      ++nwhite;
    else
      ++nother;
    cin.get(c);
  }
  cout<<"digits = "<<endl;
  for (i = 0; i < 10; ++i)
      cout<<i<<'-'<<ndigit[i]<<' '<<endl;
  cout<<" white = "<<nwhite<<" other = "<<nother<<endl;
}
```

Это несколько искусственная задача, но она позволит нам в одном примере продемонстрировать еще несколько возможностей языка C++.

Имеется двенадцать категорий вводимых литер. Удобно десять счетчиков цифр хранить в массиве, а не в виде десяти отдельных переменных. При определении значения индекса в массиве-счетчиков цифр `ndigit [c-'0']` используется то обстоятельство, что тип `char` является целочисленным и не требует преобразования типов. Для двух оставшихся категорий литер используются отдельные счетчики.

2.2. Массивы в динамической памяти

Если массив создается путем объявления, то память под него будет отведена перед началом выполнения программы или функции и будет закреплена за массивом до завершения времени жизни массива. Оператор `new` позволяет выделять память под массив во время выполнения программы, причем указывая его размер не «по максимуму», а необходимый при конкретном выполнении программы.

```
int *a; int n;
//определение размера массива n, например, cin>>n;
a=new int[n];
```

Если для создания массива использовался оператор new, то для освобождения памяти необходимо применить специальную форму оператора delete, которая указывает, что освобождается память, занимаемая массивом:

```
delete[] a;
```

Пример 17. Вычислить среднее арифметическое элементов вещественного массива.

```
#include <iostream>
using namespace std;
double avg(const double *a, int n);

int main() {
  double *a;
  int n;
  cout <<" The size of the array ";
  cin>>n;
  a= new double[n];
  for (int i=0; i<n; i++) {
    cout<<i<<" element ";
    cin>>a[i];
  }
  cout<<" average ="<<avg(a,n);
  delete[] a;
  return 0;
}
double avg(const double *a, int n) {
  double sum=0;
  for(int i=0; i<n; i++)
    sum+=a[i];
  if (n) return sum/n;
    else return 0;
}
```

При передаче массива как параметра в функцию используется указатель:

```
double avg(const double *a, int n)
```

Это возможно потому, что указатель – это адрес, а имя массива является адресом его первого элемента.

2.3. Связь массивов и указателей

Неважно, как создан массив, имя массива – это адрес (указатель) первого по счету элемента массива `a[0]`. Над указателями определены: адресная арифметика, операции присваивания и сравнения. Если к указателю прибавляется число, то это означает, что указатель сдвигается вправо на такое количество байт, какое занимает указанное число элементов заданного типа.

Указатели можно сравнивать на равенство/неравенство. Если имеется два указателя на один массив, то разность этих указателей будет равна количеству элементов массива, расположенных между этими указателями.

Если `a` – указатель на массив, то, увеличив `a`, можно сместиться на следующий элемент. Выражение `a++` ссылается на элемент массива с индексом 1, так же как и выражение `&a[1]`. Но `a++` изменяет значение указателя `a` в отличие от `&a[1]`.

Важно уметь использовать арифметику с указателями в задачах обработки массивов, так как некоторые компиляторы операцию `a[i]` выполняют дольше, чем `*(a+i)`.

Пример 18. Проверить, является ли массив целых чисел симметричным относительно своей середины.

```
#include <iostream>
using namespace std;
bool simm( int *a, int n );
int main() {
  int *a, n;
  cout <<" The size of the array ";
  cin>>n;
  a=new int[n];
  for (int i=0; i<n; i++) {
    cout<<i<<" element ";
    cin>>a[i];
  }
  cout<<simm(a,n)<<endl;
```

```
  delete []a;
  return 0;
}
bool simm( int *a, int n ) {
  int *b=a+n-1;
  while (a<=b)
    if (*a++!=*b--)
      return false;
  return true;
}
```

Выражение `a+n-1` вычисляет адрес последнего элемента массива. Указатель `b` позволяет перемещаться по элементам массива справа налево, в то время как указатель `a` движется от начала массива к середине. Условие цикла `(a<=b)` определяет, что указатели `a` и `b` еще не достигли середины массива.

Традиционный способ передачи параметров-массивов в функции заключается в передаче указателя на начало массива и размера массива. Существует еще один метод предоставления необходимой информации функции – указание диапазона элементов массива. Это осуществляется путем передачи двух указателей: один определяет начало диапазона элементов массива, другой – его конец. Такой способ передачи продемонстрирован в примере 19.

Пример 19. Описать функцию, вычисляющую сумму элементов массива, продемонстрировать ее использование для разных сегментов массива.

```
#include <iostream>
using namespace std;
int sum_arr( const int *begin, const int *end);
int main() {
  int array[] = {1,2,3,4,5,6,7,8,9,10};
  int n = sizeof array / sizeof(int);
  cout<<"The sum of all ="<<sum_arr(array, array+n)<<endl;
  cout<<"The sum of the first="<<sum_arr(array,array+3)<<endl;
  cout<<"The sum of the last 4 ="
     <<sum_arr(array+n-4,array+n)<<endl;
  cout<<"The sum from 3 to 7 ="<<sum_arr(array+2,array+7) <<endl;
  return 0;
}
```

```
int sum_arr( const int *begin, const int *end) {
  const int *pt;
  int sum=0;
  for (pt=begin; pt!=end; pt++)
    sum+=*pt;
  return sum;
}
```

2.4. Статическое определение двумерных массивов

Специального типа двумерного массива в C++ не существует. Но можно определить массив, каждый элемент которого, в свою очередь, является массивом.

```
<тип> <имя>[<размерность1>][<размерность2>];
```

Например, следующее объявление массива означает, что `matr` – это массив из 4 элементов, каждый из которых является массивом из 5 целых чисел:

```
int matr[4][5];
```

Аналогичным образом можно объявлять любые многомерные массивы.

Пример 20. Применить индексную сортировку для упорядочивания строк матрицы по возрастанию сумм их элементов.

```
#include <iostream>
#include <iomanip>
using namespace std;
void ind_sort(int matr[][10], int n, int m, int ind[]);

int main() {
  int matr[10][10];
  int n, m;
  int ind[10];
  cout<<"size matrix ";
  cin>>n>>m;
  for (int i=0; i<n; i++) {
    cout<<"row "<<i<<" ";
    for(int j=0; j<m; j++) {
      cin>>matr[i][j];
    }
  }
```

```
  ind_sort(matr,n,m,ind);
  for (int i=0; i<n; i++) {
    cout<<endl;
    for(int j=0; j<m; j++) {
      cout<<setw(4)<<matr[ind[i]][j];
    }
  }
  cout<<endl;
  return 0;
}

void ind_sort(int a[][10],int n,int m, int ind[]) {
  int sum[10];
  for (int i=0; i<n; i++) {
  ind[i]=i;
    sum[i]=0;
    for(int j=0;j<m;j++)
      sum[i]+=a[i][j];
  }
  bool flag=true;
  int j=n-1;
  while (flag) {
    flag=false;
    for (int i=0; i<j; i++) {
      if (sum[ind[i]]>sum[ind[i+1]]) {
        int k=ind[i];
        ind[i]=ind[i+1];
        ind[i+1]=k;
        flag=true;
      }
    }
    j--;
  }
}
```

Если у нас есть двумерный массив или массив с большим количеством размерностей, то при передаче параметров только первую размерность можно оставить неопределенной. Остальные размерности должны быть заданы:

```
void ind_sort(int a[][10],int n,int m, int ind[]);
```

Данный способ объявления эквивалентен передаче в качестве параметра указателя на массив, каждый элемент которого является массивом из 10 элементов:

```
void ind_sort(int (*a)[10],int n,int m, int *ind);
```

В такой записи скобки необходимы, поскольку объявление `int *a[10]` означало бы массив из 10 указателей типа `int`.

В приведенном примере для форматированного вывода элементов матрицы используется манипулятор `setw`, устанавливающий ширину поля вывода:

```
for(int j=0; j<m; j++) {
  cout<<setw(4)<<matr[ind[i]][j];
}
```

2.5. Двумерные массивы в динамической памяти

Работа с двумерным динамическим массивом начинается с описания «указателя на указатель». Например, если массив будет содержать целые значения, то описание выглядит следующим образом:

```
int **a;
```

Выделение динамической памяти для двумерного массива производится в два этапа. Сначала память выделяется для указателей на соответствующие одномерные массивы (строки матрицы):

```
a=new int *[n];
```

После этого выделяется память для каждой строки:

```
for (int i=0; i<n; i++)
  a[i]=new int[m];
```

Проанализировав вышеприведенный способ размещения двумерного динамического массива, можно сделать вывод о том, что у динамического двумерного массива не обязательно все строки должны быть одинаковой длины.

Освобождение памяти, занятой динамическим массивом, нужно проводить в обратной последовательности:

```
for (i=0; i<n; i++)
   delete []a[i];
delete []a;
```

Пример 21. Создать и распечатать верхнетреугольную матрицу следующего вида:

$$\begin{pmatrix} n & n & \dots & n & n \\ 0 & n-1 & \dots & n-1 & n-1 \\ & & \dots & & \\ 0 & 0 & \dots & 2 & 2 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}.$$

Применить экономное выделение памяти только для элементов на главной диагонали и выше.

```
#include <iostream>
using namespace std;
int main(){
  int **matr;
  int n,m,i,j;
  cout<<" size matrix ";
  cin>>n;
  matr = new int *[n];
  for ( i=0; i<n; i++) {
    m=n-i;
    matr[i]= new int[m];
    for (j=0; j<m; j++)
      matr[i][j]=m;
  }
  for (i=0; i<n; i++) {
    for (j=0; j<i; j++)
      cout<<"0 ";
    m=n-i;
    for (j=0; j<m; j++)
      cout<<matr[i][j]<<" ";
    cout<<endl;
  }
  for (i=0; i<n; i++)
    delete []matr[i];
  delete[] matr;
  return 0;
}
```

Особенность данной задачи состоит в том, что в памяти формируется нерегулярный массив. Каждая следующая строка короче предыдущей на один элемент.

```
  matr = new int *[n];
  for ( i=0; i<n; i++) {
    m=n-i;
    matr[i]= new int[m];
    for (j=0; j<m; j++)
      matr[i][j]=m;
  }
```

 Создать и распечатать нижнетреугольную матрицу, заполнив ее аналогично матрице из примера 21.

Пример 22. Упорядочить строки матрицы по возрастанию их первых элементов.

```
#include <iostream>
#include <iomanip>
using namespace std;
void sort(int **matr, int n, int m);
int main() {
  int **matr, n, m;
  cout<<"size matrix ";
  cin>>n>>m;
  matr = new int *[n];
  for (int i=0; i<n; i++)
    matr[i]= new int[m];
  for (int i=0; i<n; i++) {
    cout<<"row "<<i<<" ";
    for(int j=0; j<m; j++)
      cin>>matr[i][j];
  }
  sort(matr,n,m);
  for (int i=0; i<n; i++) {
    cout<<endl;
    for(int j=0; j<m; j++) {
      cout<<setw(4)<<matr[i][j];
    }
  }
  cout<<endl;
  for (int i=0; i<n; i++)
    delete []matr[i];
  delete[] matr;
  return 0;
}
void sort(int **a,int n,int m) {
  bool flag=true;
  int j=n-1;
```

```
  while (flag) {
    flag=false;
    for (int i=0; i<j; i++) {
      if (a[i][0]>a[i+1][0]) {
        int * k=a[i];
        a[i]=a[i+1];
        a[i+1]=k;
        flag=true;
      }
    }
    j--;
  }
}
```

Отметим, что при решении задачи используется особенность представления двумерного массива как массива указателей на одномерные массивы. В процессе сортировки местами меняются не строки, а указатели на них.

```
      if (a[i][0]>a[i+1][0]) {
        int * k=a[i];
        a[i]=a[i+1];
        a[i+1]=k;
        flag=true;
      }
```

Это значительно ускоряет время работы программы для матриц с большим количеством столбцов.

2.6. Описание и инициализация строк

В языке C++ работать со строками можно двумя способами: в стиле языка C и в стиле языка C++. Между строками в C++ и C существуют заметные различия.

📖 Строка в стиле языка C – массив символов, последним элементом которого всегда является двоичный ноль – '\0' (часто называемый *null-символом* или *null-терминатором*). Следовательно, и основные принципы работы с ними такие же, как и с массивами.

К строковым константам нулевой символ добавляется автоматически.

Длина строки на единицу больше количества символов в ней, потому что при определении длины учитывается и нулевой символ. Так, например, в результате выполнения оператора

```
cout<<sizeof("C++");
```

на экране появится число 4.

Рассмотрим пример описания и инициализации строки в стиле C:

```
char s1[20] = "String";
```

Следующая строка демонстрирует типичную ошибку:

```
char s2[6] = "String"; // Не отведено место под завершающий '\0'
```

☠ Если не отведено место под завершающий null-символ, то конец строки не определен.

Чтобы избежать этой ошибки, рекомендуется использовать следующие варианты инициализации:

```
const char s3[] = "String";
const char* s4 = "String";
```

Под массив `s3` при этом отводится память из семи элементов типа `char`. Заметим, что в отличие от `s3`, указатель `s4` может менять свое значение в процессе работы.

Также можно встретить аналогичную инициализацию без `const`:

```
char s3[] = "String";
char* s4 = "String";
```

Такая инициализация не вполне корректна, однако ее можно встретить в реальных программах. В первом случае объявляется константный указатель на массив `s3`. Значения элементов массива можно изменять, а значение указателя – нет. Во втором случае объявленный указатель `s4` ссылается на константную строку. Значение указателя изменять можно, а значения элементов массива – нет. Эти особенности продемонстрированы в следующей программе:

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

int main() {
  char s3[] = "String";
  char* s4 = "String";
  s3[1]='A';
  cout<<s3<<endl;

  //s4[1]='A'; - ошибка времени выполнения
  //cout<<s4<<endl;
  //cout<<*(++s3)<<endl; - ошибка компиляции

  cout<<*(++s4)<<endl;
  return 0;
}
```

Создать строку, так же как и массив, можно динамически.

```
char* s5 = new char[10];
```

где `s5` – указатель на строку, память для которой выделена динамически. При таком описании строка неинициализирована.

Распространенная ошибка при попытке описания строки:

```
char* s6;//это не строка, а указатель на строку
```

В действительности переменная `s6` является указателем на `char` или на строку (массив символов).

☠ Если память для строки не выделена, то переменную нельзя использовать в качестве строки.

Доступ к строке обычно осуществляется с помощью указателя `char*`. Поэтому, как правило, тип `char*` ассоциируется именно со строкой. Так, если переменная `s` имеет тип `char*`, то при выполнении оператора

```
cout<<s;
```

в поток вывода записываются символы, на которые указывает `s`, до тех пор, пока не будет встречен нулевой символ '\0'.

Оператор >> позволяет ввести слово:

```
char word[10];
cin >> word;
```

При этом в потоке ввода пропускаются все начальные символы-разделители (пробелы, символы перехода на новую строку и символы табуляции), затем в переменную `word` считываются символы до символа разделителя и дописывается нулевой символ.

Рассмотрим распространенные ошибки при вводе строковых данных.

☠ **Ошибка 1**. Попытка ввести больше символов, чем вмещает в себя массив символов `word`.

Например, при вводе строки `"␣␣abracadabra␣␣"` (пробел для наглядности изображен символом ␣) два начальных пробела будут пропущены, в массив `word` будут записаны символы `"abracadabr"`, а символы `'a'` и `'\0'` будут записаны в следующие за `word` ячейки памяти. Сообщение об ошибке при этом, как правило, не возникает.

Для решения проблемы выхода за границы массива-строки в приведенном примере следует использовать манипулятор `setw`, устанавливающий ширину поля ввода:

```
cin>>setw(10)>>word;
```

В этом случае в массив `word` попадут символы `"abracadab"`плюс завершающий нулевой символ, а символы `'ra'` останутся в потоке ввода.

Для использования `setw` необходимо подключить заголовочный файл iomanip. Использование манипулятора влияет только на непосредственно следующую за ней операцию ввода.

☠ **Ошибка 2**. Попытка ввести строку, содержащую пробельные символы.

Описанным выше способом невозможно ввести строку, содержащую пробелы или другие пробельные символы. Если требуется ввести не отдельное слово, а строку целиком, то следует использовать функцию-член `getline` объекта `cin` класса `istream`:

```
cin.getline(word,10);
```

Эта функция будет осуществлять ввод до символа перехода на новую строку `'\n'`, но максимально будет считано 9 символов, после чего к строке будет добавлен `'\0'`.

Возможен вариант функции с тремя параметрами, где третий параметр определяет, какой символ является признаком завершения ввода.

```
cin.getline(word,10,'!');
```

В этом случае считывание осуществляется до символа `'!'`, но не более 9 символов.

> ➢ Операция конкатенации для строк в стиле С не определена. Поскольку строка в стиле С – это массив символов, то копирование строк невозможно осуществить с помощью операции присваивания.

Для выполнения операций над строками используются библиотечные функции. Для строк в стиле С они собраны в библиотеке с заголовочным файлом cstring.

Для строк в стиле С++ используется класс `string` (заголовочный файл string). Рассмотрим примеры работы с классом `string`:

```
#include <string>
#include <iostream>
using namespace std;
int main() {
  string s1,s2;  //Пустые строки
  string s3 = "Hello!";  //Инициализированная строка
  string s4("I am");  //Еще один пример инициализации
  string s5(s3);  //И еще один пример инициализации
  s2 = "By";  //Присваивание
```

```
  s1 =s3 + " " + s4;  //Слияние строк - конкатенация
  s1 += " Good ";  //Присоединение новых символов к строке
  cout  << s1 + s2 + "!" << endl;
}
```

Строки `s1` и `s2` создаются пустыми, а строки:

```
string s3 = "Hello!";
string s4("I am");
```

иллюстрируют два эквивалентных способа инициализации объектов `string` символьными массивами. Объект `s5` инициализируется уже существующим объектом `s3`.

Любому объекту `string` можно присвоить новое значение оператором присваивания. Например,

```
s2 = "By";
```

Прежнее содержимое строки замещается данными, находящимися в правой части оператора присваивания, и программисту не нужно беспокоиться об удалении старых данных, об освобождении или выделении памяти – это происходит автоматически. Объекты `string` можно конкатенировать (соединять) с помощью операций + и +=.

Потоки ввода-вывода умеют работать с объектами `string`.

➢ Следует обратить внимание на то, что:

✓ операция конкатенации определена только для класса `string`, а для строк в стиле С не определена;

✓ операция присваивания для строк в стиле С, в отличие от присваивания для класса `string`, присваивает только указатели, а не сами строки.

Стандартный класс C++ `string` скрывает способ хранения строки. Объект `string` содержит служебную информацию: начальный адрес в памяти, саму строку, длину хранимой строки в символах и максимальную длину, до которой строка может увеличиться без повторного выделения памяти. При необходимости это выделение выполняется автоматически.

Строки в стиле C++ существенно снижают вероятность самых распространенных и опасных ошибок программирования в стиле языке C: выхода за границы массива, попытки обращения к массиву через неинициализированный или ошибочный указатель, появление «висячих» указателей после освобождения блока памяти, в котором ранее хранился массив.

 В языке C каждый символьный массив всегда занимает уникальный физический блок памяти. В C++ отдельные объекты `string` могут занимать или не занимать уникальные физические блоки памяти. Если два объекта `string` хранят строку с одним и тем же значением, они могут ссылаться на один и тот же физический блок памяти до тех пор, пока один из объектов не начнет изменяться. Тогда для него будет выделен новый блок памяти. Такая возможность реализуется посредством подсчета ссылок на блок памяти, в котором расположена изменяющаяся строка. С точки зрения программиста, отдельные объекты `string` должны работать так, словно каждый из них хранится в отдельном блоке.

Чтобы использовать в программе объекты `string`, необходимо включить в нее заголовочный файл C++ string. Класс `string` определен в пространстве имен `std`, поэтому в программу включается директива `using`.

2.7. Обработка строк в стиле языка C

Поскольку строка в стиле C завершается нулевым символом, ее обработка имеет определенную специфику.

Пример 23. Реализовать функцию копирования заданной строки `s2` в строку `s1`.

Рассмотрим вначале несколько способов копирования без привлечения библиотечных функций.

Следующий цикл производит посимвольное копирование из строки s1 в строку s2 до того момента, как в строке встретится нулевой символ.

```
int i;
for (i=0; s1[i]; i++)
  s2[i] = s1[i];
s2[i] = 0;
```

После цикла нулевой символ дописывается в конец строки s2 (число 0 неявно преобразуется в символ '\0').

Посимвольное копирование можно также осуществить, используя указатели:

```
char *p=s1, *q=s2;
while (*p)
  *q++=*p++;
*q=0;
```

Если *p становится равным нулю, значит достигнут конец строки s1, и цикл завершается. Завершающий нулевой символ также приходится дописывать «вручную». Следует обратить внимание на то, что после цикла длина строки может быть вычислена как p-s1.

Наконец, учитывая то, что оператор присваивания возвращает значение левой части после присваивания, алгоритм копирования можно записать максимально компактно:

```
char *p=s1, *q=s2;
while (*q++ = *p++);
```

На последней итерации цикла *p становится равным нулю, это значение присваивается *q и возвращается как результат операции присваивания, что и приводит к завершению цикла.

Оформим последний вариант реализации в виде функции copy и приведем полный текст программы с использованием этой функции.

```
#include <iostream>
using namespace std;
char * copy(char *p, const char *q) {
  while (*p++=*q++);
```

```
  return p;
}

int main(){
  char s1[20], s2[10]="Hello!";
  char *s3=new char [21];
  copy(s1,s2);
  cout <<"s1=" << s1 << endl;
  cout <<"s2=" << s2 << endl;
  copy(s3,s1);
  cout <<"s3=" << s3 << endl;
}
```

Все три рассмотренных варианта являются возможными реализациями стандартной функции `strcpy`. Она, как и остальные стандартные функции для работы со строками в стиле языка С, объявлена в заголовочном файле string.h (или, по стандарту 1998 г., в cstring).

Назовем наиболее часто используемые стандартные функции для работы со строками в стиле языка С: `strlen`, `strcpy`, `strcmp`, `strcat`.

☠ Распространенная ошибка: применение в качестве параметров для данных функций неинициализированных указателей на строки. В этом случае возникает не всегда отлавливаемая ошибочная ситуация (ошибка при работе с памятью).

```
int main(){
  char s2[10]="Hello!";
  char *s3;
  strcpy(s3,s2);    //ошибочное использование s3
                    //память для строки s3 не выделена
  cout <<"s2=" << s2 << endl;
  cout <<"s3=" << s3 << endl;
}
```

Пример 24. Дана строка, состоящая из слов, разделенных одним или несколькими пробелами. В начале и в конце строки могут находиться начальные и конечные пробелы. Требуется удалить лишние пробелы (оставить только между словами и только по одному), поменять местами четные и нечетные слова и записать результат в новую строку.

Рассмотрим *полуторапроходной алгоритм* решения данной задачи. Он не требует дополнительной структуры данных (массив, очередь, стек), в нем запоминается адрес лишь одного слова.

В данном алгоритме будут несколько раз встречаться следующие циклы:

```
while (*p == ' ') p++;   // пропуск пробелов
while (*p != ' ' && *p) p++;  // пропуск слова
while (*p != ' ' && *p) *ps++=*p++;     // копирование слова
```

Условие (`*p != ' ' && *p`) означает «пока не пробел и не конец строки».

Пусть `p` – исходная строка, `ps` – результирующая. Рассмотрим одну из реализаций алгоритма.

```
void swap_even(const char *p, char *ps){
  char *p1;
  while (*p == ' ') p++; //пропустить начальные пробелы
  do {
    p1= const_cast <char*> (p);//запомнить адрес нечетного слова
    while (*p != ' ' && *p) p++;  //пропустить нечетное слово
    while (*p == ' ') p++;    //пропустить пробелы
    if (*p) {  //если есть четное слово
      while (*p != ' ' && *p)
        *ps++=*p++; //копировать четное слово
      *ps++ = ' ';  //добавить пробел
    }
    while (*p1 != ' ' && *p1)
      *ps++=*p1++;  //копировать нечетное слово
    *ps++ = ' ';    //добавить пробел
    while (*p == ' ') p++;    //пропустить пробелы
  } while (*p);
  *--ps=0;     //удалить лишний пробел и добавить 0
}
int main(){
  char s1[100],
  s2[100]="  Hello!   1111111111    2222222222222  ";
  char *s3=new char [100];
  swap_even(s2,s1);
  cout <<"s1=" << s1 << endl;
  cout <<"s2=" << s2 << endl;
  swap_even(s1,s3);
  cout <<"s3=" << s3 << endl;
}
```

Рассмотрим строку, описывающую присваивание указателей:

```
p1= const_cast <char*> (p);
```

Параметр `p` имеет тип `const char*`, а локальная переменная `p1` функции `swap_even` имеет тип `char*`. Поэтому используется операция явного преобразования типа в стиле C++ `const_cast <char*>(a)`.

Проверить, что функция корректно работает и в случае строки, состоящей из одних пробелов, и в случае пустой строки.

2.8. Обработка строк в стиле языка C++

В классе `string` имеется большое количество различных функций-членов для обработки строк.

Пример 25. Заменить в строке `s1` каждое вхождение подстроки `s2` подстрокой `s3`.

Такую функцию несложно написать на базе готовых функций `find()` и `replace()`. Эти функции, как и многие другие, являются член-функциями класса `string`.

```
//ReplaceAll.h

#ifndef REPLACEALL_H
#define REPLACEALL_H
#include <string>
std::string & replaceAll (string &context, const string &from,
                          const string & to);
#endif    //REPLACEALL_H

//ReplaceAll.cpp

#include "ReplaceAll.h"
using namespace std;
string & replaceAll (string & context, const string & from,
                     string & to){
  size_t lookHere=0;
  size_t foundHere;
  while ((foundHere = context.find(from, lookHere)) !=
                                          string::npos){
```

```
    context.replace(foundHere, from.size(), to);
    lookHere = foundHere + to.size();
  }
 return context;
}
```

Версия `find()`, использованная в этой программе, получает во втором аргументе начальную позицию поиска. Если вхождение подстроки не найдено, то возвращается значение `string::npos`. Поиск следующего вхождения `from` в `context` необходимо начинать с позиции, следующей за позицией произведенной замены, на тот случай, если `from` является подстрокой `to`. Поэтому значение переменной `lookHere` важно увеличить на длину заменяющей строки `to`.

Следующая программа предназначена для тестирования функции `replaceAll()`:

```
//ReplaceAllTest.cpp
#include <iostream>
#include <string>
#include "ReplaceAll.h"
using namespace std;
int main() {
  string text = "a man, a plan, a canal, panama";
  cout<<text<<endl;
  replaceAll(text, "an", "XXX");
  cout<<text<<endl;
  return 0;
}
```

Количество функций-членов класса `string` примерно соответствует количеству функций в библиотеке C для работы со строками, но механизм перегрузки существенно расширяет их функциональность.

Одно из самых ценных и удобных свойств строк C++ состоит в том, что они могут автоматически изменять размер по мере надобности, не требуя вмешательства со стороны программиста. Работа со строками становится не только более надежна, из нее практически полностью устраняются

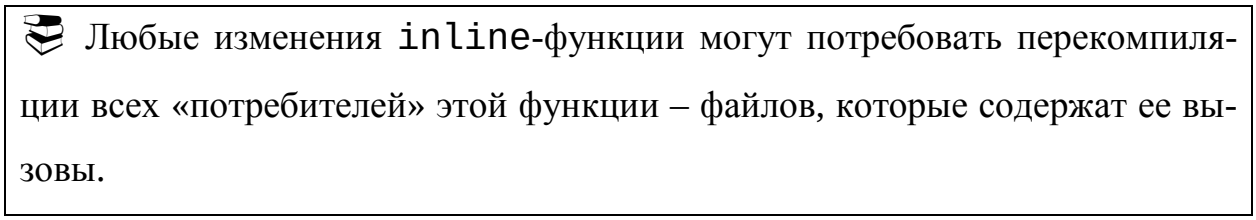
«нетворческие» операции – отслеживание границ памяти, в которой хранятся строковые данные.

2.9. Встраиваемые функции

Вызовы функций приводят к накладным расходам во время выполнения. В C++ для снижения этих накладных расходов – особенно для небольших функций – предусмотрен механизм *встраиваемых* (`inline`) *функций*, которые иногда еще называют подставляемыми.

Спецификация `inline` перед указанием типа результата в объявлении предлагает компилятору сгенерировать копию кода функции в соответствующем месте, чтобы избежать вызова этой функции. В результате получается множество копий кода функции, вставленных в программу, вместо единственного экземпляра кода, которому передается управление при каждом вызове функции.

Компилятор может игнорировать спецификацию `inline`.


Любые изменения `inline`-функции могут потребовать перекомпиляции всех «потребителей» этой функции – файлов, которые содержат ее вызовы.

Пример 26. Реализовать использование встраиваемой функции для вычисления максимума из двух значений.

```
#include <iostream>
using namespace std;
inline double max(double a, double b){
  return a>b?a:b;
}
int main(){
  cout<<max(3,2)<<endl;
  return 0;
}
```

2.10. Перегрузка функций

Перегрузка функции – возможность определить несколько функций с одним и тем же именем, если эти функции имеют разные наборы параметров (по меньшей мере, разные типы параметров).

Перегруженные функции, которые выполняют тесно связанные задачи, делают программы понятными и легко читаемыми.

Пример 27. Реализовать перегруженные функции для нахождения максимального из нескольких значений, возможно, разных типов.

Рассмотрим нахождение максимального значения для следующих вариантов параметров: два целых числа; три целых числа; два вещественных числа; две строки в стиле C; две структуры данных для хранения даты.

Листинг файла func_overload.cpp

```
#include <iostream>
#include <string.h>
using namespace std;

inline int max(const int a, const int b){
  return a>b?a:b;
}
inline int max(const int a, const int b, const int c) {
  int d=a>b?a:b;
  return d>c?d:c;
}

inline double max(const double a, const double b){
  return a>b?a:b;
}

const char* max(const char* a, const char* b) {
  if (strcmp(a,b)>0)
    return a;
  else
    return b;
}

struct date {
  int day, month, year;
};
```

```
date max(const date &a, const date &b){
  if (a.year>b.year)
    return a;
  else if (a.year<b.year)
    return b;
  else if (a.month>b.month)
    return a;
  else if (a.month<b.month)
    return b;
  else if (a.day>b.day)
    return a;
  else
    return b;
}

int main() {
  cout<<max(3,2)<<endl;
  cout<<max(3,7,2)<<endl;
  cout<<max(3.5,20.4)<<endl;
  cout<<max("Hello","By")<<endl;
  date a={27,03,2008}, b={31,03,2008}, c;
  c=max(a, b);
  cout<<c.day<<'.'<<c.month<<'.'<<c.year<<endl;
  return 0;
}
```

Рассмотрим подробнее функцию для нахождения максимальной строки.

```
const char* max(const char* a, const char* b){
  if (strcmp(a,b)>0)
    return a;
  else
    return b;
}
```

Строки сравниваются в *лексикографическом* порядке посредством функции `strcmp(const char* string1, const char* string2)` из стандартной библиотеки. Эта функция возвращает результат в соответствии с правилами, приведенными в табл. 3.

Для использования данной функции подключается заголовочный файл string.h (cstring).

Таблица 3

Результат сравнения строк

Значение	Отношение строк `string1` и `string2`
< 0	`string1` меньше, чем `string2`
==0	`string1` равна `string2`
> 0	`string1` больше, чем `string2`

Внутри тела функции для нахождения максимальной строки можно не использовать `else`. Тогда функция примет вид:

```
const char* max(const char* a, const char* b){
  if (strcmp(a,b)>0)
    return a;
  return b;
}
```

Оператор `return` не только возвращает значение, но и выполняет выход из функции. Именно поэтому в данном примере можно убрать `else`. Рекомендуется использовать или такой вариант, или условную операцию:

```
const char* max(const char* a, const char* b){
  return (strcmp(a,b)>0)?a:b;
}
```

Реализовать перегруженные функции для нахождения среднего арифметического для двух целых чисел и для двух вещественных.

Определить, возможно ли реализовать две перегруженные функции для двух целых чисел: одну, возвращающую вещественное значение, и одну, возвращающую целое (округленное среднее).

Следует помнить, что тип возвращаемого значения не участвует в формировании компилятором сигнатуры (внутреннего имени) функции. Поэтому в качестве перегружаемых функций нельзя использовать функции, различающиеся лишь типом возвращаемых значений.

Компилятор автоматически (неявно) приводит типы данных там, где это оправдано контекстом применения. Например, если целое значение присваивается вещественной переменной, компилятор незаметно вызовет функцию (или, более вероятно, вставит фрагмент кода) для приведения `int` к типу `float` или `double`. Операция приведения типов позволяет выполнять подобные преобразования явно.

Чтобы выполнить явное приведение типа в стиле C, следует указать нужный тип данных (вместе со всеми модификаторами) в круглых скобках слева от значения – переменной, константы, результата выражения или возвращаемого значения функции.

Например:

```
int b = 200;
unsigned long a = (unsigned long) b;
```

☺ Приведение типов часто становится источником всевозможных проблем. В общем случае количество таких операций должно быть минимальным.

Стандарт C++ описывает синтаксис операций явного приведения типов, способных полностью заменить операции приведения в стиле C. Новый синтаксис приведения типа (табл. 4) сразу бросается в глаза и отличается от других элементов программы.

Вернемся к рассмотрению функцию

```
const char* max(const char* a, const char* b)
```

Она возвращает значение типа `const char*`. В случае, если необходимо присвоить ее результат переменной типа `char*`, придется воспользоваться преобразованием типов.

```
(char*) max(a, b);                    //устаревшая форма
const_cast <char*>(max(a, b));        //рекомендуемая форма
```

Таблица 4

Синтаксис приведения типов в стиле C++

Оператор	Описание
`static_cast`	Для «безопасного» и «более или менее безопасного» приведения, включая то, которое может быть выполнено неявно. Например, автоматическое приведение типа.
`const_cast`	Изменение статуса объявлений `volatile` и/или `const`.
`reinterpret_cast`	Приведение к типу с совершенно иной интерпретацией. Принципиальная особенность этого приведения заключается в том, что для надежного использования значение должно быть приведено обратно к исходному типу. Тип, к которому выполняется приведение, обычно задействуется для манипуляций с битами или других неочевидных целей. Это самый опасный из всех видов приведения.
`dynamic_cast`	Понижающее приведение, безопасное по отношению к типам.

При получении адреса константного объекта результат представляет собой указатель на `const`, который без приведения типа нельзя присвоить указателю на неконстантный объект.

Операция `const_cast` позволяет лишить переменную статуса `const` или `volatile`.

☺ Несмотря на наличие возможности приведения типа в стиле C, рекомендуется использовать конструкцию `const_cast` в стиле C++.

Рассмотрим заголовок перегруженной функции для определения наибольшей даты:

```
date max(const date &a, const date &b)
```

Структуры можно рассматривать как пользовательские типы данных. Это позволяет пользоваться при их передаче в качестве аргументов теми же

приемами, что и для базовых типов данных. Структуры можно использовать и как возвращаемые значения.

☺ Рекомендуется для структур и других составных типов данных (массивов, строк, классов) использовать передачу параметров по ссылке. В случае если необходимо запретить изменение параметров внутри тела функции, следует использовать модификатор `const`.

Во многих случаях бывает удобно инициализировать структуру при объявлении.

```
date a={27,03,2008}, b={31,03,2008}, c;
```

Значения полей заключаются в фигурные скобки и перечисляются через запятую, аналогично инициализации массивов при их объявлении.

2.11. Аргументы по умолчанию

Если функция вызывается с большим количеством аргументов, многие из которых имеют одни и те же значения, повторять их при каждом вызове функций неудобно и утомительно. В таких случаях в C++ применяются *аргументы по умолчанию*, т. е. значения, которые автоматически подставляются компилятором, если аргумент не был указан при вызове.

📖 Аргументы по умолчанию могут быть расположены только в конце списка формальных параметров.

Пример 28. Описать функцию `TrimLeftC(S,C)`, удаляющую в строке `S` начальные символы, совпадающие с символом `C` (`S` является входным и выходным параметром, `C` – входной параметр). Даны три строки и три символа соответственно. Используя функцию `TrimLeftC`, преобразовать данные строки.

Поскольку очень часто требуется удалить начальные пробелы, то параметр C можно сделать параметром по умолчанию, и по умолчанию его значение будет равно символу пробела.

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

char* TrimLeftC(const char* S, char C=' ') {
  char *q=const_cast<char*>(S);
  while (*q!=0 && *q==C)
    q++;
  return q;
}

int main(){
  char *s,c;
  s=new char[101];
  cout<<endl<<"input string1 ";
  cin.getline(s,100);
  //удалить начальные пробелы
  s=TrimLeftC(s);
  cout<<endl<<"string1="<<s<<endl;
  cout<<endl<<"input string2 ";
  cin.getline(s,100);
  cout<<endl<<"input char ";
  cin>>c;
  s=TrimLeftC(s,c);
  cout<<endl<<"string2="<<s<<endl;
  //удаление пробелов в строковой константе
  s=TrimLeftC("   Hello       !");
  cout<<endl<<"string3="<<s<<endl;
  return 0;
}
```

Аргумент по умолчанию должен содержать значение, которое может требоваться чаще остальных значений. Это позволит при использовании данной функции в большинстве случаев проигнорировать его, но при необходимости можно будет задать другое значение, отличное от значения по умолчанию.

☺ Не используйте аргумент по умолчанию как условие, на основании которого выбирается одна из ветвей программы. Вместо этого постарайтесь разделить функцию на две или более перегруженные функции.

Аргументы по умолчанию должны упрощать вызов функции, особенно если она вызывается с большим количеством аргументов, принимающих типичные значения. Аргументы по умолчанию призваны упрощать не только запись, но и чтение вызовов.

 Существует еще одна важная ситуация, в которой применение аргументов по умолчанию является эффективным. Допустим, уже определена функция с неким набором аргументов, а через некоторое время выясняется, что в функцию нужно добавить дополнительные аргументы.

Объявление для всех новых аргументов значений по умолчанию гарантирует, что работоспособность клиентского кода, использующего прежний интерфейс, не будет нарушена.

2.12. Шаблоны функций

Перегруженные функции обычно используются для выполнения похожих по синитаксису, но разных по семантике операций. Если же для каждого типа данных должны выполняться идентичные операции, то более компактным и удобным решением является использование шаблонов функций.

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

Шаблоны функций и шаблонные функции – это не одно и то же! Шаблонная функция – это «реализация функции по шаблону».

➢ В языке С некоторым аналогом простейших шаблонов являются макросы, определяемые директивой препроцессора `#define`. Однако при использовании макросов компилятор не выполняет проверку соответствия типов, из-за чего нередко возникают серьезные побочные эффекты. Шаблоны же позволяют полностью контролировать соответствие типов.

Пример 29. Реализовать и использовать шаблон функции для вывода элементов массива.

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

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

template <typename T>
T sumArray(const T* array, int count){
  int sum=0;
  for (int i=0; i<count; i++)
    sum+=array[i];
  return sum;
}

int main(){
  const int aCount = 5, bCount =7, cCount = 6;
  int a[aCount]={1,2,3,4,5};
  double b[bCount]={1.1,2.2,3.3,4.4,5.5};
  char c[cCount]="Hello";
  printArray(a,aCount);
  printArray(b,bCount);
  printArray(c,cCount);
  cout<<endl<<sumArray(a,aCount)<<endl;
  return 0;
}
```

Все описания шаблонов функций начинаются с ключевого слова `template`, за которым следует список формальных параметров шаблона,

заключаемый в угловые скобки (< и >). Каждому формальному параметру типа должно предшествовать ключевое слово `class` или `typename`.

В списке параметров шаблона функции ключевые слова `typename` и `class` имеют одинаковый смысл и, следовательно, взаимозаменяемы. Любое из них может использоваться для объявления разных параметров-типов шаблона в одном и том же списке. Ключевое слово `typename` было добавлено в язык как часть стандарта C++.

☺ Рекомендуется использовать `typename` для формальных параметров типа при описании шаблонов.

Ключевое слово `typename` упрощает компилятору разбор определений шаблонов. Желающим узнать об этом подробнее рекомендуем обратиться к книге Б. Страуструпа «Язык программирования C++. Специальное издание» (М., 2004).

В шаблоне функции `printArray` объявляется один формальный параметр `T` для типа элементов массива, который будет выводиться функцией `printArray`.

В шаблоне функции для определения типов параметров, типа возвращаемого функцией значения и типов локальных переменных, могут быть использованы: встроенные типы; типы, определяемые пользователем; типы, задаваемые как формальные параметры.

Используемый в объявлении идентификатор `T` называется параметром *типа*.

📖 Каждый формальный параметр из описания шаблона функции должен появиться в списке параметров функции, по крайней мере, один раз.

Когда компилятор обнаруживает в тексте программы вызов функции `printArray`, он заменяет `T` во всей области определения шаблона на тип

первого параметра функции printArray и создает шаблонную функцию вывода массива указанного типа данных. После этого вновь созданная функция компилируется. Процесс конструирования шаблонной функции называется *конкретизацией* шаблона.

Например, при вызове

```
printArray(a,aCount);
```

реализация функции для типа int будет выглядеть следующим образом:

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

При вызове

```
printArray(b,bCount);
```

реализация функции для типа double будет выглядеть так:

```
void printArray(const double* array, int count){
  for (int i=0; i<count; i++)
    cout << array[i] << " ";
  cout << endl;
}
```

Шаблон должен быть описан либо в том же файле (.cpp), где и используется, либо в заголовочном файле (.h), который подключается к этому файлу (.cpp).

При многофайловой организации проекта может создаваться слишком много одинаковых конкретизаций шаблонов (шаблонных функций) для одинаковых списков параметров. Для предотвращения такой ситуации можно использовать явное объявление конкретизации. Тогда шаблонная функция будет создана заранее ровно один раз для одного списка параметров, указанного в объявлении.

В явном объявлении конкретизации за ключевым словом template идет объявление шаблона функции, в котором его аргументы указаны явно.

```
template <class T>
void printArray(const T* array, int count)
{/* ... */}

// явное объявление конкретизации
template void printArray < int >(const int*, int);
```

Не всегда удается написать шаблон функции, который годился бы для всех возможных типов, с которыми он может быть конкретизирован.

Иногда общее определение, предоставляемое шаблоном, для некоторых типов вообще не работает, а для некоторых работает неэффективно. Например, когда шаблон функции суммирования конкретизируется с аргументом типа `char`, то обобщенное определение оказывается семантически некорректным. В этом случае необходимо предоставить специализированное определение для конкретизации шаблона (*специализации*) или использовать перегруженную функцию с нужным списком параметров.

Порядок определения, какой экземпляр функции соответствует данному вызову, следующий:

- ✓ Сначала компилятор ищет функцию или конкретизацию шаблона, которая точно соответствует по своему имени и типам параметров вызываемой функции.
- ✓ Если на этом этапе компилятор терпит неудачу, то он ищет шаблон функции, с помощью которого он может сгенерировать шаблонную функцию с точным соответствием типов параметров и имени функции. Если такой шаблон обнаруживается, то компилятор генерирует и использует соответствующую шаблонную функцию. При этом автоматическое преобразование типов не производится.
- ✓ И только в случае неудачи в качестве последней попытки компилятор последовательно выполняет процесс подбора перегруженной функции с учетом автоматического преобразования типов.

2.13. Указатели на функции

В языке C++ аналогом процедурного типа языка Паскаль является тип – указатель на функцию.

Рассмотрим пример:

```
void (*funcPtr)();
```

В данном случае `funcPtr` является указателем на функцию, которая не имеет аргументов и возвращаемого значения.

Если в описании убрать круглые скобки:

```
void *funcPtr();
```

то в этом случае данная строка объявляет не переменную-указатель на функцию, а собственно функцию без параметров, которая возвращает `void *`.

 Проанализировать следующие объявления и записать, что означает каждое из них:

```
1)  void *(*(*fp1)(int))[10];
2)  float (*(*fp2)(int,int,float))(int);
3)  typedef double (*(*(*fp3)())[10])();
4)  int (*(*f4())[10])();
```

После определения указателя на функцию перед дальнейшим использованием ему необходимо присвоить адрес функции. Подобно тому, как адрес массива задается именем массива без квадратных скобок, адрес функции задается именем функции без списка аргументов и без круглых скобок. Например, имеется функция `int func(int,double)`. Тогда ее адрес – `func`. Допустим и более очевидный синтаксис `&func()`.

Пример 30. Определить и применить указатель на функцию.

```
#include <iostream>
using namespace std;
void func(){
  cout<<"func() called... "<<endl;
}
```

```
int main(){
  void (*fp)();  //Определение указателя на функцию
  fp=func;  //Инициализация
  (*fp)();  //Разыменование означает вызов
  void (*fp2)() = func;  //Определение и инициализация
  (*fp2)();
}
```

Стоит особо остановиться на такой конструкции, как *массив указателей на функции*. Этот механизм воплощает концепцию *кода, управляемого таблицами*. Вместо конструкций выбора выполняемая функция выбирается в зависимости от переменной состояния. Такой прием особенно удобен при частом добавлении или удалении функций из таблицы, а также при динамическом создании или изменении таблицы.

Пример 31. Дан массив целых чисел из N элементов. Создать функцию Accumulate(f,s,A,n), применяющую функцию f(A,n) к массиву A. Функция f(A, n)передается в качестве параметра и возвращает целое значение. Результат применения функции f(A,n) к массиву A записать в выходной параметр s функции Accumulate(f,s,A,n). С помощью функции Accumulate(f,s,A,n) найти минимальный элемент в массиве, сумму и произведение элементов массива.

Заголовочный файл funcs.h

```
#ifndef FUNCS_H
#define FUNCS_H
void input (int a[], int& n, int maxn);
int min(int a[], int n);
int sum(int a[], int n);
int mult(int a[], int n);
typedef int (*pf) (int[], int);
void Accumulate(pf f, int& s, int a [], int n);
#endif
```

В заголовочном файле используется определение типа указателя на функцию

```
typedef int (*pf) (int[], int);
```

Параметр такого типа передается в функцию Accumulate.

Описания объявленных функций – файл funcs.cpp

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

void input (int a[], int& n, int maxn){
  do {
    cout << "array size ";
    cin>> n;
  }
  while (n < 1 || n > maxn);
  for (int i = 0; i < n; i++) {
    cout << i <<" array element ";
    cin >> a[i];
  }
}

int min(int a[], int n){
  int min = a[0];
  for (int i = 1; i < n; i++)
    if (a[i] < min)
      min = a[i];
  return min;
}

int sum(int a[], int n){
  int s = 0;
  for (int i = 0; i < n; i++)
    s += a[i];
  return s;
}

int mult(int a[], int n){
  int p = 1;
  for (int i = 0; i < n; i++)
    p *= a[i];
  return p;
}
void Accumulate(pf f, int& s, int a [], int n){
  s = f(a,n) ;
  return;
}
```

Программа, демонстрирующая передачу параметров и использование массива указателей на функции:

```
#include "funcs.h"
#include <iostream>
using namespace std;
```

```
int main(){
  const int nmax = 100;
  int a[nmax], n;
  input(a, n, nmax);
  int s;
  Accumulate(min, s, a, n);
  cout << "min = " << s << endl;
  Accumulate(mult, s, a, n);
  cout << "mult = " << s << endl;
  Accumulate(sum, s, a, n);
  cout << "sum = " << s << endl;
  int (*func_table[])(int*,int) = {min,mult,sum};
  string namef[3]={"min","mult","sum"};
  for (int i=0; i<3; i++) {
     Accumulate(func_table[i], s, a, n);
     cout << namef[i]<< " = "<< s << endl;
  }
  return 0;
}
```

При вызове функции

```
void Accumulate(pf f, int& s, int a [], int n)
```

в качестве фактического параметра `f` можно использовать как имя функции, так и указатель на функцию. Оба варианты приведены в теле функции `main()`.

Использование массива указателей на функции

```
int (*func_table[])(int*,int) = {min,mult,sum};
```

позволяет передавать указатель на функцию по номеру, что может быть использовано при организации программ с выбором функции из многих вариантов.

2.14. Ввод/вывод и работа с файлами

В языки C и C++ функции ввода/вывода не встроены. Первоначально в языке C реализация ввода/вывода была оставлена на усмотрение разработчиков компиляторов. Практически для большинства компиляторов использовался набор функций, первоначально разработанный для среды

UNIX. По стандарту ANSI C этот набор с заголовочным файлом cstdio является обязательным компонентом стандартной библиотеки C.

При запуске C-программы операционная система открывает три файла и обеспечивает три файловые ссылки на них. Этими файлами являются: стандартный ввод, стандартный вывод и стандартный файл ошибок. Соответствующие им указатели называются `stdin`, `stdout` и `stderr`; они описаны также в cstdio. Файловые указатели `stdin`, `stdout` и `stderr` представляют собой объекты типа `FILE*`. Это константы, а не переменные, следовательно, им нельзя ничего присваивать. Обычно `stdin` соотнесен с клавиатурой, а `stdout` и `stderr` – с экраном. Однако их можно связать с файлами. Часто вывод в `stderr` отправляется на экран, даже если вывод `stdout` перенаправлен в другое место.

В C++ чаще используется ввод/вывод при помощи набора классов, определенных в заголовочных файлах iostream и fstream. Использованные ранее в примерах объекты `cin` и `cout` определены в заголовочном файле iostream. При этом `cin` является объектом класса `istream` и соответствует стандартному потоку ввода, связанному по умолчанию со стандартным устройством ввода – клавиатурой. Аналогично, `cout` является объектом класса `ostream` и соответствует стандартному потоку вывода, который по умолчанию связан со стандартным устройством вывода – монитором. Еще двумя стандартно определенными потоковыми объектами являются `cerr` и `clog`. Эти объекты используются для вывода отладочной информации и сообщений об ошибках, по умолчанию они тоже связаны с монитором. Отличие между ними заключается в том, что поток `cerr` является небуферизованным, поэтому в нем будет вся направляемая в поток информация, даже если программа завершится аварийно.

2.15. Работа с текстовыми файлами в стиле C++

Пример 32. Создать текстовый файл, содержащий ведомость о результатах сдачи студентами трех экзаменов. Вывести содержимое этого файла на экран.

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main(){
  char filename[20];
  cout<<"Enter name for a new file->";
  cin>>filename;
  ofstream fout(filename);
  fout<<"Students\n";
  char name[20];
  int mark[3],n;
  cout<<"Enter number of students->";
  cin>>n;
  fout<<"----------NAME-------|-M-|-A-|-I-|\n";
  for (int i=0; i<n; i++){
    cout<<"Student name->";
    cin>>name;
    cout<<"Sudent mark's->";
    cin>>mark[0]>>mark[1]>>mark[2];
    fout<<setw(20)<<name<<" | "<<
          mark[0]<<" | "<<mark[1]<<" | "<<mark[2]<<"\n";
  }
  fout.close();
  ifstream fin(filename);
  char ch;
  cout<<"Contents of file \n";
  while(fin.get(ch))
    cout<<ch;
  fin.close();
  return 0;
}
```

Для записи информации в файл создается объект класса `ofstream`. Используемый конструктор с параметром (именем файла) создает новый файл с таким именем (или очищает существующий файл) и открывает его в текстовом режиме для записи.

```
ofstream fout(filename);
```

Для записи в текстовый файл можно использовать операцию << так же, как и для стандартного потока вывода cout. При этом могут быть использованы средства форматирования, например, манипулятор setw() для установления ширины поля вывода. Манипуляторы описаны в iomanip.

```
fout<<"----------NAME-------|-1-|-2-|-3-|\n";
fout<<setw(20)<<name<<" | "<<mark[0]
    <<" | "<<mark[1]<<" | "<<mark[2]<<"\n";
```

После завершения записи файл должен быть закрыт; это гарантирует, что буфер будет выгружен.

```
fout.close();
```

Для того чтобы прочитать содержимое файла, используется объект класса ifstream. Конструктор с одним параметром связывает этот объект с только что созданным файлом и открывает файл для чтения в текстовом режиме.

```
ifstream fin(filename);
```

Поскольку в задаче не требуется разбирать и анализировать содержимое текстового файла, самый простой способ – прочитать его посимвольно. При этом используется не операция >>, а функция посимвольного ввода get(char&)). Это связано с тем, что операция >> игнорирует пробельные (служебные) символы, а функция get(char&) считывает и сохраняет в параметре любой символ, даже если это пробел, символ табуляции или символ новой строки. При этом, если считан символ, то функция возвращает значение true, при достижении конца файла функция возвратит значение false. Это позволяет использовать вызов в условии цикла

```
while(fin.get(ch))
  cout<<ch;
```

Пример 33. Написать программу, подсчитывающую общее количество символов в нескольких текстовых файлах. Список имен файлов, для

которых нужно выполнить подсчет, задается в списке аргументов командной строки при запуске программы.

```
// count.cpp – подсчет количества символов в файлах по списку
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc,char* argv[]){
  if (argc==1){
    cerr<<"Usage: "<<argv[0]<<" filename[,filename[…]]\n";
    exit(1);
  }
  ifstream fin;
  long count;
  long total =0;
  char ch;
  for (int file=1; file<argc; file++){
    fin.open(argv[file]);
    if (!fin.is_open()){
      cerr<<"couldn't open file "<<argv[file] << "\n";
      fin.clear(); // сброс failbit
      continue;
    }
    count = 0;
    while (fin.get(ch))
      count++;
    cout<<count<< " in "<<argv[file]<<"\n";
    total += count;
    fin.close();
    fin.clear();
  }
  cout<<total<<" in all files \n";
  return 0;
}
```

В программах, использующих файлы, часто применяют прием ввода имен обрабатываемых файлов через аргументы командной строки. Для рассматриваемой программы, например, после ее компиляции и получения исполнимого файла count.exe на диске G в корневом каталоге, подсчитать общее количество символов в файлах text1.txt, text2.txt, info.dat можно следующим образом:

```
G:\count text1.txt text2.txt info.dat
```

Чтобы обеспечить возможность работы в программе с аргументами командной строки, используется альтернативный вариант заголовка функции `main()`:

```
int main(int argc,char* argv[])
```

Параметр `argc` передает в программу количество аргументов командной строки, в число аргументов командной строки входит и само имя выполняемой программы. Параметр `argv[]` представляет собой массив указателей на символьные строки, содержащие аргументы командной строки.

Обычно программы, использующие аргументы командной строки, начинаются с проверки правильности вызова и, в случае отсутствия необходимых аргументов, выдачи сообщения с подсказкой.

```
if (argc==1){
  cerr<<"Usage: "<<argv[0]<<" filename[,filename[…]\n";
  exit(1);
}
```

Функция `exit(1)` приводит к нормальному завершению программы. Значение параметра функции передается операционной системе. Значение 0 означает успешное завершение программы, а отличное от 0 может интерпретироваться как код ошибки.

Для работы с файлами в программе создан объект класса `ifstream`. В отличие от предыдущего примера, используется конструктор без параметров. Созданный объект не связан ни с каким файлом.

Для связи и открытия файла в нужном режиме используется метод `open()`:

```
fin.open(argv[file]);
```

Он может завершиться успешно или с ошибкой. Каждый объект потокового класса содержит элемент данных (битовую маску), описывающий состояние потока. При возникновении таких ситуаций, как достижение

конца файла, невозможность прочесть очередной байт (символ), попытка чтения из недоступного файла, попытка записи в защищенный файл и пр., один из битов состояния потока устанавливается в 1. Нормальная работа с потоками возможна только тогда, когда все биты маски равны 0.

Для проверки состояния потока используются соответствующие методы. Для того чтобы проверить, не было ли задано имя несуществующего файла, и пропустить его при обработке, используется метод `is_open()`, проверяющий состояние потока после открытия.

```
if (!fin.is_open()) {
  cerr<<"couldn't open file "<<argv[file] << "\n";
  fin.clear(); // сброс failbit
  continue;
}
```

☺ Правильно написанные программы должны проверять результат выполнения операций с файлами и корректно обрабатывать возникающие ошибки времени выполнения.

Чтобы продолжить работу со следующим файлом, нужно сбросить информацию о состоянии потока (обнулить все биты) с помощью метода `clear()`.

Применять метод `clear()`нужно всякий раз перед открытием следующего файла, так как достижение конца файла тоже отражается в маске состояния потока.

Как и в предыдущем примере, для подсчета всех символов в файлах используется посимвольный ввод методом `get()`. Этот метод возвращает значение `true`, пока поток имеет «хорошее» состояние (все биты сброшены). При достижении конца файла меняется состояние потока, и метод возвращает значение `false`.

Пример 34. Написать программу, выполняющую несколько способов обработки текстовых файлов (вывод содержимого на экран, создание копии файла, выдача содержимого файла по словам). Выбор режима обработки организовать посредством меню.

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

void echoFile (char* fileName){
  ifstream inFile(fileName);
  if (!inFile.is_open()){
    cerr<<"Can't open "<<fileName<<"\n";
    return;
  }
  char c;
  while (inFile.get(c))
    cout<<c;
  cout<<endl;
  inFile.close();
}

void copyTextFile(char* inName, char* outName){
  char c;
  ifstream inFile(inName);
  if (!inFile.is_open()){
    cerr<<"Can't open "<<inName<<"\n";
    return;
  }
  ofstream outFile(outName);
  while (inFile.get(c))
    outFile<<c;
  inFile.close();
  outFile.close();
}

//пропуск пробелов
void skipBlank(ifstream& inFile){
  while (inFile.peek() ==' ')
    inFile.ignore(1);
}

//выделяет и распечатывает по очереди слова в строке
void echoWord(ifstream& inFile){
  char w;
  while (inFile.peek() != '\n'){
```

```
    while (inFile.peek() !=' ' && inFile.peek()!='\n') {
      inFile.get(w);
      cout<<w;
    }
    cout<<"\n";
    skipBlank(inFile);
  }
  inFile.ignore(1);
}

//выделение и печать каждого слова в текстовом файле
void printWords(char* nameFile){
  ifstream inFile(nameFile);
  if (!inFile.is_open()){
    cerr<<"Can't open "<<nameFile<<"\n";
    return;
  }
  skipBlank(inFile);
  while (inFile.peek() != EOF)
    echoWord(inFile);
  inFile.close();
}

int main(){
  char n, name[20],name1[20];
  do {

    cout<<"1. Echo text file \n";
    cout<<"2. Copy text file \n";
    cout<<"3. Print words \n";
    cout<<"4. Exit \n";

    cin>>n;
    if ((n>'0')&&(n<'5')) {
      switch (n) {
        case '1':
                 cout<<"File name ->";
                 cin>>name;
                 echoFile (name);
                 break;
        case '2':
                 cout<<"File name ->";
                 cin>>name;
                 cout<<"Copy name ->";
                 cin>>name1;
                 copyTextFile(name,name1);
                 echoFile(name1);
                 break;
```

```
        case '3':
                 cout<<"File name ->";
                 cin>>name;
                 printWords(name);
                 break;
      }
    }
    else{
      cout<<"Error number \n";
    }
  }
  while (n!='4');
    return 0;
}
```

Приведенные в программе функции

```
void echoFile (char* fileName)
void copyTextFile(char* inName, char* outName)
```

организованы так же, как и в предыдущих примерах. Входными параметрами для них являются имена файлов.

Для выделения и печати каждого слова в текстовом файле описаны основная функция `void printWords(char* nameFile)` и две вспомогательные функции. При реализации вспомогательных функций использованы следующие предположения. Словом является любая последовательность непробельных символов. Между словами, в начале и конце строки может быть любое количество пробелов. Слово не может быть расположено на нескольких строках.

Функция `void skipBlank(ifstream& inFile)` пропускает пробелы между словами, в начале и конце строки. Входным параметром для этой функции является ссылка на объект класса `ifstream`. В функции использован метод `peek()`, который позволяет проанализировать очередной символ из потока, не читая его. Если очередной символ – пробел, его можно пропустить; для этого использован метод `ignore()`. Благодаря такой организации функция остановится перед первым непробельным символом, оставив его в потоке.

```
void skipBlank(ifstream& inFile){
  while (inFile.peek() == ' ')
    inFile.ignore(1);
}
```

Функция `void echoWord(ifstream& inFile)` выделяет и распечатывает по очереди слова в строке. В ней использован тот же принцип «заглядывания» на символ вперед, чтобы определить конец слова или конец строки. Для пропуска пробелов между словами внутри строки используется функция `skipBlank()`.

Пример 35. Написать программу, позволяющую вводить строки и дописывать их в текстовый файл.

```
#include <iostream>
#include <fstream>
using namespace std;
void echoFile (char* filename){
  ifstream inFile(filename);
  if (!inFile.is_open()){
    cerr<<"Can't open "<<filename<<"\n";
    return;
  }
  char c;
  while (inFile.get(c))
    cout<<c;
  cout<<endl;
  inFile.close();
}
void appendTextFile(char* filename){
  ofstream fout(filename, ios::out | ios::app);
  if (!fout.is_open()){
    cerr<<"Can't open "<<filename<<"\n";
    exit(1);
  }
  cout<< "enter new strings (blank line to exit)\n";
  char line[80];
  cin.get(line,80);
  while (line[0]!='\0'){
    fout<<line<<"\n";
    cin.ignore(1,'\n');
    cin.get(line,80);
  }
  fout.close();
}
```

```
int main() {
  appendTextFile("copy.dat");
  echoFile("copy.dat");
  return 0;
}
```

В функции `appendTextFile()` продемонстрирована возможность устанавливать режим использования файла. Режим файла служит для описания характера использования файла – для чтения, для записи, для добавления, текстовый или бинарный и т. д. Режим файла можно задавать в конструкторе или в методе `open()` вторым параметром. При использовании только одного параметра режим задается по умолчанию. Например, для класса `ifstream` по умолчанию используется режим, задаваемый константой `ios::in` (открыть для чтения). Для класса `ofstream` значение по умолчанию `ios::out|ios::trunc` (открыть для записи и усечь файл). Поразрядный оператор «или» (`|`) используется для объединения двух режимов.

Использованный в функции `appendTextFile()` режим означает, что файл должен быть открыт для записи с сохранением имеющейся в нем информации и добавлением новых данных в конец файла.

```
ofstream fout(filename, ios::out | ios::app);
```

Добавление данных к файлу осуществляется построчно в предположении, что каждая добавляемая строка не длиннее 80 символов. Ввод пустой строки является признаком окончания ввода.

```
while (line[0]!='\0')
```

При использовании функции `cin.get(line,80)` ввод строки в переменную `line` завершается или при вводе символа новой строки, или после ввода 79 символов. Символ новой строки остается в потоке ввода и должен быть пропущен. Однако в потоке ввода могут остаться и другие

символы, если была набрана строка из 80 или более символов. Чтобы перейти к вводу следующей строки, нужно пропустить эти символы:
`cin.ignore(1,'\n');`

2.16. Работа с бинарными файлами в стиле C++

Бинарные (двоичные) файлы можно использовать для организации внешних таблиц (массивов структур во внешней памяти).

Пример 36. Написать программу для организации работы с таблицей, хранящей данные о студентах (имя, год рождения, средний балл по итогам последней сессии).

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
inline void end_of_line() {cin.ignore(1,'\n');}
struct dates{
  char name[20];
  int year;
  double rate;
};
void toFile(char* nameF){
  dates p;
  ofstream fout(nameF,ios::app |ios::binary);
  if (!fout.is_open()){
    cerr<<"Can't open "<<nameF<<"\n";
    exit(1);
  }
  cout<<"Enter name \n";
  cin.get(p.name,20);
  while (p.name[0]!='\0'){
    end_of_line();
    cout<<" Enter year ";
    cin>>p.year;
    cout<<"Enter rate ";
    cin>>p.rate;
    fout.write(reinterpret_cast <char*>(&p),sizeof p);
    end_of_line();
    cout<<"Enter name (blank line to quit) \n";
    cin.get(p.name,20);
  }
  fout.close();
}
```

```
void echoFile(char* nameF){
  dates p;
  ifstream fin(nameF, ios::in | ios::binary);
  if (fin.is_open()) {
    while (fin.read(reinterpret_cast <char*>(&p),sizeof p)) {
      cout<< setw(20)<<p.name<<" : "
          <<setw(10)<<p.year<<setprecision(2)
          <<setw(15)<<p.rate<<"\n";
    }
  }
  fin.close();
}

void numbDates (char* nameF, int n){
  dates p;
  fstream finout(nameF, ios::in | ios::out | ios::binary);
  streampos place = n * sizeof p;
  finout.seekg(place);
  if (finout.fail()){
    exit(1);
  }
  finout.read(reinterpret_cast <char*>(&p),sizeof p);
  cout<< setw(20)<<p.name<<" : "<<setw(10)<<p.year
      <<setprecision(2)<<setw(15)<<p.rate<<"\n";
  finout.close();
}

int main() {
  toFile("inf.dat");
  echoFile("inf.dat");
  numbDates("inf.dat",2);
  numbDates("inf.dat",1);
  numbDates("inf.dat",0);
  return 0;
}
```

Использование двоичного режима файла приводит к тому, что данные пересылаются из памяти в файл или наоборот без какого-либо их преобразования. Чтобы сохранять данные в двоичном формате, при создании потокового объекта (или при открытии) необходимо указать режим `ios::binary`. В отличие от текстового режима, этот режим должен быть задан явно. При явном указании режима требуется определить все режимы открытия файла, соединив их поразрядной операцией | (или).

```
ifstream fin(nameF, ios::in | ios::binary);
ofstream fout(nameF,ios::app |ios::binary);
fstream finout(nameF, ios::in | ios::out | ios::binary);
```

Для записи данных в двоичном формате используется метод `write()`:

```
fout.write(reinterpret_cast <char*>(&p),sizeof p);
```

Этот метод копирует определенное число байтов из памяти в файл. Количество байтов, которое должно быть скопировано, задается вторым параметром. Первый параметр определяет адрес, где расположены данные, которые необходимо скопировать. Особенностью метода является то, что адрес должен быть преобразован к типу «указатель на char». Для этого используется преобразование в стиле C++:

```
reinterpret_cast <char*>(&p)
```

Для чтения из двоичного файла используется метод `read()`, имеющий такой же список параметров, как и метод `write()`:

```
fin.read(reinterpret_cast <char*>(&p),sizeof p);
```

Данный метод возвращает значение `true`, если операция чтения завершилась нормально, и `false` – в случае возникновения ошибки, например, при достижении конца файла.

Поскольку при организации внешней таблицы файл состоит из записей одинакового размера, легко обеспечить доступ к записи с определенным номером. Для этого используются методы: `seekg()` – с объектами классов `ifstream` и `fstream`, и `seekp()` – с объектами классов `ofstream` и `fstream`.

Оба метода требуют использования двух параметров. Первый параметр определяет количество байт, на которое нужно переместиться от позиции, задаваемой вторым параметром. Для второго параметра определено значение по умолчанию – «от начала файла».

2.17. Работа с файлами в стиле языка С

Пример 37. Написать программу, объединяющую несколько именованных файлов и направляющую результат в стандартный вывод.

```
#include <cstdio>

void filecopy(FILE *ifp,FILE *ofp){
  int c;
  while ((c=getc(ifp))!=EOF)
    putc(c,ofp);
}

int main(int argc,char *argv[]){
  FILE *fp;
  char *prog=argv[0]; //имя программы
  //нет аргументов, используется стандартный ввод
  if (argc==1)
    filecopy(stdin,stdout);
  else
    while (--argc>0)
      if ((fp=fopen(*++argv,"r"))==NULL){
        fprintf(stderr,
                "%s: I can not open file %s\n", prog, *argv);
        return(1);
      }
      else {
        filecopy(fp,stdout);
        fclose(fp);
      }
  if (ferror(stdout)) {
    fprintf(stderr,
            "%s: error writing to stdout \n", prog);
    return(2);
  }
  return 0;
}
```

Вначале файл должен быть открыт библиотечной функцией `fopen`. Функция `fopen` получает внешнее имя файла и информацию о режиме открытия и возвращает файловый указатель `fp` на структуру типа `FILE`. Для использования и функции, и структуры необходимо подключить файл cstdio.

Параметр для режима открытия – строка. Возможные режимы открытия приведены в табл. 5.

Таблица 5

Режимы открытия файлов

"r"	Открыть существующий файл для чтения.
"w"	Открыть новый файл для записи.
"a"	Открыть файл для дозаписи в конец. Если файл не существует, он создается.
"r+"	Открыть существующий файл для чтения и записи.
"w+"	Открыть новый файл для чтения и записи.
"a+"	Открыть файл для чтения и дозаписи в конец. Если файл не существует, он создается.

По умолчанию файл открывается в текстовом режиме. В случае открытия файла в бинарном режиме в строку необходимо добавить букву «b». При наличии любой ошибки чтения-записи `fopen` возвращает `NULL`.

```
if ((fp=fopen(*++argv,"r"))==NULL){…}
```

Данная строка показывает, что открытие файла удобно совмещать с проверкой корректности данной операции.

Функция `fclose(fp)` – обратная по отношению к `fopen`. Она разрывает связь между файловым указателем и внешним именем и освобождает буфер, в котором функция `putc` накопила предназначенные для вывода данные.

Рассмотрим функцию

```
void filecopy(FILE *ifp,FILE *ofp)
```

Она предназначена для посимвольного копирования содержимого одного файла в другой.

Функция getc возвращает следующую литеру из файла (потока), на который ссылается при помощи *fp; в случае исчерпания файла или ошибки она возвращает EOF:

```
c=getc(ifp);
```

Функция putc пишет литеру в файл и возвращает записанную литеру или EOF в случае ошибки:

```
putc(c,ofp);
```

Следует помнить, что если в командной строке присутствуют аргументы, то они рассматриваются как имена последовательно обрабатываемых файлов. Если аргументов нет, то обработке подвергается стандартный ввод:

```
int main(int argc,char *argv[]) {...}
```

Программа сигнализирует об ошибках двумя способами. Первый – сообщение об ошибке при помощи fprintf посылается в stderr с тем, чтобы оно попало на экран, а не оказалось в файле вывода. Имя программы, хранящееся в argv[0], включено в сообщение, чтобы в случаях, когда программа работает совместно с другими, был ясен источник ошибок. Второй способ указать на ошибку – использовать инструкцию:

```
return выражение;
```

Функция ferror(stdout) выдает ненулевое значение, если в файле fp была обнаружена ошибка. Хотя при выводе редко возникают ошибки, все же они возможны (например, диск оказался переполненным). Поэтому в программах их желательно контролировать.

Функция fprintf идентична функции printf форматированного вывода на экран в языке С с той лишь разницей, что первым ее аргументом является указатель, ссылающийся на файл, для которого осуществляется вывод, формат же указывается вторым аргументом (см. об этом книгу Б. Кернигана и Д. Ритчи «Язык программирования Си» (М., 1992)).

```
int fprintf(FILE *fp, char *format, ...)
```

Пример 38. Написать программу для демонстрации использования функции `fprintf` для различных типов данных. Вывод данных необходимо осуществить в файл fprintf.out.

```
#include <cstdio>
#include <cprocess>

FILE *stream;

int main( void ) {
  int i = 10;
  double fp = 1.5;
  char s[] = "this is a string";
  char c = '\n';

  fopen_s(&stream, "fprintf.out", "w");
  fprintf(stream, "%s%c", s, c);
  fprintf(stream, "%d\n", i);
  fprintf(stream, "%f\n", fp);
  fclose(stream);
  system("type fprintf.out");
}
```

Содержимое файла fprintf.out выводится на экран средствами функции `system` для вызова команды TYPE операционной системы. Для функции `system` необходимо подключить файл cprocess.

В результате изучения модуля студент должен уметь:

- работать со статическими и динамическими массивами;
- работать со строками как в стиле C, так и в стиле C++;
- использовать указатели при работе с массивами и строками в стиле C;
- описывать и использовать шаблоны функций;
- работать с указателями на функции;
- работать с файлами как в стиле C, так и в стиле C++.

В результате изучения модуля студент должен знать:

- как описывать и использовать статические массивы; как создаются динамические массивы; как можно создать нерегулярный динамический массив (матрицу);
- в чем заключается связь массивов и указателей; какие преимущества дает использование указателей при работе с массивами;
- чем отличаются строки в стиле С и строки, основанные на библиотеке классов string;
- какую роль играют указатели при работе со строками в стиле С;
- какие особенности использования функций из библиотеки cstring необходимо учитывать для организации правильной работы со строками в стиле С; какие преимущества имеет библиотека классов string;
- как использовать массивы в качестве параметров в функциях; какие новые возможности в описании и использовании функций появились в языке С++;
- какие преимущества дают такие механизмы, как встраиваемые функции, перегрузка функций, значения параметров по умолчанию;
- особенности приведения типов в С++;
- основные принципы работы с файлами как в стиле С, так и в стиле С++.

Вопросы для рубежного контроля

1) Привести примеры описания статического одномерного и двумерного массива.
2) Какие способы описания параметров-массивов в функциях можно использовать?
3) Как можно описать и создать динамический одномерный массив?

4) Как можно уничтожить динамический одномерный массив?
5) Как описывается динамический двумерный массив? В каком порядке выделяется память под динамический двумерный массив?
6) Как можно создать нерегулярный динамический двумерный массив?
7) Как можно уничтожить динамический двумерный массив?
8) Как для строк в стиле С определяется конец строки?
9) Какие ошибки возможны при работе со строками в стиле С?
10) Существует ли операция конкатенации для строк, завершающихся нулевым символом?
11) Как организовать обработку строк, завершающихся нулевым символом, с помощью указателей?
12) Назвать преимущества, которые имеют объекты класса `string` над С-строками с точки зрения простоты использования.
13) Для чего предназначены встраиваемые функции? Каковы их преимущества по сравнению с макроподстановками директивой `define`?
14) Для чего используются перегруженные функции?
15) Каким правилам должны удовлетворять перегруженные функции?
16) Как согласуются перегрузка функций и аргументы по умолчанию?
17) Для чего необходимо явное приведение типов?
18) Что такое шаблон функции?
19) Что такое шаблонная функция?
20) Как можно описать и использовать указатель на функцию?

Проектные задания к модулю

1. Описать функцию `MoveLeft(A,N,K)`, осуществляющую циклический сдвиг элементов вещественного массива `A` размера `N` на `K` позиций влево `(K<N)`. Массив `A` – входной и выходной параметр, `N` и `K` – входные

параметры. Даны массив размера N и числа K1, K2. С помощью функции MoveLeft осуществить сдвиг элементов данного массива на K1 позиций влево, а затем – сдвиг элементов полученного массива на K2 позиций влево, выводя элементы преобразованного массива на экран после каждого вызова процедуры MoveLeft.

2. Описать функцию MoveRight(A,N,K), осуществляющую циклический сдвиг элементов вещественного массива A размера N на K позиций вправо (K<N). Массив A – входной и выходной параметр, N и K – входные параметры. Даны массив размера N и числа K1, K2. С помощью функции MoveRight осуществить сдвиг элементов данного массива на K1 позиций вправо, а затем – сдвиг элементов полученного массива на K2 позиций вправо, выводя элементы преобразованного массива на экран после каждого вызова процедуры MoveRight.

3. Описать функцию Smooth(A,N), заменяющую каждый элемент вещественного массива A размера N на его среднее арифметическое со своими соседями (сглаживание массива). Массив A – входной и выходной параметр, N – входной параметр. С помощью этой функции выполнить пятикратное сглаживание данного массива A размера N, выводя на экран результаты каждого сглаживания.

4. Описать функцию RemoveX(A,N,X), удаляющую элементы, равные числу X, из массива A целых чисел размера N. Массив A и число N являются входными и выходными параметрами, X – входной параметр. С помощью этой функции удалить числа XA, XB, XC из массивов A, B, C размера NA, NB, NC соответственно.

5. Описать функцию DoubleX(A,N,X), дублирующую элементы, равные числу X, в массиве A целых чисел размера N. Массив A и число N

являются входными и выходными параметрами, X – входной параметр. С помощью этой функции продублировать числа XA, XB, XC в массивах A, B, C размера NA, NB, NC соответственно.

6. Дан массив целых чисел из N элементов, отсортированный по неубыванию. Удалить из него дубликаты. Решение оформить в виде функции.

7. Даны массив целых чисел из N элементов и целое число A. Переставить элементы, меньшие A, в начало массива, меняя их с предыдущими элементами. Порядок элементов, меньших A, а также порядок элементов, больших A, не менять. Решение оформить в виде функции.

8. Дан массив целых чисел из N элементов. Сформировать по нему массив, содержащий длины всех серий (подряд идущих одинаковых элементов). Одиночные элементы считать сериями единичной длины. Решение оформить в виде функции.

9. Дан массив целых чисел из N элементов. Из каждой серии удалить один элемент. Серия – подряд идущие одинаковые элементы. Одиночные элементы считать сериями единичной длины. Решение оформить в виде функции.

10. Дан массив целых чисел из N элементов. Удалить все серии, длина которых меньше K. Серия – подряд идущие одинаковые элементы. Одиночные элементы считать сериями единичной длины. Решение оформить в виде функции.

11. Описать функцию `MinStr(S,N)`, находящую минимальное значение в массиве S строк из N элементов. Массив S и число N являются входными параметрами.

12. Описать функции `SortName(A,N)` и `SortCourse(A,N)`, выполняющие сортировку по возрастанию массива A из N структур типа `struct Student{char* Name; int Course;}` по полям Name

и Course соответственно. Массив A является входным и выходным параметром, N – входной параметр.

13. Дан массив целых чисел из N элементов. Оформить функцию Count_If(A,N,f), вычисляющую количество элементов в массиве A, удовлетворяющих данному условию f, передаваемому в качестве параметра (условие должно представлять собой функцию, принимающую параметр целого типа и возвращающую значение логического типа).

14. Оформить функцию заполнения массива целых чисел из N элементов, передавая ей в качестве параметра функцию, задающую алгоритм генерации следующего значения, вида int f(). Для генерации данная функция может запоминать значения, сгенерированные на предыдущем шаге, либо в глобальных переменных, либо в статических локальных переменных.

15. Дан массив целых чисел из N элементов. Оформить функцию Remove_If(), удаляющую из него все элементы, удовлетворяющие условию, передаваемому в качестве параметра.

16. Дан массив целых чисел из N элементов. Оформить функцию для нахождения в нем пары чисел A и B с минимальным значением f(A,B), где f передается в качестве параметра.

17. Даны три массива целых чисел A, B, C. Составить функцию change(f,ar,n), применяющую функцию преобразования целого числа f(x), передаваемую в качестве параметра, к каждому элементу массива ar. С ее помощью удвоить все элементы массива A, заменить отрицательные элементы массива B нулем, в каждом из чисел массива C вычеркнуть последнюю цифру.

18. Реализовать функцию `char *mystrcat(char *p,const char *q)`, добавляющую строку `q` к строке `p`.

19. Описать функцию `TrimRightC(S,C)`, удаляющую в строке `S` конечные символы, совпадающие с символом `C` (`S` является входным и выходным параметром, `C` – входной параметр). Даны символ `C` и пять строк. Используя функцию `TrimRightC`, преобразовать данные строки.

20. Реализовать функцию, удаляющую из строки `S` каждое вхождение подстроки `S1`.

21. Реализовать функцию, заменяющую в строке `S` каждое вхождение подстроки `S1` на подстроку `S2`.

22. Реализовать функцию, возвращающую первое вхождение подстроки `q` в строку `p` (или `0`, если подстрока не найдена):

`char *mystrstr(const char *p, const char *q)`.

23. Реализовать функцию, возвращающую указатель на первое вхождение в строку `p` какого-либо символа из строки `q` (или 0, если совпадений не обнаружено):

`char *mystrpbrk(const char *p, const char *q)`.

24. Реализовать функцию, возвращающую число начальных символов в строке `p`, которые не совпадают ни с одним из символов из строки `q`:

`int mystrspn(const char *p, const char *q)`.

25. Реализовать функцию, возвращающую число начальных символов в строке `p`, которые совпадают с одним из символов из строки `q`:

`int mystrcspn(const char *p, const char *q)`.

МОДУЛЬ 3. КЛАССЫ И ОБЪЕКТЫ

Задачи и цели: Систематизировать знания студентов о принципах описания классов, о способах создания и использования объектов. Ознакомить студентов с приемами и правилами перегрузки операций. Ввести понятие шаблонов классов как механизма обобщенного программирования. Ознакомить студентов с различными способами реализации шаблонных классов по шаблонам. Научить создавать новые классы путем наследования существующих классов. Дать понятие того, как наследование способствует повторному использованию программного обеспечения. Научить использовать виртуальные функции, строить иерархию классов.

Требуемый начальный уровень подготовки. Знакомство с основами программирования, алгоритмизацией и процедурным программированием на языке C++. Представление об основных принципах объектно-ориентированного подхода в программировании.

3.1. Основы создания классов

Попытаемся ответить на вопрос: в чем же состоит принципиальное отличие языка C++ от языка C? Сошлемся при этом на одного из специалистов по языку C++ Брюса Эккеля: «Включение функций в структуры составляет подлинную суть того, что язык C++ добавил в C». При внесении функций в структуры к характеристикам (как у структур в C) добавляется поведение. Так возникает концепция объекта.

Для объявления класса можно использовать либо ключевое слово `struct`, либо ключевое слово `class`. Рекомендуется использовать слово `class`. Различие проявляется в способах доступа к полям и методам класса по умолчанию (табл. 6).

Таблица 6

Различие в доступе по умолчанию

Описание	`class <имя класса>{ <поля и методы> };`	`struct <имя класса>{ <поля и методы> };`
Способ доступа по умолчанию	`private`	`public`

Существует три вида спецификаторов доступа:

`private` (закрытый) – доступен только для членов класса;

`protected` (защищенный) – доступен для членов класса и их наследников;

`public` (открытый) – доступен для всех.

Эти спецификаторы определяют уровень доступа для всех объявлений, следующих за ними. После любого спецификатора должно стоять двоеточие.

Следующие два объявления эквивалентны:

```
struct A{
private:
  int i, j, k;
public:
  int f();
  void g();
};

int A::f() {
  return i+j+k;
}

void A::g() {
  i=j=k=0;
}
```

```
class B{
  int i, j, k;
public:
  int f();
  void g();
};

int B::f() {
  return i+j+k;
}

void B::g() {
  i=j=k=0;
}
```

```
int main(){
  A a;
  B b;
  a.f(); a.g();
  b.f(); b.g();
}
```

Приняты два варианта объявления класса:

- ✓ сначала располагаются поля, потом – методы;
- ✓ сначала располагаются методы, потом – поля.

Пример 39. Реализовать класс для описания геометрической фигуры «круг».

```
//circle.h
#ifndef CIRCLE_H
#define CIRCLE_H
class circle {
private:
  double x,y,r;
public:
  circle();
  circle(double x1, double y1, double r1);
  double s();
  void set_r(double r1);
  double get_r();
  bool equal_1(circle a);
  bool operator ==(circle a);
};
#endif

//circle.cpp
#include <math>
#include "circle.h"
circle::circle(){
  x=10;
  y=10;
  r=10;
}
circle::circle(double x1, double y1, double r1){
  x=x1;
  y=y1;
  r=r1;
}
double circle::s(){
  return 3.14*r*r;
}
void circle::set_r(double r1){
  if (r1<0)
    r=0;
  else
    r=r1;
}
```

```
double circle::get_r(){
  return r;
}
bool circle::equal_1(circle a){
  const double eps=0.0001;
  if (abs(x-a.x)<eps && abs(y-a.y) <eps && abs(r-a.r)<eps)
    return true;
  else
    return false;
}
bool circle::operator ==(circle a){
  return equal_1(a);
}

//главный файл – main.cpp
#include "circle.h"
#include <iostream>
using namespace std;

int main(){
  circle a,b(0,0,1);
  cout<<a.s()<<endl<<b.s()<<endl;
  cout<<a.equal_1(b)<<endl;
  cout<<a.operator==(b)<<endl;
  cout<<(a==b)<<endl;
  return 0;
}
```

Описание класса размещено в заголовочном файле. В описание класса включены поля и заголовки методов.

☺ В C++ рекомендуется реализацию методов располагать в большинстве случаев вне класса, но допустимо и внутри описания класса. Описание класса рекомендуется располагать в заголовочном файле.

Поля класса `x`, `y`, `r` объявлены закрытыми. Все методы этого класса – открытые. Среди методов присутствуют:

- ✓ два конструктора

```
circle();
circle(double x1, double y1, double r1);
```

- ✓ функция вычисления площади

```
double s();
```

- ✓ пара методов для установки доступа к закрытому полю класса

```
void set_r(double r1);
double get_r();
```

- ✓ два метода для сравнения двух объектов на равенство – один реализован как функция, другой как перегруженная операция

```
bool equal_1(circle a);
bool operator ==(circle a);
```

☺ В основном поля рекомендуется делать закрытыми, интерфейсные методы – открытыми, а служебные методы – закрытыми.

Реализация методов расположена в отдельном cpp-файле. При их реализации вне описания классов имена методов должны быть уточненными именем класса. Для этого используется операция разрешения области видимости :: :

```
double circle::s(){
  return 3.14*r*r;
}
```

Текст файла main.cpp начинается с создания объектов a и b. Для создания объекта a вызывается конструктор по умолчанию, для объекта b – конструктор с параметрами:

```
circle a,b(0,0,1);
```

Все методы класса `circle` являются методами объектов (экземпляров класса). Они вызываются через точку a.s( ):

```
cout<<a.s()<<endl<<b.s()<<endl;
```

3.2. Конструкторы и деструкторы

Чтобы обеспечить должную инициализацию каждого объекта, разработчик класса должен включить в него специальную функцию, которая называется *конструктором*. Имя конструктора должно совпадать с именем

класса. Конструктор предназначен для создания объекта в памяти и инициализации полей начальными значениями.

В примере 39 реализованы два конструктора – без параметров и с параметрами:

```
circle();
//без параметров - конструктор по умолчанию
circle(double x1, double y1, double r1);
//конструктор с параметрами
```

Конструктор без параметров называется *конструктором по умолчанию*.

☺ Если в дальнейшем предполагается размещать объекты класса в массиве или в любом другом контейнере, необходимо объявлять конструктор по умолчанию.

В случае, если явно не описан ни один конструктор, компилятор автоматически сгенерирует конструктор по умолчанию. Поведение конструктора по умолчанию будет таким же, как если бы он был объявлен явно без списка инициализации и с пустым телом.

Когда программа достигает точки выполнения, в которой определяется объект

```
circle a(3,1,4.2),b(0,0,1);
```

происходит *автоматический вызов конструктора*. Конструктор, как и все остальные функции классов, получает в скрытом аргументе `this` адрес того объекта, для которого он вызывается.

Ключевым словом `this` обозначается указатель на объект в функциях членах класса и может использоваться только в нестатических функциях-членах.

Синтаксис *деструктора* в целом схож с синтаксисом конструктора: имя функции тоже определяется именем класса. Но чтобы деструктор от-

личался от конструктора, его имя начинается с префикса ~ (тильда). Кроме того, деструктор всегда вызывается без аргументов.

Деструктор автоматически вызывается компилятором при выходе объекта из области видимости. При этом очищается память, занимаемая объектом. Явное описание деструктора необходимо в тех случаях, когда класс использует динамическую память для размещения полей объекта. В случае отсутствия явного задания деструктора, вызывается деструктор по умолчанию.

 Конструкторы и деструкторы обладают одной уникальной особенностью: они не имеют возвращаемого значения. В этом они принципиально отличаются от функций, возвращающих пустое значение (значение типа `void`).

Пример 40. Реализовать класс – динамический массив целых чисел. Перегрузить операции: + для двух массивов одинакового размера; + для массива и целого числа; += для двух массивов одинакового размера; присваивания; обращение к элементу по индексу; вывода в поток.

```
//classArray.h
#ifndef CLASSARRAY_H
#define CLASSARRAY_H

#include <iostream>
using namespace std;

class Array{
private:
  int n;
  int *A;
public:
  Array();     //конструктор по умолчанию
  Array(int _n, int x=0);
//конструктор с параметром по умолчанию
  Array(const Array &B);      //конструктор копии
  int max( );
//Функция для нахождения максимального элемента в массиве
  Array operator + (const Array &B);
```

```
  Array operator += (const Array &B);
  Array operator + (const int x);
  Array &operator = (const Array &B);
  int& operator [] (int i);
  ~Array ( );
  friend ostream & operator<<(ostream &out, const Array &B);
};
#endif

//classArray.cpp
#include "classArray.h"
Array::Array( ){   //конструктор по умолчанию
  n=10;
  A=new int[n];
  for (int i=0; i<n; i++)
    A[i]=0;
}

Array::Array(int _n, int x){
  n=_n;
  A=new int[n];
  for (int i=0; i<n; i++)
    A[i]=x;
}

Array::Array (const Array &B){    //конструктор копии
  n=B.n;
  A=new int[n];
  for (int i=0; i<n; i++)
    A[i]=B.A[i];
}

int Array::max( ){
//функция для нахождения максимального элемента в массиве
  int maxi = A[0];
  for (int i=1; i<n; i++)
    if (A[i]>maxi)
      maxi=A[i];
  return maxi;
}

Array Array::operator + (const Array &B){
  Array C(n) ; //можно: Array C(n,0);
  for (int i=0; i<n; i++)
    C.A[i]=A[i]+B.A[i];
  return C;
}
```

```
Array Array::operator += (const Array &B){
  for (int i=0; i<n; i++)
    A[i]=A[i]+B.A[i];
  return *this;
}
Array Array::operator + (const int x){
  Array C(n) ; //можно: Array C(n,0);
  for (int i=0; i<n; i++)
    C.A[i]=A[i]+x;
  return C;
}
Array &Array::operator = (const Array &B){
  if (this != &B){
    delete[] A;
    n=B.n;
    A=new int[n];
    for (int i=0; i<n; i++)
      A[i]=B.A[i];
  }
  return *this;
}
int& Array::operator [] (int i){
  return A[i];
}
Array::~Array( ){   //деструктор
  delete[] A;
}

ostream & operator << (ostream & out, const Array &B){
  for (int i=0; i<B.n; i++)
    out << B.A[i] << ' ';
  out << endl;
  return out;
}

//main.cpp
#include "classArray.h"
int main(){
  Array Q(10,5);
  Array W,E;
  Array R(Q);
  cout<<Q<<R;
  W=Q+5;
  cout<<W;
  E=W+R;
  cout<<E;
  return 0;
}
```

Для размещения поля A объекта Array используется динамическая память. Для классов, размещающих данные в динамической памяти, необходим *конструктор копии* (или *копирующий конструктор*). Он создает копию объекта, передаваемого в конструктор в качестве параметра:

```
Array::Array (const Array &B){
  //конструктор копии
  n=B.n;
  A=new int[n];
  for (int i=0; i<n; i++)
    A[i]=B.A[i];
}
```

☺ В случае использования динамической памяти в реализации класса – настоятельно рекомендуется реализовать конструкторы и деструктор. Среди конструкторов обязательно должен присутствовать конструктор копии.

Копирующие конструкторы настолько важны, что компилятор автоматически генерирует копирующий конструктор, если этого не делает программист. Такой автоматически создаваемый конструктор копии является конструктором с побитовым копированием. В этом случае будет создана копия ссылки на объект, т. е. две ссылки будут указывать на одно и то же место в памяти.

☠ Для динамических структур данных отсутствие конструктора копии является грубой ошибкой.

Кроме явного вызова конструктора копии при создании объекта, конструктор копии вызывается каждый раз в случае, когда функция возвращает объект.

3.3. Перегрузка операций

Перегружать операции можно только для определенных пользователем типов, т. е. для классов (операции для стандартных типов перегружать нельзя).

Определение перегруженной операции очень похоже на определение функции с именем `operator@`, где `@` – идентификатор перегружаемой операции.

В примере 39 для класса `circle` была перегружена операция сравнения на равенство (==):

```
bool circle::operator ==(circle a){
  return x==a.x && y==a.y && r==a.r;
}
```

Вызов перегруженной операции можно выполнять двумя способами:

- ✓ используя синтаксис вызова функции `a.operator==(b)`, например:
  `cout<<a.operator==(b)<<endl;`
- ✓ используя синтаксис операции `a==b`, например:
  `cout<<(a==b)<<endl;`

Попробовать убрать скобки в примере 39 в операторе вывода в поток `cout<<(a==b)<<endl;`

Объяснить, почему они необходимы.

Количество аргументов в списке перегруженной операции зависит от двух факторов:

- ✓ категории операции – унарная (один аргумент) или бинарная (два аргумента);
- ✓ способа определения операции – в виде глобальной функции (один аргумент для унарной, два – для бинарных операций) или функции класса

(для унарных операций аргументы отсутствуют, для бинарных операций – один аргумент).

Демонстрация перегрузки операции сложения двух объектов Array:

```
Array Array::operator + (const Array &B){
Array C(n);    //можно: Array C(n,0);
  for (int i=0; i<n; i++)
    C.A[i]=A[i]+B.A[i];
  return C;
}
```

Напомним, что по условию требовалось перегрузить операцию для массивов одинаковой размерности.

Перечислим некоторые особенности перегрузки операций.

Важно, чтобы перегруженная операция доступа к элементу массива по индексу [ ] возвращала ссылку на элемент массива.

```
int& Array::operator [] (int i){
  return A[i];
}
```

Присваивание по умолчанию реализуется побайтным копированием. Поэтому для динамических объектов (как в нашем примере) операцию присваивания нужно перегрузить. При этом следует сначала проверить, не происходит ли самоприсваивание (A=A).

```
Array Array::operator = (const Array &B){
  if (this != &B){
    delete[] A;
    n=B.n;
    A=new int[n];
    for (int i=0; i<n; i++)
      A[i]=B.A[i];
  }
return *this;
}
```

Функция operator= должна быть обязательно функцией класса. В определении функции operator= необходимо скопировать всю необходимую информацию из правостороннего объекта в левосторонний.

В случае, если для класса перегружены операции + и =, то реализация перегруженной операции += может быть выполнена с их использованием.

Например, реализация перегруженной операции `+=` для класса `Array` из примера 40

```
Array Array::operator += (const Array &B){
  for (int i=0; i<n; i++)
    A[i]=A[i]+B.A[i];
  return *this;
}
```

может быть изменена следующим образом:

```
Array Array::operator += (const Array &B){
  *this=(*this)+B;
  return *this;
}
```

 Выполнить эти изменения в примере. Проверить работоспособность программы.

Хотя C++ позволяет перегружать почти все операции, доступные в C, возможности перегрузки ограничены. В частности, отсутствует возможность изменения приоритета или количества аргументов у операций.

3.4. Дружественные функции

Иногда бывает нужно, чтобы левосторонний операнд был объектом другого класса. Например, для часто перегружаемых операций потокового ввода-вывода левосторонним операндом должен быть объект-поток. Следовательно, перегрузку такой операции нужно организовывать в виде внешней функции. А внешним функциям запрещен доступ напрямую к полям класса, объявленным как `private`.

Способ решения проблемы – объявить такую операцию дружественной для класса. Дружественными по отношению к рассматриваемому клас-

су могут быть внешние функции, или функции другого класса, или другой класс.

Для того чтобы внешнюю функцию сделать дружественной классу, нужно внести в интерфейс класса заголовок функции, добавив впереди ключевое слово `friend`.

В примере 40 для класса `Array` реализована перегрузка операции << вывода в поток:

```
ostream & operator << (ostream & out, const Array &B){
  for (int i=0; i<B.n; i++)
    out << B.A[i] << ' ';
  out << endl;
  return out;
}
```

После выполнения всех действий с потоком ввода или вывода операция возвращает ссылку на поток, что позволяет использовать результат в более сложных выражениях.

Дружественность функции `operator<<` по отношению к классу `Array` дает право функции напрямую обращаться к закрытым членам класса.

Самостоятельно реализовать для класса `Array` операцию ввода из потока.

На функции, объявленные дружественными, не распространяются действия модификаторов (`public`, `private`, `protected`). В связи с этим необходимо придерживаться следующего правила.

☺ Объявление функций дружественными рекомендуется располагать либо в самом начале, либо в самом конце описания класса.

Пример 41. Реализовать класс – `Time` («Время»).

```
//classTime.h
#ifndef CLASSTIME_H
```

```
#define CLASSTIME_H

class Time{
  private:
  int sec,min,hour;
  void Increment(); // Функция-утилита
  public:
  Time(int _hour=0,int _min=0,int _sec=0);
  // Конструктор с параметрами по умолчанию
  int GetHour() const;
  Time operator++();
  //Префиксная форма инкремента
  Time operator++(int);
  //Постфиксная форма инкремента
};

#endif
```

Реализация методов класса `Time` будет располагаться в файле classTime.cpp. В данном классе используется конструктор с параметрами по умолчанию.

☠ Нельзя в классе объявлять одновременно конструктор по умолчанию и конструктор со всеми параметрами по умолчанию. Это приведет к ошибке. Компилятор не сможет определить, какой из конструкторов нужно вызвать.

Реализация конструктора использует инициализаторы элементов. В связи с этим тело конструктора пусто.

```
Time::Time(int _hour,int _min,int _sec):hour(_hour),min(_min),sec(_sec)
{}
```

В общем случае такой способ не является обязательным, но в некоторых ситуациях без него не обойтись. Например, константные элементы класса должны получать начальные значения с помощью инициализаторов элементов. Присваивания в теле конструктора в этом случае недопустимы.

Член-функцию класса можно объявить константной, если она не должна изменять значение полей объекта. Например:

```
int Time::GetHour() const{
  return hour;
}
```

☺ Рекомендуется все функции для доступа к полям на чтение (`getXX`-методы) объявлять константными.

Если объявить константный объект, то такой объект может использовать только константные член-функции. Например, можно объявить константу для некоторого эталона времени

```
const Time X(12,0,0);
```

Для объекта X можно использовать только метод `GetHour()`.

 Уберите модификатор `const` из метода `GetHour()` и попробуйте использовать его для константного объекта.

3.5. Перегрузка префиксной и постфиксной операций инкремента

При перегрузке операции инкремента (декремента) для получения возможности использования и префиксной, и постфиксной форм, каждая из этих двух перегруженных функций-операций должна иметь разную сигнатуру. Это даст возможность компилятору определить, какая версия инкремента имеется в виду в каждом конкретном случае.

Допустим, мы объявили объект `t` типа `Time`:

```
Time T;
```

Когда компилятор встречает выражение с префиксным инкрементом `++t`, генерируется вызов функции-элемента `t.operator++()`, объявление которой должно иметь вид:

```
Time operator++();
```

По соглашению, принятому в C++, когда компилятор встречает выражение постфиксной формы инкремента `t++`, он генерирует вызов функции `t.operator++(0)`, объявлением которой является:

```
Time operator++(int);
```

Нуль (0) в генерируемом вызове функции является чисто формальным значением, введенным для того, чтобы сделать список аргументов функции `operator++`, используемой для постфиксной формы инкремента, отличным от списка аргументов функции `operator++`, используемой для префиксной формы инкремента. Заметьте, что формальный параметр является фиктивным, и поэтому не имеет имени.

```
Time Time::operator++(){
  Increment();
  return *this;
}
// фиктивный целый параметр не имеет имени
Time Time::operator++(int){
  Time temp(*this);
  Increment();
  return temp;
}
// Функция-утилита увеличения времени на 1 секунду
void Time:: Increment(){
  sec++;
  if (sec==60){
    sec=0;
    min++;
  }
  if (min==60){
    min=0;
    hour++;
  }
}
```

Перегруженная операция префиксного инкремента возвращает копию текущего объекта с измененным временем. Это происходит потому, что текущий объект `*this` возвращается как объект класса `Time`, что активизирует конструктор копии.

Чтобы эмулировать действие постфиксного инкремента, необходимо вернуть неизмененную копию объекта `Time`. При входе в `operator++` текущий объект `*this` сохраняется во временном объекте `temp`. Затем вызывается `Increment()`, чтобы инкрементировать объект. В итоге, возвращается неизмененная копия объекта `temp`. Отметим, что эта функция не может вернуть ссылку на объект класса `Time`, потому что значение, которое надо вернуть, сохраняется в локальной переменной в определении функции. Локальные переменные уничтожаются, когда функция, в которой они объявлены, завершена. Таким образом, объявление типа, возвращаемого функцией, как `Time&`, привело бы к ссылке на объект, который после возвращения больше не существует.

Возвращение ссылки на локальную переменную является типичной ошибкой, которую трудно найти.

Пример функции `main` для класса `Time`:

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

void main (){
  Time t1;
  Time t2(11,59,59);
  cout<<t1.GetHour()<<endl;
  t1++;
  cout<<t1.GetHour()<<endl;
  cout<<t2.GetHour()<<endl;
  t2++;
  cout<<t2.GetHour()<<endl;
}
```

3.6. Реализация преобразования типов

В языке C++ определены операции преобразования между встроенными типами данных. А преобразования между встроенными типами и типами, определенными пользователями, или только между типами, определенными пользователями, требуют определения. Для этого используются

или конструкторы преобразований, или операции преобразований. Выбор механизма реализации преобразования зависит от типов данных, между которыми производится преобразование.

Конструктор преобразования (конструктор с единственным аргументом) может быть использован для преобразования объектов разных типов (включая встроенные типы) в объекты данного класса.

Операция преобразования (называемая также *операцией приведения*) может быть использована для преобразования объекта одного класса в объект другого класса или в объект встроенного типа. Такая операция преобразования должна быть нестатической функцией-членом. Операция преобразования этого вида не может быть дружественной функцией.

Пример 42. Реализовать класс «Строка». Предусмотреть конструктор для преобразования строк в стиле С `char*` в объекты класса «Строка» и операцию приведения для преобразования класса «Строка» в строку в стиле С `char*`.

```
//STR.H
#ifndef STR_H
#define STR_H

#include <iostream>
using namespace std;

class Str{
  friend ostream & operator << (ostream &, const Str &);
  friend istream & operator >> (istream &, Str &);
public:
  Str(const char * = "");     //конструктор преобразования
  Str(const Str &);           //конструктор копии
  ~Str();                     //деструктор
  Str & operator= (const Str &);   //присваивание
  Str & operator+= (const Str &); //сцепление (конкатенация)
  bool operator!() const;          //проверка строки на пустоту
  bool operator== (const Str &) const;//проверка на равенство
  char & operator[] (int);    //получение ссылки на символ
  Str operator() (int, int); //получение подстроки
  int getLength() const;      //определение длины строки
```

```
  operator char*() const;     //операция приведения к char*
private:
  char * sPtr;      //указатель на начало строки
  int length;       //длина строки
};
#endif

//STR.cpp
// Определение некоторых функций элементов класса Str
#include <iostream>
#include <cstring>
#include <cassert>
#include <iomanip>
#include "str.h"

// конструктор преобразования: преобразовывает char* в Str
Str::Str(const char* s){
  length=(int)strlen(s);
  sPtr=new char[length + 1];
  assert(sPtr != 0);     //завершение, если память не выделена
  strcpy(sPtr,s);
}

// конструктор копии
Str::Str(const Str & copy){
  length=copy.length;
  sPtr=new char[length + 1];
  assert(sPtr != 0);     //завершение, если память не выделена
  strcpy(sPtr,copy.sPtr);
}

// деструктор
Str::~Str(){
  delete [] sPtr;
}

//операция присваивания
Str & Str::operator =(const Str & right){
  if (&right != this){
    delete [] sPtr;
    length = right.length;
    sPtr = new char[length + 1];
    assert(sPtr != 0);
    strcpy(sPtr, right.sPtr);
  }
  return *this;
}
```

```
//сцепление (конкатенация)
Str & Str::operator +=(const Str & right) {
  char * tempPtr = sPtr;
  length += right.length;
  sPtr = new char [length+1];
  assert(sPtr!=0);
  strcpy(sPtr, tempPtr);
  strcat(sPtr, right.sPtr);
  delete [] tempPtr;
  return *this;
}

//проверка строки на пустоту
bool Str::operator !() const {
  return length ==0;
}

//проверка двух строк на равенство
bool Str::operator == (const Str & right) const {
  return strcmp(sPtr, right.sPtr) == 0;
}

//получение ссылки на символ
char & Str::operator[] (int ind){
  //проверка, не находится ли индекс вне диапазона
  assert(ind >= 0 && ind < length);
  return sPtr[ind];     //создание L-величины
}

//получение подстроки, начинающейся с заданного индекса,
//заданной длины
Str Str::operator() (int ind, int sublength){
  //проверка, что находится индекс в диапазоне и длина подстроки >=0
  assert(ind >= 0 && ind < length && sublength>=0);
  Str sub;
  //определение длины подстроки
  if ((sublength==0) || (ind+sublength>length))
    sub.length=length-ind+1;
  else
    sub.length=sublength+1;
  //выделение памяти для подстроки
  sub.sPtr=new char[sub.length];
  assert(sub.sPtr!=0);  //подтверждение выделения памяти
  //копирование подстроки в новый Str
  strncpy(sub.sPtr,&sPtr[ind],sub.length);
  sub.sPtr[sub.length]='\0'; //завершение новой строки sPtr
  return sub;
}
```

```
//определение длины строки
int Str::getLength() const{
  return length;
}

//операция приведения к char*
Str::operator char*() const{
  char *c=new char[length];
  strcpy(c,sPtr);
  return c;
}

//перегруженная операция вывода данных
ostream &operator << (ostream & output, const Str & s){
  output<<s.sPtr;
  //возврат ссылки на поток для возможности многократного
  //последовательного использования операции <<
  return output;
}

//перегруженная операция ввода данных
istream &operator >> (istream & input, Str & s){
  char temp[100];   //буфер для хранения входных данных;
  input>>setw(100)>>temp;
  s=temp;
  return input;
}

// String.cpp
#include "Str.h"
int main(){
  char *s=new char[100];
  cin>>s;
  Str s1("Hello"),s2(s), s3;
  cin>>s3;
  cout<<s1<<endl;
  cout<<s2<<endl;
  cout<<s3<<endl;
  s=s3;
  cout<<s<<endl;
  cin>>s;
  return 0;
}
```

Объявление перегруженной функции-операции приведения типа из определенного пользователем типа `Str` в тип `char*`:

```
operator char*() const;
```

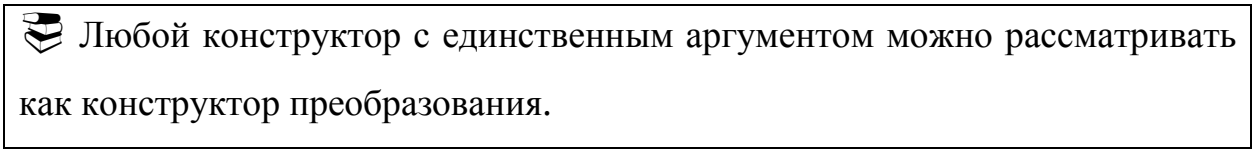
Перегруженная функция-операция приведения не указывает тип возвращаемой величины, потому что им является тип, к которому преобразован объект.

Если `s` – объект класса `Str`, то когда компилятор встречает выражение `(char*)s`, он порождает вызов `s.operator char*()`. Для этого вызова операнд `s` – это объект класса `Str`, для которого была активизирована функция-элемент `operator char*`.

Конструктор преобразования получает аргумент `char*` и создает объект `Str`

```
Str(const char * = "");
```

Конструктор преобразования преобразует соответствующую строку в объект `Str`, который затем присваивается создаваемому объекту `Str`.


Любой конструктор с единственным аргументом можно рассматривать как конструктор преобразования.

Одной из особенностей операций приведения и конструкторов преобразований является то, что при необходимости компилятор может вызывать эти функции автоматически для создания временных объектов.

Если в программе в том месте, где ожидается `char*`, появился объект `s` определенного пользователем типа `Str`, то в этом случае компилятор для преобразования объекта в `char*` вызывает перегруженную функцию-операцию приведения `operator char*` и использует в выражении результирующий `char*`. Например:

```
cout << s;
```

Поэтому операция приведения для класса `Str` позволяет не перегружать операцию << (поместить в поток), предназначенную для вывода `Str` с использованием `cout`.

Наличие конструктора преобразования означает, что нет необходимости применять перегруженную операцию специально для присваивания строк символов объектам `Str`. Компилятор автоматически активизирует конструктор преобразования для создания временного объекта `Str`, содержащего строку символов. Затем активизируется перегруженная операция присваивания, чтобы присвоить временный объект другому объекту `Str`.

В тексте программы активно используется не упоминавшийся ранее идентификатор `assert`. Рассмотрим его подробнее. Прежде всего отметим, что `assert` является макросом.

> 🕮 *Макросы* (макрокоманды) в мире программирования на языках C и C++ представляют собой особый тип команд препроцессора.
>
> *Препроцессор* – специальная программа, которая обрабатывает весь исходный код программы перед тем, как он будет передан компилятору.

Обычно макрокоманды используются для повышения быстродействия кода, а также для быстрого освобождения памяти, что может быть полезно в системах реального времени, в мобильных и иных устройствах.

Макросы бывают двух видов – макросы-объекты и макросы-функции.

Общий вид определения макроса таков:

```
#define_[идентификатор]_[выражение]
```

Макросы уже были применены при рассмотрении принципа единственности определения заголовочных файлов.

Для использования макроса `assert` необходимо подключить заголовочный файл <cassert>. Макрос `assert` используется для включения в программу диагностических сообщений.

```
void assert(int выражение)
```

Например,

```
assert(sPtr != 0);  //завершение, если память не выделена
```

или

```
assert(ind >= 0 && ind < length && sublength>=0);
//проверка, что индекс находится в диапазоне и длина подстроки >=0
```

Если значение выражения есть нуль, то assert(выражение) напечатает сообщение следующего вида:

```
Assertion failed: выражение, file имя файла, line nnn
```

После чего будет вызвана функция abort, которая завершит вычисления.

3.7. Коллекции и итераторы

Коллекция – объект, содержащий в себе, тем или иным образом, набор значений одного или различных типов и позволяющий обращаться к этим значениям.

Назначение коллекции – служить хранилищем объектов и обеспечивать доступ к ним. Обычно коллекции используются для хранения групп однотипных объектов, подлежащих стереотипной обработке. Для обращения к конкретному элементу коллекции могут использоваться различные методы, в зависимости от ее логической организации. Реализация может допускать выполнение отдельных операций над коллекциями в целом. Наличие операций над коллекциями во многих случаях может существенно упростить программирование.

Итератор – объект, позволяющий программисту перебирать все элементы коллекции без учета особенностей ее реализации.

В простейшем случае итератором в низкоуровневых языках является указатель. Операцию индексирования можно считать примитивной формой итератора. Необходимо отметить, что счетчик цикла иногда называют ите-

ратором цикла. Тем не менее счетчик цикла обеспечивает только перебор элементов, но не доступ к элементу.

Использование итераторов в обобщенном программировании позволяет реализовать универсальные алгоритмы работы с контейнерами. Примерами коллекций (контейнеров), по которым может перемещаться итератор, могут быть: список, очередь, множество, ассоциативный массив и др.

Итератор похож на указатель своими основными операциями: указание одного отдельного элемента в коллекции объектов (доступ к элементу) и изменение своего значения так, чтобы указывать на следующий элемент (перебор элементов). Также должен быть определен способ создания итератора, указывающего на первый элемент коллекции, и способ узнать, достигнут ли конец коллекции. В зависимости от используемого языка и цели, итераторы могут поддерживать дополнительные операции или определять различные варианты поведения.

Главное предназначение итераторов заключается в предоставлении возможности пользователю обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера от пользователя. Это позволяет контейнеру хранить элементы любым способом при допустимости работы пользователя с ним как с простой последовательностью или списком. Проектирование класса итератора тесно связано с соответствующим классом контейнера. Обычно контейнер предоставляет методы создания итераторов.

Существует множество разновидностей итераторов, различающихся своим поведением, включая: однонаправленные, обратные (реверсные) и двунаправленные итераторы; итераторы ввода и вывода; константные итераторы (защищающие контейнер или его элементы от изменения).

В тех случаях, когда коллекция представляет собой структуру, основанную на указателях, итератор может быть реализован в виде дополнительного поля – текущего указателя. В этом случае следует заботиться о его состоянии при любых модификациях коллекции, даже если его не придется использовать. Кроме того, такой встроенный итератор обычно только один, а могут возникнуть задачи, в которых потребуется наличие двух, трех или даже большего количества итераторов.

Итераторы-объекты имеют преимущество по сравнению со встроенным текущим указателем, так как позволяют создавать для одной коллекции любое количество итераторов, действующих независимо. Таким образом, итератор позволяет вынести рабочий указатель за пределы коллекции, но при этом дает возможность перемещаться по объектам, содержащимся в коллекции, аналогично тому, как текущий указатель позволяет перемещаться внутри коллекции.

Обычно итераторы реализуются через дружественные классы.

Как правило, итератор имеет операцию доступа к элементу коллекции по ссылке. Такая операция может быть реализована путем перегрузки операции разыменования *. Для итераторов предусматриваются также операции перемещения вперед и назад по коллекции объектов. Чаще всего для этого перегружаются операции `++` и `--`. Кроме того, операции `==` и `!=` обычно перегружаются для проверки равенства итераторов. Для того чтобы коллекция могла использовать итератор, он должен объявить класс коллекции дружественным.

В классе коллекции для использования итератора обычно создаются две функции:

`iterator begin()` – инициализация итератора ссылкой на первый элемент коллекции;

`iterator end()` – значение, сообщающее, что итератор достиг конца коллекции.

Кроме того, в классе коллекции имеется доступ ко всем приватным полям и методам итератора, поскольку он является дружественным классом к классу итератора. Поэтому через итератор могут быть реализованы другие операции над коллекцией, например, вставка элемента после позиции итератора или удаление элемента в позиции итератора. При реализации таких операций очень важно оговорить правила поведения итератора после выполнения операции.

Пример 43. Реализовать итератор для линейного односвязного списка.

Класс список `list` предполагается уже описанным, список содержит элементы, являющиеся структурой `el`:

```
struct el {
  int item;
  el* next;
};
```

Вот как выглядит описание класса итератора для списка:

```
class listIterator {
  private:
    const list *collection;
    el *cur;
  public:
    listIterator(const list *s, el *e):collection(s), cur(e){}
    const int &operator *(){
      return cur->item;
    }
    listIterator operator++(); //префиксный ++
    int operator == (const listIterator &ri) const;
    int operator != (const listIterator &ri) const;
    friend class list;
};
```

В закрытых полях класса хранится указатель на коллекцию, с которой работает итератор и указатель на текущее положение итератора в коллекции. Конструктор с параметрами позволяет инициализировать эти поля.

Структуры являются открытыми классами с открытым доступом, поэтому возможно обращение к полям структуры, представляющей элемент списка напрямую. Если бы элемент списка был представлен классом с закрытыми полями, для доступа пришлось бы использовать соответствующие методы или класс итератора нужно было бы объявить дружественным классу элемента списка.

Поскольку рассматривается итератор для линейного односвязного списка, для него имеет смысл движение по списку только в одном направлении, поэтому операция «--» не определена.

Реализация методов итератора:

```
listIterator listIterator:: operator++() {
  cur = cur->next;
  return *this;
}
int listIterator:: operator==(const listIterator &ri) const {
  return ((collection == ri.collection) && (cur == ri.cur));
}
int listIterator:: operator!=(const listIterator &ri) const {
  return !(*this==ri);
}
```

Теперь в класс, представляющий список, добавим методы для работы с итераторами:

```
class list{
  private:
    el* head;
    el* tail;
  public:
    listIterator begin() const;
    listIterator end() const;
    listIterator next (listIterator it) throw (IterException);
};
```

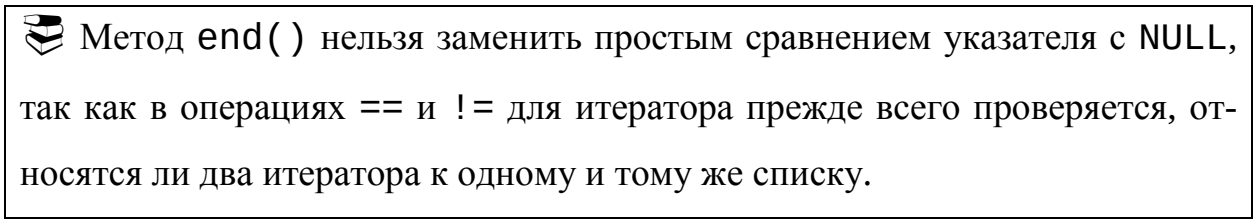
Инициализация итератора ссылкой на первый элемент списка осуществляется следующим методом:

```
listIterator list::begin() const {
  return listIterator (this,head);
}
```

В качестве параметров конструктору итератора передаются указатель на объект-список, для которого будет создан итератор, и указатель на голову списка (динамической структуры).

Метод `listIterator list::end() const` возвращает значение, которое является признаком того, что итератор достиг конца списка:

```
listIterator list::end() const{
  listIterator iter(this,NULL);
  return iter;
}
```


Метод `end()` нельзя заменить простым сравнением указателя с `NULL`, так как в операциях `==` и `!=` для итератора прежде всего проверяется, относятся ли два итератора к одному и тому же списку.

Реализация метода `next(listIterator it)` демонстрирует, как организовать перебор элементов коллекции с помощью итератора:

```
listIterator list::next(listIterator it) throw (IterException)
{
  if (it.collection != this)
    throw IterException();
  if (it == end())
    throw IterException();
  return (++it);
}
```

Если имеется несколько списков и несколько итераторов, может возникнуть ошибка использования итератора не с тем списком, для которого он создан:

```
  if (it.collection != this)
    throw IterException();
```

Перемещение итератора допустимо до тех пор, пока итератор не станет равен признаку того, что список закончился. Если при использовании метода `next()` не предусмотреть такую проверку, то возникает ошибка:

```
  if (it == end())
    throw IterException();
```

Листинг примера, демонстрирующего использование итератора для линейного односвязного списка:

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

class IterException
{};
struct el {
  int item;
  el* next;
};

//предварительное объявление класса итератора
class listIterator;

class list {
  private:
    el* head;
    el* tail;
  public:
    list ();
    ~list ();
    void operator+=(int x);
    listIterator begin() const;
    listIterator end() const;
    listIterator next (listIterator it) throw (IterException);
};

class listIterator {
  private:
    const list *collection;
    el   *cur;
  public:
    listIterator (const list *s, el *e):collection(s), cur(e){}
    const int &operator *() {
      return cur -> item;
    }
    listIterator operator++(); // префиксный ++
    int operator == (const listIterator &ri) const;
    int operator != (const listIterator &ri) const;
    friend class list;
};

listIterator listIterator:: operator++() {
  cur = cur->next;
  return *this;
}
```

```
int listIterator:: operator==(const listIterator &ri) const{
  return ((collection == ri.collection) && (cur == ri.cur));
}

int listIterator:: operator!=(const listIterator &ri) const {
  return ! (*this==ri);
}

list::list () {
  head=0;
  tail=0;
}
list::~list () {
  el *q;
  while (head!=0){
    q=head;
    head=head->next;
    delete q;
  }
  tail=0;
}

listIterator list::begin() const {
  listIterator iter(this,head);
  return iter;
}

listIterator list::end() const {
  listIterator iter(this,NULL);
  return iter;
}

listIterator list::next(listIterator it) throw (IterException) {
  if (it.collection != this)
    throw IterException();
  if (it == end())
    throw IterException();
  return (++it);
}

void list::operator+=(int x){
  el* q=new el;
  q->item=x;
  q->next=0;
  if (head!=0) {
    tail->next=q;
    tail=q;
  }
```

```
  else {
    head=q;
    tail=q;
  }
}

int main() {
  list s1;
  int n,a,i;
  cout<<"count=\n";
  cin>>n;
  cout<<"elements=\n";
  cout<<"1>\n";
  for( i=0;i<n; i++) {
    cin>>a;
    s1+=a;
  }
  listIterator it = s1.begin();
  cout<<" list \n";
  try {
    while (it!= s1.end()){
      cout << *it<<" ";
      it=s1.next(it); // ++it;
    }
  }
  catch (IterException){
  //обработка исключения IterException
  }
  return 0;
}
```

В этом примере использовано предварительное описание класса `listIterator`.

```
class listIterator;
```

Это связано с рекурсией в определении классов коллекции `list` и итератора `listIterator`.

В приведенной программе для обработки исключений используется еще один класс `IterException`.

3.8. Обработка исключений

Когда во время выполнения программы происходит ошибка, генерируется так называемое *исключение (исключительная ситуация)*, которое можно *перехватить* и *обработать*. Если исключение не обработать, то программа завершится с ошибкой.

Если в программе возникла исключительная ситуация и в текущем контексте не хватает информации для принятия решения о том, как действовать дальше, информацию об ошибке можно передать во внешний, более общий контекст. Для этого в программе создается объект с информацией об исключении, который затем «запускается» из текущего контекста (говорят, что в программе *запускается исключение*).

Для работы с исключениями можно использовать как встроенные типы, так и создаваемые пользователями (классы).

Вот как в примере 43 выглядит описание собственного класса исключения:

```
class IterException
{};
```

В методе `list::next` выполняется запуск исключения:

```
throw IterException ();
```

Ключевое слово `throw` производит целый ряд действий. Сначала оно создает копию запускаемого объекта и фактически возвращает ее из функции, содержащей выражение `throw`, даже если тип этого объекта не соответствует типу, который положено возвращать этой функции. При этом управление передается в специальную часть программы, называемую *обработчиком исключения*; она может находиться далеко от того места, где было запущено исключение. Также уничтожаются все локальные объекты, созданные к моменту запуска исключения. Автоматическое уничтожение локальных объектов называется «раскруткой стека».

Как правило, для каждой категории ошибок используется свой тип исключений. Предполагается, что информация будет передаваться как внутри объекта, так и в имени его класса; благодаря этому в контексте вызова можно будет решить, как поступить с исключением.

Если вы не хотите, чтобы команда `throw` приводила к выходу из функции, следует создать внутри функции специальный блок, который должен решать проблемы. Этот блок, называемый блоком `try`, представляет область видимости, перед которой ставится ключевое слово `try`:

```
try {
  // Программный код, который может генерировать исключения
}
```

При использовании обработки исключений выполняемый код помещается в блок `try`, а обработка исключений производится после блока `try`. Это существенно упрощает написание и чтение программы, поскольку основной код не смешивается с кодом обработки ошибок. Часто не сам код, который может генерировать исключения, а вызов функции, содержащей этот код, помещают в блок `try`.

Программа должна где-то среагировать на запущенное исключение. Это место называется *обработчиком исключения*. В программу необходимо включить обработчик исключения для каждого типа перехватываемого исключения. Обработчик может перехватывать как определенный тип исключения, так и исключения классов, производных от этого типа.

Обработчики исключений следуют сразу же за блоком `try` и обозначаются ключевым словом `catch`:

```
try {
  // Программный код, который может генерировать исключения
}
catch (type1 id1){
  // Обработка исключений типа type1
}
...
```

```
catch (typeN idN){
  // Обработка исключений типа typeN
}
//Здесь продолжается нормальное выполнение программы...
```

Если id1, ..., idN не нужны, их можно опускать. Тип исключения обычно предоставляет достаточно информации для его обработки, например:

```
try {
  while (it!= s1.end()){
    cout << *it<<" ";
    it=s1.next(it); // ++it;
  }
}
catch (IterException){
  //обработка исключения IterException
}
```

Обработчики должны располагаться сразу же после блока try. Если в программе запускается исключение, механизм обработки исключений начинает искать первый обработчик с аргументом, соответствующим типу исключения. Управление передается в найденную секцию catch, и исключение считается обработанным (т. е. дальнейший поиск обработчиков прекращается). Выполняется только нужная секция catch, а работа программы продолжается, начиная с позиции, следующей за последним обработчиком для данного блока try.

Иногда требуется написать обработчик для перехвата любых типов исключений. Для этой цели используется специальный список аргументов в виде многоточия (...):

```
catch (…){
  cout<<"an exception was thrown"<<endl;
}
```

Поскольку такой обработчик перехватывает все исключения, он размещается в конце списка обработчиков.

Осталось прокомментировать объявление метода:

```
listIterator list::next(listIterator it) throw (IterException);
```

В объявлении функции после списка аргументов указана необязательная *спецификация исключений*. Спецификация исключений состоит из ключевого слова `throw`, за которым в круглых скобках перечисляются типы всех потенциальных исключений, которые могут запускаться данной функцией.

☺ Вообще говоря, вы не обязаны сообщать пользователям вашей функции, какие исключения она может запускать. Однако такое поведение считается нецивилизованным – оно означает, что пользователи не будут знать, как написать код перехвата потенциальных исключений.

3.9. Шаблоны классов

Шаблоны классов, так же как и шаблоны функций, являются трафаретами, по которым компилятор создает *шаблонные классы* и шаблонные функции.

Шаблоны классов часто называют *параметризованными типами*, так как они имеют один или большее количество параметров типа, определяющих настройку шаблона класса на специфический тип данных при создании объекта класса.

Шаблон класса `Stack`, который будет описан ниже, например, может служить основой для создания многочисленных классов `Stack`, необходимых программе типов: «`Stack` для данных типа `int`», «`Stack` для данных типа `double`», «`Stack` для данных типа `Time`» и т. д.

Пример 44. Создать шаблон класса `Stack` на базе массива.

```
//Простой шаблон класса Stack      файл stack.h
#ifndef TSTACK_H
#define TSTACK_H
```

```
#include <iostream>
template <typename T>
class Stack{
private:
  T * start;   //указатель на стек
  int top;     //положение вершины стека
  int size;    //размер стека
public:
  //конструктор с параметром по умолчанию - размер стека
  Stack (int = 10);
  ~Stack() { delete [] start; }        //деструктор
  bool push (const T&);  //помещение элемента в стек
  bool pop (T&);    //выталкивание элемента из стека
  bool view (T&);   //просмотр элемента из вершины стека
  bool isEmpty() const {return top==-1;}
  //true, если стек пустой
  bool isFull() const {return top==size-1;}
  //true, если стек полон
};

//конструктор
template < typename T>
Stack<T>::Stack(int n){
  //создается «разумный» размер стека
  size = n>0 && n<1000 ? n : 10;
  start=new T[size];
  top=-1;
}

//помещение объекта в стек
//в случае успеха возвращается true, в противном случае false
template < typename T>
bool Stack<T>::push (const T& x){
  if (!isFull()) {
    start[++top]=x;     //элемент помещается в стек
    return true;
  }
  return false;
}
//выталкивание элемента из стека
template < typename T>
bool Stack<T>::pop (T& x) {
  if (!isEmpty()) {
    x=start[top--];     //элемент выталкивается из стека
    return true;
  }
  return false;
}
```

```
//просмотр элемента из вершины стека
template < typename T>
bool Stack<T>::view (T& x){
  if (!isEmpty()) {
    x=start[top];   //элемент выталкивается из стека
    return true;
  }
  return false;
}
#endif
```

Описание шаблона класса `Stack` выглядит как традиционное описание класса, за исключением заголовка:

```
template <typename T>
```

Заголовок указывает на то, что описание является шаблоном класса с параметром типа `T`, обозначающим тип элементов объектов класса `Stack`, которые будут создаваться на основе этого шаблона. Идентификатор `T` определяет тип данных-элементов, хранящихся в стеке, и может использоваться при описании типов полей класса и в функциях-членах класса.

Каждое описание функции-члена класса вне шаблона класса начинается с заголовка:

```
template <typename T>
```

Определение функций-членов походит на стандартное описание функций, за исключением того, что тип элементов класса `Stack` всегда указывается имеющим тип `T`.

Чтобы связать каждое описание функции-элемента с областью действия шаблона класса, используется бинарная операция разрешения области действия с именем шаблона класса `Stack<T>`. В этом случае в качестве имени класса выступает `Stack<T>`.

Рассмотрим текст программы, в которой используется класс `Stack`.

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

```
using namespace std;

int main(){
  Stack<int> intStack(5);
  for (int i=0; i<5; i++)
    intStack.push(10-i);
  int x;
  while (intStack.pop(x))
    cout<<x<<' ';
  cout<<endl;
  return 0;
}
```

Объект `intStack` объявляется как экземпляр класса `Stack<int>` (произносится как «`Stack` типа `int`»). При генерации исходного кода для класса `Stack` типа `int` компилятор заменит параметр `T` на тип `int`.

Когда создается объект `intStack` типа `Stack<int>`, конструктор класса `Stack` создает массив элементов типа `int`, представляющий элементы данных стека. Компилятор замещает оператор

```
start=new T[size];
```

в описании шаблона класса `Stack` на оператор

```
start=new int[size];
```

в шаблонном классе `Stack<int>`.

Шаблон класса `Stack` использовал только параметр типа в заголовке шаблона. Но в шаблонах имеется возможность использования и так называемых *нетиповых параметров*. Например, заголовок можно модифицировать, указав в нем нетиповой параметр `int elements` следующим образом:

```
//заголовок для шаблона класса Stack1
//аналога шаблона класса Stack
template < typename T, int elements>
Stack1 {
...
}
```

Тогда объявление типа

```
Stack1<int, 100> intStack1;
```

приведет к созданию во время компиляции шаблонного класса Stack1 с именем intStack1, состоящего из 100 элементов данных типа int. Этот шаблонный класс будет иметь тип Stack1<int,100>. В описании класса в разделе закрытых данных-элементов можно поместить следующее объявление массива:

```
T stackHolder[elements];
//массив для размещения данных стека
```

Если размер класса контейнера, например, массива или стека, может быть определен во время компиляции (например, при помощи нетипового параметра шаблона, указывающего размер), это устранит расходы на динамическое выделение памяти во время выполнения программы.

Определение размера класса контейнера во время компиляции (например, через нетиповой параметр шаблона) исключает возможность возникновения потенциально неисправимой ошибки во время выполнения программы, если оператору new не удастся получить необходимое количество памяти.

Если для специфического типа данных нужен класс, который не соответствует общему шаблону класса, можно явно определить его, отменив тем самым действие шаблона для этого типа. Например, шаблон класса Stack может использоваться для создания массивов любого типа.

Однако можно создать класс Stack для специфического типа, например, типа Student. Для этого нужно просто создать новый класс с именем Stack <Student>.

```
Stack <Student>{
...
}
```

3.10. Наследование классов

В языке C++ абстракция данных осуществляется с помощью классов, инкапсулирующих типы данных и операции над ними. Однако объектно-ориентированный подход выходит за рамки простой инкапсуляции. Применяя наследование и полиморфизм, в языке C++ можно на основе существующих классов создавать новые.

> 🕮 *Наследование* (inheritance) – это способность класса приобретать свойства ранее определенного класса. Один класс может наследовать структуру и поведение другого класса.

В языке C++ производный класс наследует все члены базового класса, за исключением конструкторов, деструктора, перегруженной операции присваивания и определения друзей класса. Таким образом, производный класс содержит в себе все данные-члены и функции-члены базового класса, добавляя к ним новые члены, определенные в нем самом. Кроме того, производный класс может изменять (переопределять) любую наследуемую функцию-член.

При использовании механизма наследования в описании класса появляется новый раздел – защищенный (protected). Члены, помещенные в защищенный раздел, доступны из методов классов-наследников, но скрыты вне определения класса. Общие правила доступа относительно разделов класса в языке C++ представлены в табл. 7.

Определение производного класса начинается с указания типа наследования – `public`, `protected` или `private` и имени базового класса:

```
class <имя производного класса>:
                {public|protected|private} <имя базового класса>
```

Таблица 7

Правила доступа

Раздел базового класса	Открытый (public)	Защищенный (protected)	Закрытый (private)
Доступность из методов базового класса, методов дружественных классов и из дружественных функций	Да	Да	Да
Доступность из методов классов-наследников базового	Да	Да	Нет
Доступность из других классов и функций	Да	Нет	Нет

В языке C++ существует три вида наследования.

Открытое наследование. Открытые и защищенные члены базового класса остаются, соответственно, открытыми и защищенными членами производного класса.

Защищенное наследование. Открытые и защищенные члены базового класса становятся защищенными членами производного класса.

Закрытое наследование. Открытые и защищенные члены базового класса становятся закрытыми членами производного класса.

Среди указанных видов наследования открытое наследование является наиболее важным.

3.11. Открытое наследование

Открытое наследование устанавливает между классами отношение «является», или в английской нотации «is-a».

 При открытом наследовании все, что характеризует объекты класса-предка, является справедливым и для объектов класса-наследника. Это свойство называется *совместимостью типов* объектов. Благодаря этому объект производного класса можно применять вместо объекта базового класса, но не наоборот.

Например, объекту базового типа можно присвоить объект производного типа, указателю на объект базового типа – указатель на объект производного. Объект производного типа также может быть передан в качестве параметра в функцию, вместо объекта базового типа.

Пример 45. Описать иерархию классов `Sphere` («сфера») – `Ball` («мяч»).

Определение класса «сфера» – файл sphere.h.

```
#ifndef SPHERE_H
#define SPHERE_H
class Sphere{
  public:
    //Конструкторы
    Sphere();
    Sphere(double iniRadius);
    // Операции
    void setRadius(double newRadius);
    double getRadius();
    double getVolume();
    void showInform();
  private:
    double radius; //радиус сферы
};
#endif
```

Реализация класса «сфера» – файл sphere.cpp

```
#include "sphere.h"
#include <iostream>
#include <cmath>
using namespace std;

Sphere::Sphere(): radius(1.0){}
Sphere::Sphere(double iniRadius){
  if (iniRadius>0)
    radius=iniRadius;
  else
    radius=1.0;
}
void Sphere::setRadius(double newRadius){
  if (newRadius>0)
    radius=newRadius;
  else
    radius=1.0;
}
```

```
double Sphere::getRadius(){
  return radius;
}

double Sphere::getVolume(){
  double rad3=radius*radius*radius;
  return (4.0*3.14*rad3)/3.0;
}

void Sphere::showInform(){
  cout<<"\n Radius = "<<getRadius()
      <<"\n Volume = "<<getVolume()<<endl;
}
```

Определение класса «мяч» – файл ball.h

```
#ifndef BALL_H
#define BALL_H
#include "sphere.h"
#include <string>
using namespace std;

class Ball: public Sphere{
  public:
    //конструкторы
    Ball();
    Ball(double iniRadius, const string iniName);
    //дополнительные или измененные операции
    void getName(string currentName);
    void setName(const string newName);
    void resetBall(double newRadius, const string newName);

    void showInform();
  private:
    string name;
};
#endif
```

Реализация класса «мяч» – файл ball.cpp

```
#include "ball.h"
#include <iostream>
using namespace std;

Ball::Ball(): Sphere(){
  setName("");
}
Ball::Ball(double iniRadius, const string iniName):
                                          Sphere(iniRadius){
  setName(iniName);
```

```
}
void Ball::setName(const string newName){
  name = newName;
}

void Ball::getName(string currentName){
  currentName=name;
}

void Ball::resetBall(double newRadius, const string newName) {
  setRadius(newRadius);
  setName(newName);
}

void Ball::showInform(){
  cout<<" Ball type - "<<name<<". Specifications:";
  Sphere::showInform();
}
```

📖 Конструктор производного класса выполняется после конструктора базового класса.

Например, конструктор класса `Ball` выполняется после конструктора класса `Sphere`. Это правило действует и в более длинных цепочках наследования.

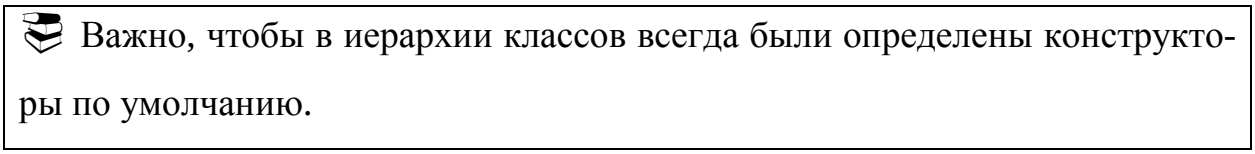
📖 Деструктор производного класса выполняется перед деструктором базового класса.

Соответственно, в нашем примере вначале выполнится деструктор класса `Ball`, затем – деструктор класса `Sphere`. Поскольку они явно не определены, то используются автоматические деструкторы.

Конструктор производного класса отвечает за инициализацию всех элементов данных, добавленных к унаследованным данным из базового класса. Конструктор базового класса выполняет инициализацию унаследованных элементов данных.

Конструкторы класса `Ball` вызывают соответствующие конструкторы класса `Sphere`, используя синтаксис инициализатора.

Для указания, какой именно из конструкторов базового типа следует вызвать в каждом конкретном конструкторе класса-потомка, используется синтаксис списка инициализаторов. Если конструктор базового класса отсутствует в списке инициализации, используется конструктор базового класса по умолчанию.


Важно, чтобы в иерархии классов всегда были определены конструкторы по умолчанию.

Если в классе-потомке не определен ни один конструктор, то будет создан конструктор по умолчанию, который вызовет конструкторы по умолчанию всех предков.

Наследование не открывает доступ к закрытым членам. В нашем примере закрытая переменная `radius` недоступна в классе `Ball`. Однако внутри класса `Ball` можно использовать открытые функции класса `Sphere` – `setRadius` и `getRadius`, которые позволяют получать и изменять значение закрытой переменной.

В реализации класса `Ball` можно вызывать функции, наследуемые от класса `Sphere`. Например, новая функция `resetBall` вызывает наследуемую функцию `setRadius`. Функция `showInform`, переопределенная в классе `Ball`, вызывает унаследованную версию функции класса предка `Sphere::showInform`. Это осуществляется с помощью операции разрешения области видимости `::`.

При переопределении функции базового класса в производном классе списки параметров могут не совпадать. При этом *замещающая функция* переопределяет исходную функцию, но с другим списком параметров.

 Перегрузка при этом не происходит, так как она возможна только в одном пространстве имен. Каждый класс имеет свое пространство имен. Следовательно, производный класс вводит новое пространство имен.

Объекты производного класса могут вызывать открытые функции-члены базового класса. В этом выражается связь между производным и базовым классом, например:

```
Ball myBall(5.0, "Volleyball");
cout<<myBall.getVolume();
```

Две другие важные связи заключаются в том, что указатель базового класса может указывать на производный объект без явного преобразования типов. Ссылка на базовый класс тоже может иметь значением адрес объекта производного класса.

```
Ball myBall(5.0, "Volleyball");
Sphere &pb = myBall;
Sphere *pt = &myBall;
Sphere *p = new Ball(9.0, "basketball");
  cout<<pb.getRadius()<<endl;
  cout<<pt->getVolume()<<endl;
  cout<<p->getVolume()<<endl;
```

Однако указатель или ссылка базового типа позволяет вызывать только методы базового класса, поэтому воспользоваться `pb`, `pt` или `p` для вызова метода `getName()` производного класса нельзя.

Следует обратить внимание и на то, что вызов метода `showInform()` через указатель или ссылку на базовый класс обратится к реализации этой функции в классе `Sphere`, игнорируя ее переопределение в классе `Ball`.

3.12. Позднее связывание и виртуальные функции

Вернемся к последнему замечанию, сделанному в предыдущем параграфе. В случае следующего кода:

```
Sphere *p = new myBall(9.0, "basketball");
  p->showInform();
```

мы получим информацию только о характеристиках сферы. Это обусловлено тем, что определение, какую из перегруженных функций вызвать, происходит в момент компиляции по типу переменной-объекта или переменной-ссылки (указателя).

📖 Определение на этапе компиляции вызываемого варианта перегруженной функции называется *статическим*, или *ранним*, *связыванием*.

Для того чтобы обеспечить возможность определения, какой из перегруженных методов должен быть вызван, не на основе типа переменной-ссылки или указателя, а на основе типа объекта, на который они ссылаются, нужен другой механизм – *динамическое*, или *позднее*, *связывание*.

📖 Определение на этапе выполнения вызываемого варианта перегруженной функции называется *динамическим*, или *поздним*, *связыванием*.

Позднее связывание реализует принцип *полиморфизма*. Полиморфизм позволяет выбирать вызываемую функцию в ходе выполнения программы.

📖 Позднее связывание реализуется с помощью механизма *виртуальных функций*.

Для того чтобы сделать функцию виртуальной, нужно перед ее объявлением в базовом классе указать ключевое слово `virtual`.

📖 Если объявление метода в базовом классе начинается с ключевого слова `virtual`, то это делает функцию виртуальной для базового класса и всех классов, производных от базового класса, – «виртуальная функция всегда виртуальна».

Например, описание класса Sphere должно быть изменено следующим образом:

```
class Sphere{
  public:
    //все, как было описано выше, кроме функции showInform
    ...
    virtual void showInform();
  private:
    double radius; //радиус сферы
};
```

Реализация функции showInform() остается без изменений.

Пример 46. Создать базовый класс figure, для хранения размеров различных двумерных фигур и вычисления их площадей. Создать два класса-потомка rectangle и triangle, представляющие, соответственно, прямоугольник и треугольник на плоскости.

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

class figure {
  protected:
    double x,y;
  public:
    //используется автоматический конструктор по умолчанию
    void set_dim(double x0, double y0) {
      x=x0;
      y=y0;
    }
    virtual void show_area(){
     cout<<"Area figures for this class is not defined"<<endl;
    }
};

class triangle: public figure {
  public:
    //используется автоматический конструктор по умолчанию
    //вызывающий конструктор по умолчанию базового класса
    void show_area() {
      cout<<endl<<"triangle with height "<<x<<" and base "<<y;
      cout<<endl<<"has an area "<<x*0.5 *y<<endl;
    }
};
```

```
class rectangle : public figure {
  public:
    //используется автоматический конструктор по умолчанию
    //вызывающий конструктор по умолчанию базового класса
    void show_area() {
      cout<<endl<<"rectangle with dimensions "<<x<<" * "<<y;
      cout<<endl<<"has an area "<<x*y<<endl;
    }
};

int main() {
  figure *p;
  triangle t;
  rectangle r;

  p=&t;
  p->set_dim(10.0,5.0);
  p->show_area();

  p=&r;
  p->set_dim(10.0,5.0);
  p->show_area();
  return 0;
}
```

В примере 46 наследование не имеет целью расширение базового класса. Каждый из классов-наследников реализует свое собственное поведение на основе единого описанного в базовом классе характера поведения.

Следует обратить внимание на то, что благодаря использованию раздела `protected` поля класса `figure` являются закрытыми для объектов класса, но могут быть использованы в классах-потомках без применения методов доступа типа `set/get`.

Во многих случаях вообще нет смысла давать определение виртуальной функции в базовом классе. Например, в классе `figure` определение функции `show_area()` – лишь способ корректно сообщить об ошибке использования. Объект «фигура» без конкретизации вида фигуры не может иметь площади. Для того чтобы не прибегать к такому искусственному

приему, имеется возможность определять виртуальную функцию без реализации.

📖 Виртуальная функция без реализации называется *чисто виртуальной*.

Синтаксис объявления чисто виртуальной функции:

```
virtual <тип> <имя> (<список параметров>) = 0;
```

📖 Если класс имеет хотя бы одну чисто виртуальную функцию, его называют *абстрактным классом*.

Для абстрактного класса не могут быть созданы объекты. Такой класс может служить только в качестве базового в системе наследования и для создания указателей и ссылок, которые будут использованы при реализации динамического полиморфизма.

Если в базовом классе имеется чисто виртуальная функция, производный класс должен иметь определение ее собственной реализации. Если реализация хотя бы одной из чисто виртуальных функций не будет выполнена, производный класс, в свою очередь, будет абстрактным.

Рассмотрим еще одну иерархию наследования для геометрических фигур на плоскости, на базе абстрактного класса `shape`.

Пример 47. Создать на базе абстрактного класса `shape` классы `circle` и `filled_circle`.

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

class shape {
  public:
    //используется автоматический конструктор по умолчанию
    virtual ~shape() {}
    virtual double square() const =0;
    virtual shape* clone() const =0;
    virtual void debug(ostream &out) const=0;
};
```

```
class circle: public shape {
  public:
    circle(double r=0): radius(r) { }
    ~circle() {}
    double square() const {
      return 3.14*radius*radius;
    }
    circle* clone() const {
      return new circle(getRadius());
    }
    void debug (ostream & out) const {
      out<<" radius = "<<radius <<endl;
    }
    double getRadius() const {
      return radius;
    }
  private:
    double radius;
};

class filled_circle: public circle {
  public:
    filled_circle (double r, int c): circle(r), color(c) { }
    ~filled_circle() {}
    filled_circle* clone() const {
      return new filled_circle(getRadius(), getColor());
    }
    void debug (ostream & out) const {
      circle::debug(out);
      out<<" color = "<<color <<endl;
    }
    int getColor() const {
      return color;
    }
  private:
    int color;
};

int main() {
  shape *p;
  circle t(5);
  filled_circle r(10,255);
  p=&t;
  p->debug(cout);
  p=&r;
  p->debug(cout);
  return 0;
}
```

Следует обратить внимание на то, что деструктор базового класса тоже объявлен виртуальным.

```
virtual ~shape() {}
```

Важно, чтобы деструктор был виртуальным, если класс будет использоваться в качестве базового класса.

 Если базовый класс не имеет явного деструктора, а у потомков класса появятся явные деструкторы, например, освобождающие динамическую память, то возможна ошибка утечки памяти.

Важно гарантировать, чтобы при уничтожении объекта был вызван деструктор именно того класса-наследника, к которому он относится. Если базовый класс не требует выполнения явного деструктора, не следует полагаться на деструктор по умолчанию. Вместо этого необходимо описать виртуальный деструктор с пустой реализацией.

Конструкторы не могут быть виртуальными. Производный класс не наследует конструкторы базового класса, поэтому бессмысленно делать их виртуальными.

3.13. Отношения включения и подобия

Открытое наследование, рассмотренное ранее, устанавливает между классами отношение «является», или в английской нотации «is-a». Иными словами, все, что характеризует объекты класса-предка, является справедливым и для объектов класса-наследника. Это свойство называется совместимостью типов объектов. Благодаря этому объект производного класса можно применять вместо объекта базового класса, но не наоборот.

Между классами могут существовать и другие отношения. Отношение «содержит», или включение («has-a»), означает, что один класс содер-

жит в качестве члена объект другого/других классов. Таким образом, наследование вообще не применяется для отношения «содержит».

Используя закрытое наследование, можно реализовать отношение «подобен», или «as-a». В этом случае потомок может воспользоваться при своей реализации средствами предка, не позволяя, однако, ни объектам, ни собственным потомкам их использовать.

Отношение включения («has-a») определено в том случае, когда описываемый класс в качестве элементов содержит объекты другого (других) классов.

Пример 48. Описать класс для представления сведений о студенте. Сведения содержат информацию: имя студента, курс, группа, массив оценок, полученных за время обучения. Информация об оценках представляется объектом класса `ArrMark`.

Сначала рассмотрим описание и реализацию класса `ArrMark`.

```
#include <string>
#include <iostream>
using namespace std;
class error{ };
class ArrMark{
  private:
    int size;
    int *arr;
    error er;
  public:
    ArrMark() {
      arr = NULL;
      size=0;
    }
    explicit ArrMark (int n, int val=0);
    ArrMark(const int* pn, int n);
    ArrMark(const ArrMark &a);
    virtual ~ArrMark();
    int Size() const {
      return size;
    }
    double Average() const;
    int & operator[](int i);
```

```
    ArrMark & operator= (const ArrMark & a);
};
ArrMark::ArrMark(int n, int val){
  arr = new int[n];
  size = n;
  for (int i=0; i<size; i++)
    arr[i] = val;
}
ArrMark::ArrMark(const int* pn, int n){
  arr = new int[n];
  size = n;
  for (int i=0; i<size; i++)
    arr[i] = pn[i];
}
ArrMark::ArrMark(const ArrMark & a){
  size = a.size;
  arr = new int[size];
  for (int i=0; i<size; i++)
    arr[i] = a.arr[i];
}
ArrMark::~ArrMark(){
  delete []arr;
}
double ArrMark::Average() const{
  double sum=0;
  for (int i=0; i<size; i++)
    sum+=arr[i];
  if (size>0)
    return sum/size;
  else
    throw er;
}
int & ArrMark:: operator[](int i){
  if (i<0 || i>=size)
    throw er;
    else
    return arr[i];
}
ArrMark & ArrMark:: operator=(const ArrMark &a){
  if (this==&a)
    return *this;
  delete []arr;
  size= a.size;
  arr= new int[size];
  for (int i=0; i<size; i++)
    arr[i] = a.arr[i];
  return *this;
}
```

Теперь перейдем к описанию класса Student.

```
class Student{
  private:
    string name;
    ArrMark mark;
  public:
    Student (): name("NONAME"), mark(){}
    Student (const string &s, int n):name(s), mark(){}
    Student (const string & s, const ArrMark &a):
                                          name(s),mark(a){}
    Student(const string &s, const int* pd, int n):
                                        name(s), mark(pd,n){}
    ~Student() {}
    int & operator[](int i);
    double Average() const;
};

int &Student::operator[](int i){
  return mark[i];
}
double Student::Average() const{
  return mark.Average();
}
```

Приведем текст главной программы с примерами использования классов ArrMark и Student.

```
int main(){
  int X[]={2,3,4};
  Student st1("tIvanov",X,3);
  cout<<st1.Average()<<endl;
  Student st2("Ivanov",X,3);
  cout<<st2.Average()<<endl;
  ArrMark ab1(2,5);
  Student st3("Petrov",ab1);
  cout<<st3.Average()<<endl;
  return 0;
}
```

 Конструкторы объекта, являющегося членом класса, выполняются до конструктора определяемого класса. Для этого используется список инициализации.

Если вызов конструктора для подобъекта в списке инициализации отсутствует, то вызывается конструктор по умолчанию. Если в классе под-

объекта его нет, то происходит ошибка времени компиляции. Если у основного объекта отсутствуют конструкторы, то генерируется конструктор по умолчанию, который вызывает конструкторы по умолчанию для подобъектов.

📖 Деструкторы объектов, являющихся членами класса, выполняются после деструктора класса.

Пример 49. Используя ссылочную реализацию класса «список», создать класс «стек».

```
struct el{
  int item;
  el* next;
};
class error{
};
class list{
  private:
    el* head;
    el* end;
    error err;
    public:
    list() {
      head=NULL; end=NULL;
    }
    ~list();
    bool isEmpty() const;
    void inHead(int val);
    void inTail(int val);
    int getFirst()const;
    int getLast()const;
    void delFirst();
    void delLast();
    //в полной реализации могут быть еще методы
};
list::~list(){
  while (head){
    el* cur=head;
    head=head->next;
    delete cur;
  }
  end=head=NULL;
}
```

```
bool list::isEmpty() const {
  return (head==NULL);
}
void list::inHead(int val){
  el* t = new el;
  t->item=val;
  t->next=head;
  head=t;
  if (!head)
    end=head;
}
void list::inTail(int val){
  el* t = new el;
  t->item=val;
  t->next=NULL;
  if (head){
    end->next=t;
    end  = t;
  }
  else{
    head=end=t;
  }
}
int list::getFirst() const{
  if (head)
    return head->item;
  else
    throw err;
}
int list::getLast()const{
  if (head)
    return end->item;
  else
    throw err;
}
void list::delFirst(){
  if (head){
    el *t=head;
    head=head->next;
    delete t;
  }
  else
    throw err;
}
void list::delLast(){
  if (head){
    el *t=head;
    while (t->next != NULL)
```

```
      t=t->next;
    delete end;
    end=t;
  }
  else
    throw err;
}
```

Теперь реализуем класс «стек».

```
class Stack:private list{
  public:
    Stack():list(){}
    bool isEmpty(){
      return list::isEmpty();
  }
  void push(int i){
    inHead(i);
  }
  int top() const{
    return getFirst();
  }
  int pop (){
    int res=getFirst();
    delFirst();
    return res;
  }
};
```

Приведем текст основной программы.

```
int main() {
  Stack p;
  for (int i=0;i<10;i++)
    p.push(i);
  while (!p.isEmpty())
    cout<<p.pop()<<' ';
  return 0;
}
```

📖 Если класс реализует оба отношения – и наследование и включение, то вначале вызывается конструктор предка, затем конструкторы полей объектов, а последним – конструктор наследника. Деструкторы вызываются в обратной последовательности.

Если в классе несколько членов объектов, то их конструкторы вызываются в порядке их объявления, а не в порядке вызова инициализаторов.

В результате изучения модуля студент должен уметь:

- разрабатывать решение задачи в терминах описания взаимодействия объектов;
- создавать новые классы, в том числе путем наследования существующих классов;
- перегружать необходимые операции;
- использовать дружественные классы и дружественные функции;
- создавать итераторы для классов-коллекций;
- использовать виртуальные функции;
- строить иерархию классов;
- описывать шаблоны классов.

В результате изучения модуля студент должен знать:

- основные принципы объекно-ориентированного подхода к программированию;
- назначение и специфику описания и использования конструкторов и деструкторов;
- различия в использовании отношений «является», «содержит» и «подобен»;
- различие между абстрактными и конкретными классами;
- различия между шаблонами классов и шаблонными классами; назначение и специфику шаблонов.

Вопросы для рубежного контроля

1) Что такое конструктор по умолчанию?
2) Когда необходимо явно описывать конструктор по умолчанию?
3) Для чего нужен конструктор копирования?
4) Что должен делать деструктор?
5) Для чего используются спецификаторы доступа?
6) Какие операции можно перегружать только через дружественные функции?
7) Какие операции можно перегружать только через функции-члены класса?
8) Какие особенности имеет перегрузка операции присваивания?
9) Какие преимущества при работе с коллекциями дают итераторы?
10) Какие особенности имеются для перегрузки операций инкремента и декремента?
11) Для чего нужны дружественные классы и дружественные функции?
12) Объяснить различия между шаблоном класса и шаблонным классом.
13) Почему нередко шаблон класса называют параметризованным типом?
14) Объяснить, почему шаблоны для классов контейнеров типа массива или стек часто используют нетиповые параметры.
15) Описать, как отменить действие шаблона, и объяснить, в каких случаях может возникнуть в этом необходимость.
16) Кратко определить каждый из следующих терминов: «наследование», «базовый класс» и «производный класс».
17) В каком порядке вызываются конструкторы в случае, когда описываемый класс является наследником другого класса?

18) В каком порядке вызываются деструкторы в случае, когда описываемый класс является наследником другого класса?
19) В каком порядке вызываются конструкторы в случае, когда описываемый класс в качестве элементов содержит объекты других классов?
20) В каком порядке вызываются деструкторы в случае, когда описываемый класс в качестве элементов содержит объекты других классов?
21) Кратко определить различия в использовании отношений «является», «содержит» и «подобен».
22) В какой последовательности конструируются объекты-элементы, входящие в состав другого класса?
23) Что такое виртуальные функции?
24) Могут ли быть виртуальными конструкторы?
25) Могут ли быть виртуальными деструкторы?
26) Описать различия между статическим и динамическим связыванием.
27) Каким образом полиморфизм способствует расширяемости?

Проектные задания к модулю

Задания группы 1

Реализовать класс, включая конструктор по умолчанию, конструктор копии, операцию присваивания и другие перегруженные операции.

Варианты заданий

1. Односвязный список с однонаправленным итератором.

Функции-члены списка: конструктор по умолчанию; деструктор; конструктор копии; `operator=`; возвращение количества элементов; проверка списка на пустоту; `operator<<` – вывод в поток; `operator+=` – добавление элемента в конец списка; `operator+=` – добавление в конец

списка другого списка; `find` – поиск значения и возвращение итератора на него; `remove` – удаление элемента в позиции, задаваемой итератором; `remove` – удаление диапазона элементов, задаваемого двумя итераторами; `cut` – вырезание подсписка, задаваемого двумя итераторами; `insert` – вставка элемента в позицию, задаваемую итератором; `insert` – вставка подсписка в позицию, задаваемую итератором; `divide` – расщепление списка на два в позиции, задаваемой итератором.

Функции-члены итератора списка: конструктор; `operator*` – ссылка на текущий элемент в списке; `operator++` – префиксный и постфиксный; `eol` – состояние «конец списка».

Внешние функции*:* `foreach` – применение функции, передаваемой в качестве параметра, к диапазону значений, задаваемому парой итераторов; `find` – поиск значения в диапазоне, задаваемом парой итераторов, и возвращение итератора на него; `operator==`; `operator!=`; `operator+` – слияние двух списков в третий; `operator<<` – вывод в поток; `operator>>` – ввод из потока (признак конца ввода – ввод некоторого значения).

2. Двусвязный список с двунаправленным итератором.

Функции-члены списка и внешние функции: те же, что и в варианте 1, кроме `cut` и `divide`. Функции-члены итератора списка: те же, что и в варианте 1, а также : `bol` – состояние «начало списка»; `operator--` – префиксный и постфиксный.

3. Односвязный список с подсчетом ссылок.

Функции-члены и внешние функции: те же, что и в варианте 1. Отличие: в узле списка необходимо хранить не значение, а указатель на объект, содержащий значение и счетчик ссылок (в скольких списках содержится).

4. Множество на базе списка с двунаправленным итератором.

Функции-члены множества: те же, что и в варианте 3; добавить `foreach` – применение функции, передаваемой в качестве параметра, к элементам множества. Функции-члены итератора множества: конструктор; `eos` – состояние «конец множества»; `bos` – состояние «начало множества»; `operator++` – префиксный и постфиксный; `operator--` – префиксный и постфиксный; `operator*` – ссылка на текущий элемент во множестве.

5. Множество на базе бинарного дерева с двунаправленным итератором.

Функции-члены множества и его итератора: те же, что и в варианте 4. При сложностях в реализации сделать операцию удаления элемента из множества простым перебором элементов с перезаписью в другое множество.

6. Очередь с приоритетом с подсчетом ссылок.

Элементы очереди должны иметь перегруженный `operator<`, сравнивающий приоритеты. При вставке элемента в конец очереди он продвигается вперед до элемента с более высоким приоритетом. Элементы могут находиться в нескольких очередях, поэтому должны иметь счетчик очередей, в которых стоят. В очереди необходимо хранить не сами элементы, а указатели на них.

Функции-члены очереди: конструктор по умолчанию; деструктор; конструктор копии; `operator=`; пуста ли очередь; количество объектов в очереди; `operator<<` и `operator+=` – добавление элемента в очередь; `operator>>` – взятие элемента из начала очереди; `operator+=` – добавление очереди к очереди.

Внешние функции: operator==; operator!=; operator+ – слияние очередей; operator<< – вывод в поток; operator>> – ввод из потока (признак конца ввода – ввод некоторого значения).

7. Матрица и вектор.

Функции-члены вектора: конструктор; деструктор; конструктор копии; operator =; возвращение размера; operator[]; унарные операции operator+, operator-; все возможные комбинации operator+=, operator-=, operator*= – и т. д.; operator() с двумя параметрами для выделения подвектора.

Функции-члены матрицы: конструктор; деструктор; конструктор копии; operator=; конструктор с одним параметром, преобразующий число c в матрицу cE; возвращение размеров; operator[] – возвращение строки матрицы в виде вектора; operator+, operator- – бинарные (с контролем размерности); operator* – умножение матрицы на вектор (с контролем размерности); operator+=, operator-=; operator*= – все возможные комбинации; operator() с четырьмя параметрами для выделения подматрицы; swap(i,j) – перестановка двух строк.

Внешние функции: транспонирование матрицы; вычисление определителя матрицы; operator==, operator!= для матрицы и вектора; operator+, operator- – бинарные (с контролем размерности) для матрицы и вектора; operator* – умножение вектора на число; operator* – умножение матрицы на число; operator* – скалярное произведение векторов; operator* – умножение матрицы на матрицу (с контролем размерности); operator<< – вывод в поток для матрицы и вектора.

8. Разреженный массив на базе списка.

Функции-члены массива: конструктор; деструктор; конструктор копии; `operator=`; унарные операции `operator+`, `operator-`; `operator*` – умножение каждого элемента на число; `operator+=`, `operator-=`, `operator*=` – все возможные комбинации.

Операция индексирования `operator[]`, если такого элемента в массиве нет и `operator[]` – в правой части операции присваивания, то возвращать 0 (точнее, значение конструктора по умолчанию для данного типа). Если такого элемента в массиве нет и `operator[]` – в левой части операции присваивания, то вставлять элемент в список. Внешние функции: `operator==`, `operator!=`; бинарные `operator+`, `operator-`.

9. Многочлен на базе списка.

Функции-члены многочлена: конструктор; деструктор; конструктор копии; `operator =`; `operator[]` (см. комментарий к `operator[]` для разреженного массива); унарные `operator+`, `operator-`; все возможные комбинации `operator+=`, `operator-=`, `operator*=`; `operator()` – вычисление многочлена в точке.

Внешние функции: `operator==`; `operator!=`; `operator+`, `operator-`, `operator*`– бинарные; `operator*` – умножение на число; `operator<<` – вывод в поток; `operator>>` – ввод из потока.

10. Ассоциативный массив на базе списка с итератором.

В ассоциативном массиве в качестве ключа использовать строку.

Функции-члены массива: конструктор по умолчанию; деструктор; конструктор копии; `operator=`; `operator[]`; `operator<<`; `operator+=` – добавление элемента; `operator+=` – добавление в конец

списка другого списка; `find` – поиск значения по ключу и возвращение итератора на него; `remove` – удаление элемента по ключу.

Функции-члены итератора массива: конструктор; `eoa` – состояние «конец массива»; `operator++` – префиксный и постфиксный; `operator*` – ссылка на текущий элемент в массиве.

Внешние функции: `operator==`; `operator!=`; `operator+` – соединение двух массивов в третий; `operator<<` – вывод в поток; `operator>>` – ввод из потока (признак конца ввода – ввод некоторого значения); `foreach` – применение функции, передаваемой в качестве параметра, к диапазону значений, задаваемому парой итераторов; `find` – поиск значения в диапазоне, задаваемом парой итераторов, и возвращение итератора на него.

11. Бинарное дерево поиска с итераторами.

Функции-члены дерева: конструктор по умолчанию, деструктор, конструктор копии; `operator=`; определение глубины дерева; `operator<<` и `operator+=` – вставка элемента в дерево; `remove` – удаление элемента из дерева.

Функции-члены итераторов дерева (три итератора для трех направлений обхода): конструктор; `eot` – состояние «конец дерева»; `operator++` и `operator--` – префиксные и постфиксные; `operator*` – ссылка на текущий элемент в дереве.

Внешние функции: `operator==`, `operator!=`; `operator+` – объединение деревьев; `operator<<` – вывод в поток (с одним из итераторов).

12. Дерево с подсчетом ссылок.

Каждый элемент хранит список элементов следующего уровня, на которые он ссылается. Элементы могут находиться в нескольких де-

ревьях, поэтому должны иметь счетчик своих владельцев. В дереве необходимо хранить не сами элементы, а указатели на них.

Функции-члены: конструктор по умолчанию; деструктор; конструктор копии; `operator=`; `operator<<` и `insert` – вставка поддерева или элемента в данное поддерево; `remove` – удаление поддерева или элемента; `find` – поиск элемента в дереве (поддереве); `foreach` – применение функции, передаваемой в качестве параметра, ко всем элементам дерева.

Внешние функции: `operator==`; `operator!=`; `operator<<` – вывод в поток.

Задания группы 2

Реализовать шаблон класса.

Общие указания для реализации шаблонных классов заданий группы 2:

- ✓ Шаблоны классов следует поместить в .h-файл.
- ✓ Тестовая программа должна содержать вызовы всех операций для классов, содержащих данные, как минимум, двух различных типов.
- ✓ Вариант описания конкретной задачи взять из заданий группы 1 текущего модуля.

ЛИТЕРАТУРА

1. *Абрамян М. Э.* 1000 задач по программированию. Ч. I–III. – Ростов н/Д: УПЛ РГУ, 2004. – 128 с.
2. *Абрамян М. Э., Захаренко Е. О., Михалкович С. С., Столяр А. М.* Программирование и вычислительная физика. Вып. IV: Указатели, ссылки и массивы в C++. – Ростов н/Д: УПЛ РГУ, 2005. – 41 с.
3. *Дейтел Х. М., Дейтел П. Дж.* Как программировать на C++. – М.: ЗАО «Издательство БИНОМ», 2000. – 1024 с.
4. *Кениг Эндрю, Му Барбара.* Эффективное программирование на C++. Практическое программирование на примерах. – М.: Издательский дом «Вильямс», 2002. – 384 с.
5. *Керниган Б., Ритчи Д.* Язык программирования Си. – М.: Финансы и статистика, 1992. – 250 с.
6. *Лишнер Рэй.* C++. Справочник. – СПб.: Питер, 2005. – 907 с.
7. *Страуструп Бьерн.* Язык программирования C++. Специальное издание. – М.: ООО «Бином-Пресс», 2004. – 1104 с.
8. *Эккель Брюс.* Философия C++. Введение в стандартный C++. – СПб.: Питер, 2004. – 572 с.
9. *Эккель Брюс, Эллисон Чак.* Философия C++. Практическое программирование. – СПб.: Питер, 2004. – 608 с.

Учебное издание

Русанова Яна Михайловна
Чердынцева Марина Игорьевна

C++ КАК ВТОРОЙ ЯЗЫК В ОБУЧЕНИИ ПРИЕМАМ И ТЕХНОЛОГИЯМ ПРОГРАММИРОВАНИЯ

Учебное пособие

Редактор *Г. А. Бибикова*
Корректор *Г. А. Бибикова*
Компьютерная верстка *Я. М. Русанова, М. И. Чердынцева*
Дизайнер обложки *А. В. Киреев*

Формат 60x841/16. Бумага офсетная. Гарнитура Times.
Печать офсетная. Усл. печ. л. 11,63. Уч.-изд. л. 6,5.
Тираж 75 экз. Заказ № 1420 от 30.11.2010

Издательство Южного федерального университета

Отпечатано в типографии ЮФУ.
344090, г. Ростов-на-Дону, пр. Стачки, 200/1. Тел. 247-80-51.

200