Я. М. Демяненко
М. И. Чердынцева

как второй язык в обучении приемам и технологиям программирования

учебное пособие

ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ЮФУ
РОСТОВ-НА-ДОНУ-ТАГАНРОГ

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Я. М. Демяненко

М. И. Чердынцева

C++ как второй язык в обучении приемам и технологиям программирования

Учебное пособие

2-е издание, переработанное и дополненное

Ростов-на-Дону – Таганрог
Издательство Южного федерального университета
2025

УДК 004.43:004.424(075.8)
ББК 32.973.2+32.973.4я73
Д32

Печатается по решению кафедры прикладной математики и программирования Института математики, механики и компьютерных наук им. И.И. Воровича Южного федерального университета (протокол № 11 от 14 июня 2024 г.)

Рецензенты:
доктор технических наук, профессор кафедры «Информатика» ФГБОУ ВО РГУПС *М. А. Бутакова*;
доцент, заведующий кафедрой информатики и вычислительного эксперимента Южного федерального университета *С. С. Михалкович*

Демяненко Я. М.

Д32 С++ как второй язык в обучении приемам и технологиям программирования: учебное пособие: 2-е изд., испр. и доп./ Я. М. Демяненко, М. И. Чердынцева; Южный федеральный университет. – Ростов-на-Дону; Таганрог: Издательство Южного федерального университета, 2025. – 421 с.
ISBN 978-5-9275-4971-9

В учебном пособии внимание уделяется языку С++ и использованию объектно-ориентированного подхода. Пособие состоит из десяти глав. Излагаемый материал рассматривается на большом количестве подробно разобранных примеров. Пособие адресовано студентам первого и второго курсов, обучающимися по бакалаврским программам по направлениям «Прикладная математика и информатика» и «Фундаментальная информатика и информационные технологии».

УДК 004.43:004.424(075.8)
ББК 32.973.2+32.973.4я73

ISBN 978-5-9275-4971-9

ISBN 978-5-9275-0749-8

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Книга написана в поддержку разработанного нами авторского курса «Программирование на C++», который читается студентам бакалавриата по направлению 02.03.02 «Прикладная математика и информатика». Учебное пособие можно использовать как вспомогательные материалы к лекциям, а также и для самостоятельного изучения языка C++.

При написании книги мы попытались достичь двух целей: во-первых, продемонстрировать особенности языка C++, разбирая приёмы решения задач; во-вторых, облегчить изучение языка C++ студентам, имеющим знания по основам алгоритмизации и программирования, например, на языках Python или PascalABC.NET.

Авторы благодарны своим коллегам сотрудникам института математики, механики и компьютерных наук им. И. И. Воровича ЮФУ доценту М. Э. Абрамяну, доценту Р. М. Мнухину, ассистенту А. С. Коваленко, тщательно проверивших приведённые в книге примеры и высказавших замечания по улучшению текста пособия, и многим нашим коллегам, проявившим интерес к использованию нашей книги. Мы также хотим выразить признательность доценту С. С. Михалковичу за возможность воспользоваться некоторыми интересными примерами.

ВВЕДЕНИЕ

В данном учебном пособии затрагиваются темы, важные для понимания основных концепций языка программирования C++. Книга ориентирована не на конкретную версию компилятора, а на базовые концепции языка C++. Внимание уделяется именно особенностям языка C++ и его парадигмам, в том числе: объектно-ориентированному подходу, использованию шаблонов, обобщённому программированию.

Глава 1 знакомит с базовыми элементами языка C++. На основе примеров решения простых задач происходит неформальное введение в синтаксис языка. Такой подход позволяет студентам, имеющим знания по основам алгоритмизации и программирования, например, на языках Python или PascalABC.NET, достаточно легко перейти к использованию нового языка программирования.

В главе 2 рассматриваются базовые типы данных языка C++, в том числе нативные указатели и особенности операций над ними.

Глава 3 посвящена особенностям представления и использования таких структур данных языка C++, как строки и массивы. В ней делается акцент на использование указателей и адресной арифметикой в языке C++ при работе с массивами и строками.

В главе 4 более подробно рассмотрены типы данных: создание новых типов, функциональные типы. А также представлены вопросы приведения типов и шаблоны функций.

Глава 5 содержит сведения для работы с файлами и списочными структурами. В ней подробно рассмотрено большое количество примеров и решённых задач, что позволит студентам освоить применение теоретических знаний при решении различных задач на файлы и списки.

Глава 6 посвящена парадигме объектно-ориентированного программирования и особенностями её реализации в языке C++.

Объяснение основных принципов и подходов ООП иллюстрируется разбором примеров.

Глава 7 даёт введение в отношения между классами, в том числе в различные виды наследования. Рассматриваются основополагающие понятия ООП, такие как: механизм виртуальности и принцип полиморфизма. Уделяется внимание полиморфным контейнерам и примерам их использования.

В главе 8 вводятся базовые понятия обобщённого программирования. Рассматриваются шаблоны как одно из основных средств реализации парадигмы обобщённого программирования в языке C++.

В главе 9 рассмотрены основные компоненты стандартной библиотеки шаблонов и их использование при решении задач.

Глава 10 содержит материал об умных указателях.

Для облегчения работы с текстом в книге используются принятые авторами в предыдущих книгах [5,12] следующие соглашения.

- ✓ Коды программ, фрагменты примеров, операторы, классы, объекты и методы обозначены специальным шрифтом (`Courier`), что позволяет легко найти их в тексте.
- ✓ Для указания имён файлов и каталогов используется шрифт Arial.
- ✓ Важные термины, встречающиеся впервые, выделены *курсивом*.
- ✓ Определения, термины и материал для запоминания предваряются специальным символом 🕮.
- ✓ Более подробные объяснения отмечены специальным символом 🎓.
- ✓ Знак ➢ означает, что проводится сравнение языка C++ с языками PascalABC.NET или C.
- ✓ Специальным символом ☺ обозначены рекомендации по стилю написания программ.

- ✓ Предостережения от ошибок начинаются со знака ☠.
- ✓ Упражнения, которые необходимо выполнить, обозначены специальным символом  .

ИСПОЛЬЗУЕМАЯ ТЕРМИНОЛОГИЯ

Необходимость включить раздел, посвященный терминологии, вызвана тем, что в литературе по языкам программирования, в том числе и по языку C++, встречается противоречивое употребление основных терминов (оператор, операция, выражение, инструкция, команда). В первую очередь это связано с неоднозначностью переводов оригинальных терминов, используемых в документации. Это приводит к стиранию особенностей синтаксиса языка.

Основные проблемы возникают вокруг переводов трех терминов:

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

Русскоязычная литература не дает строгого определения понятия «оператор». Чаще всего термином «оператором» называется инструкция. Но при этом некоторые авторы к операторам относят и знаки операций +, – и т. д., оправдывая тем, что что трудно перепутать оператор + с условным оператором или оператором цикла.

С этим можно было бы согласиться, но в некоторых случаях возникает существенная путаница, например в случае с оператором присваивания. С одной стороны это знак операции присваивания, а с другой стороны это инструкция присваивания. Из-за этой неоднозначности возникает путаница с терминами «перегрузка операции» и «перегрузка оператора».

Если изучаемый язык программирования интересует нас не только как инструмент создания программ, но и как объект для понимания принципов построения языка и методов работы компилятора, требуется внести формализм в используемую терминологию.

Определим термины «операция», «выражение» и «оператор», поскольку в контексте нашего учебного пособия нам необходимо их однозначная трактовка.

Операция— это команда, которая всегда возвращает результат и может иметь один, два или три операнда.

Операциями являются, например: +, –, ++, =, >, ==, `new`, &, `sizeof`, операция запятая «,» и пр. Для создаваемых типов данных, к которым относятся классы, имеется возможность собственного определения (перегрузки) операций.

Выражения строятся из литералов, идентификаторов, операций и специальных символов. Они предназначены для вычисления значений и/или получения побочных эффектов.

Операторы — это синтаксические конструкции, которые определяют и контролируют то, что должна делать программа. К операторам языка C++ относятся операторы-выражения, операторы-объявления, составные операторы (блоки), операторы выбора, циклы, операторы перехода и операторы обработки исключений [5,12].

ГЛАВА 1. БАЗОВЫЕ ЭЛЕМЕНТЫ ЯЗЫКА С++

1.1. Немного об истории языка С++

Язык С++ был создан в 1983 году Бьёрном Страуструпом [16] на базе языка С. В настоящее время эти два языка развиваются независимо. В то же время между этими языками сохраняется совместимость на базовом уровне и приверженность основным принципам. Благодаря этому программы написанные как на языке С, так и на языке С++ обладают такими характеристиками как эффективность, компактность, быстродействие и переносимость.

 Сам Страуструп [16] неоднократно призывал к максимальному сокращению различий между языками С и С++ для создания максимальной совместимости между ними.

До сих пор в программах, написанных на языке С++, есть возможность применять библиотеки, разработанные для языка С. Это возможно, поскольку ядро языка С является подмножеством ядра языка С++. Но при этом далеко не все средства языка С включены в стандарт С++. В то же время новые стандарты языка С включают некоторые возможности языка С++. К таким возможностям относятся: более строгая проверка типов, размещение объявления переменных как можно ближе к месту их использования и другие.

В первую очередь язык С++ разрабатывался для реализации парадигмы объектно-ориентированного программирования. В то же время он продолжает поддерживать парадигму процедурного программирования. Дальнейшее развитие языка и включение в него шаблонов привело поддержке парадигмы обобщённого программирования. Таки образом язык С++ относится к мультипарадигмальным языкам.

1.2. Особенности языка C++ и его синтаксиса

Каждый язык, в том числе и язык программирования, должен подчиняться определённым правилам. Остановимся на основных правилах языка C++ и особенностях его синтаксиса.

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

Дадим определения некоторых основных терминов, используемых при описании синтаксиса языка.

Идентификаторы, зарезервированные в стандарте языка, называются *ключевыми словами.* Их назначение зафиксировано в стандарте, и их нельзя использовать для других целей. Стандарт ISO/IEC 14882:2020 [15] предусматривает набор ключевых слов, представленный в таблице 1.

Любой идентификатор, используемый в программе и не относящийся к ключевым словам, должен быть объявлен до его первого использования. Идентификаторами могут быть имена переменных, типов, констант, функций и т.д.

Объявление позволяет сообщить компилятору, какой идентификатор программист будет использовать и для каких целей. Объявление добавляет каждый новый идентификатор в область видимости.

Область видимости — это фрагмент программного кода, в котором объявленный идентификатор будет доступен для использования. Внутри области видимости компилятор однозначно определяет, что обозначает идентификатор, и как он может быть использован.

Области видимости бывают двух видов: именованные и неименованные. К именованным областям видимости относятся классы и пространства имён, к неименованным — блоки инструкций, тела функций и неименованные пространства имён.

Таблица 1

Ключевые слова

alignas (C++11)	const_cast	int	static_assert (C++11)
alignof (C++11)	continue	long	static_cast
and	co_await (C++20)	mutable	struct
and_eq	co_return (C++20)	namespace	switch
asm	co_yield (C++20)	new	template
auto	decltype (C++11)	noexcept (C++11)	this
bitand	default	not	thread_local (C++11)
bitor	delete	not_eq	throw
bool	do	nullptr (C++11)	true
break	double	operator	try
case	dynamic_cast	or	typedef
catch	else	or_eq	typeid
char	enum	private	typename
char8_t (C++20)	explicit	protected	union
char16_t (C++11)	export	public	unsigned
char32_t (C++11)	extern	register	using
class	false	reinterpret_cast	virtual
compl	float	requires (C++20)	void
concept (C++20)	for	return	volatile
const	friend	short	wchar_t
consteval (C++20)	goto	signed	while
constexpr (C++11)	if	sizeof	xor
constinit (C++20)	inline	static	xor_eq

Чтобы использовать идентификаторы из именованных областей видимости, нужно уточнить идентификатор (квалифицировать), добавив впереди имя его области видимости с операцией :: . Такое уточнение называется разрешением области видимости.

Программы на языках С и С++ перед компиляцией подвергается предварительной обработке, выполняемой препроцессором. *Препроцессор* удаляет комментарии и вносит изменения в код программы в соответствии с командами препроцессора, называемых *директивами*. Каждая директива пишется на отдельной строке и начинается со знака #.

При создании языка С его авторы Керниган Б., Ритчи Д. [7] особое внимание уделили возможностям работы с адресами памяти через

указатели. *Указатель* является типом данных. Значением переменной типа указатель является адрес ячейки памяти или *нулевой адрес* (признак того, что указатель не хранит никакой адрес). Для таких типов вводится адресная арифметика. Поэтому языки С и С++ относятся и к языкам высокого уровня, и к языкам низкого уровня по возможностям работы с памятью.

Использование указателей является характерной особенностью языков С и С++ Реализация полиморфизма, одного из принципов объектно-ориентированного программирования, в языке С++ основана на использовании указателей. С другой стороны, многолетний опыт работы показывает, что использование указателей и адресной арифметики является источником большого количества ошибок. В результате требование к надёжности и безопасности кода привело к созданию в языке С++ классов-обёрток для безопасной работы с указателями.

1.3. Первые шаги

Программа «Hello, world!» — классический образец первой программы на всех языках, которая приводится практически во всех учебниках. Начнём с разбора аналогичной программы для языка С++.

Пример 1. Реализовать программу, выводящую на экран фразу «Привет, мир!».

Текст этой программы на языке С++ выглядит следующим образом:

```
#include <iostream>
int main(){
  //Для правильного отображения символов кириллицы
  setlocale(LC_ALL, "Rus");
  std::cout << "Привет, мир!" << std::endl;
  return 0;
}
```

Директива препроцессора для подключения заголовочного файла:

```
#include <iostream>
```

Препроцессор подставляет текст заголовочного файла в тот файл, в который мы его подключаем. О директиве `#include` в более ранних (до стандарта 1998 г.) версиях компиляторов языка С++ можно почитать в [5].

В файле <iostream> находятся объявления типов, переменных и функций, предназначенных для реализации ввода/вывода. Стандартные потоковые объекты `cin` и `cout` связаны со стандартными устройствами ввода и вывода.

Чтобы компилятор нашёл объекты `cin` и `cout`, необходимо уточнить их имена указанием пространства имён `std`, в котором они определены. Это пространство содержит определения всех стандартных идентификаторов языка С++.

Для уточнения имён следует использовать операцию разрешения области видимости (`::`). В случае если количество обращений к пространству имён достаточно большое, что приводит к загромождению текста программы, рекомендуется использовать глобальное подключение пространства имён. Оператор подключения выглядит так

```
using namespace std;
```

В этом случае текст программы «Привет, мир!» изменится следующим образом:

```
#include <iostream>
using namespace std;
int main() {
  //Для правильного отображения символов кириллицы
  setlocale(LC_ALL, "Rus");
  cout << "Привет, мир!" << endl;
  return 0;
}
```

Использование пространств имён решает проблему коллизии имён в больших программах.

Для вывода в поток указывается имя потока `cout` и знак операции вывода в поток <<. Операцию можно применять последовательно несколько раз, если нужно вывести несколько объектов.

```
cout << "Привет, мир!" << endl;
```

В этом примере первым объектом вывода является константная строка, заключённая в двойные кавычки, а вторым объектом — манипулятор `endl` (из пространства имён `std`). Он предназначен для перехода в потоке вывода на новую строку и принудительной очистки буфера вывода.

Для корректного отображения символов кириллицы используется функция `setlocale`:

```
setlocale(LC_ALL, "Rus");
```

Вызов этой функции необходимо располагать в самом начале функции `main()`.

> 🕮 Точкой входа в программу на языке C++ является функция с именем `main()`. Она должна быть только одна в программе.

С функции `main()` начинается выполнение программы и в ней же происходит завершение. По окончанию работы функция должна передать управление операционной системе и вернуть ей код завершения с помощью оператора `return`. Все ненулевые коды возврата являются признаками ошибок. В случае нормального завершения функция возвращает `0`.

```
return 0;
```

Заголовок функции `int main()` говорит о том, что функция должна возвращать значение целого типа и в её теле должен присутствовать оператор возврата кода завершения.

Пример 2. С клавиатуры вводится *n* целых чисел. Написать программу для определения максимального из введённых значений.

Вариант текста программы FindIntMax.cpp может выгдядеть так:

```
// FindIntMax.cpp
#include <iostream>
using namespace std;
int main() {
  setlocale(LC_ALL, "Rus");
  int n, max;
  cout<<"Введите количество чисел"<<endl;
  cin>>n;
  cout<<"Введите "<<n<<" чисел"<<endl;
  cin>>max;  //Если число одно, то оно и есть максимальное
  int x;
  for (int i=1;i<n; ++i) {
    cin>>x;
    if (x>max)  //Если текущее число больше максимума
      max=x;
  }
  cout<<"Максимум = "<<max<<endl;
  return 0;
}
```

Для ввода-вывода используются стандартные потоки `cin` и `cout` и операции >> и << соответственно. При этом операции >> и << могут применяться последовательно.

```
cout<<"Введите количество чисел"<<endl;
cin>>n;
cout<<" Введите "<<n<<" чисел "<<endl;
cin>>max;
```

Слева от знака операции >> указывается имя потока, из которого вводятся данные, а справа — переменная, в которую вводится информация. При этом последовательность вводимых символов преобразуется к типу переменной. В случае невозможности преобразования генерируется ошибка.

Чтобы использовать объекты потоков `cin` и `cout` нужно обязательно подключать заголовочный файл <iostream>. Чтобы не использовать операцию разрешения области видимости для имён потоков подключено пространство имён `std`.

Рассмотрим используемые управляющие конструкции `if` и `for`.

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

При объявлении переменных рекомендуется следовать некоторым правилам: размещать объявления переменных как можно ближе к месту их первого использования; параметр цикла `for` (в рассматриваемом примере это переменная `i`) объявлять в заголовке цикла. Это позволит избежать неочевидных ошибок использования параметра цикла за его пределами, поскольку время жизни параметра ограничено циклом.

📖 В циклах `for` и `while` могут быть нужны вспомогательные переменные. Их рекомендуют объявлять прямо в заголовке цикла, чтобы ограничивать телом цикла их область видимости и время жизни.

Оператор `if-else` возможен в двух вариантах: полный (с секцией `else`) и неполный (без неё).

Неполный условный оператор:

```
if (выражение)
  оператор
```

Полный условный оператор:

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

В примере используется неполный оператор:

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

📖 В C++, как и в C выражение после `if` может быть любого типа, но его результат всегда интерпретируется как логический тип. Нулевой результат выражения соответствует логическому значению `false`, а любое ненулевое — значению `true`.

В предлагаемом тексте программы в условном операторе используется логическое выражение `x > max`.

➢ В отличие от языка PascalABC.NET «выражение» в условном операторе `if`, операторе выбора `switch` и операторе цикла `while` всегда заключается в круглые скобки.

В циклах, условном операторе и операторе выбора вместо одиночного оператора может использоваться блок операторов в фигурных скобках, который является аналогом составного оператора в языке PascalABC.NET либо блока операторов, выделяемого отступами, в языке Python.

➢ В отличие от языка PascalABC.NET, где символ точки с запятой (;) разделяет операторы, в языке C++ точка с запятой завершает любой оператор, кроме блока.

Используемый в программе цикл является циклом `for` следующей структуры

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

Он имеет более широкие возможности чем операторы цикла `for` в языках PascalABC.NET и Python.

1.4. Функции как блоки программы

Как правило, алгоритм решения задачи разбивается на относительно независимые подзадачи, которые в языке C++ оформляются в виде функций.

📖 Программа на языке C++ состоит из набора функций, среди которых обязательно присутствует функция `main()`. Кроме того, она может включать объявления и определения глобальных объектов, подключение пространства имён и заголовочных файлов.

Простейшие примеры программ, состоящих только из функции `main()` и подключения заголовочных файлов, были рассмотрены в примерах 1 и 2.

☺ Обратите внимание, что все объекты, объявленные вне функций, входящих в программу, являются глобальными. Рекомендуется минимизировать количество глобальных объектов.

Пример 3. Разработать функции для обмена значениями двух переменных. Одну — для переменных целого типа, и вторую — для переменных вещественного типа. Продемонстрировать использование функций в программе на различных примерах.

```
#include <iostream>

using namespace std; //подключение пространства имён

/*
  Предварительные объявления функций, которые нужны
  компилятору для проверки правильности их вызова
  (соответствие списков параметров)
*/
void func_swap(double &a, double &b);
void func_swap(int &a, int &b);

int main() {
  setlocale(LC_ALL, "Rus");
  int k,m;
  cout<<"Введите два числа целого типа:";
  cin>>k>>m;
  cout<<k<<"  "<<m<<endl;
  func_swap(k, m);
  cout<<k<<"  "<<m<<endl;
  double a,b;
  cout<<" Введите два числа вещественного типа:";
  cin>>a>>b;
  cout<<a<<"  "<<b<<endl;
  func_swap(a,b);
  cout<<a<<"  "<<b<<endl;
  return 0;
}
```

```
//определения функций
void func_swap(double &a, double &b){
  double r = a;
  a=b;
  b=r;
}
void func_swap(int &a, int &b){
  int r=a;
  a=b;
  b=r;
}
```

В тексте программы используются как однострочные, так и многострочные комментарии. Началом однострочного комментария является двойной слэш //. Он может располагаться в любом месте строки. Весь текст, начиная с двойного слэша и до конца строки является комментарием. Многострочный комментарий, находится между группами символов /* и */. В многострочном комментарий может быть любое количество строк.

☺ Правила хорошего стиля рекомендуют использовать в программах на С++ однострочные комментарии. Если всё-таки используются многострочные комментарии, то следует помещать символы /* и */ на отдельных строках друг под другом, а строки с комментариями располагать между ними.

Комментарии предназначены для улучшения читаемости кода программы и используются для документирования.

Необходимость объявления заголовков пользовательских функций в программе вызвана правилами синтаксиса языка и стиля написания программ на языке С++.

Заголовки наших функций, вынесенные в начало программы, являются предварительным объявлением.

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

📖 Всякое имя, в том числе и имя функции, должно быть объявлено до своего первого использования.

В языках C и C++ описание объектов программы состоит из двух частей — объявления и определения [16]. В случае переменных, эти части, как правило, совпадают.

📖 Объявление извещает компилятор о существовании некоторого имени (идентификатора) в программе.

Объявление фактически сообщает компилятору, что объект с указанными именем и характеристиками существует и может использоваться в коде после объявления.

📖 *Определение* отвечает за создание в памяти объекта с указанными именем и характеристиками.

Определение означает, что компилятор должен разместить в памяти переменную или функцию.

📖 В языке C++ определения функций нельзя вкладывать друг в друга.

Определения функций в программе могут располагаться в любом порядке. Но для улучшения читаемости следует придерживаться определённых рекомендаций.

☺ Хороший стиль предполагает размещение определений всех функций в одном месте: либо перед функцией `main()`, либо после функции `main()`.

Размещение определений функций после функции `main()` считается предпочтительным. В этом случае необходимо до функции `main()` использовать объявление функции. Это становится особенно важным в случае многомодульной структуры программы, поскольку определения функций и их вызовы располагаются в разных файлах.

Определение функции помимо заголовка включает в себя тело функции, которое заключено между символами { и }, поскольку является блоком (составным оператором).

В примере 3 тело функции `main()` является линейной программой, состоящей из операторов объявления, ввода-вывода, вызова функции и возврата результирующего значения функции.

✓ операторы объявления переменных, совпадающие с их определением

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

✓ операторы ввода

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

✓ операторы вывода

```
cout<<" Введите два числа целого типа:";
cout<<k<<"  "<<m<<endl;

cout<<" Введите два числа вещественного типа:";
cout<<a<<"  "<<b<<endl;
```

✓ операторы вызова функции

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

✓ оператор возврата результата выполнения функции

```
return 0;
```

В отличие от операций и выражений операторы в языке С++ обязательно завершаются точкой с запятой.

Алгоритмы обработки данных в языке С++, как и в других языках, поддерживающих императивную парадигму, реализуются в подпрограммах. Но С++ в качестве подпрограмм использует только функции.

В языке C++ существуют два вида вызова функции: в качестве операции внутри выражения и как оператор вызова функции.

Если функция возвращает результат, то она может быть вызвана и как операция в составе другого оператора, и как отдельный оператор.

Если же функция не возвращает результат, то вызывать её имеет смысл только как оператор. В этом случае она семантически эквивалентна процедуре в языке PascalABC.Net.

Обычно если у функции нет возвращаемого результата или возвращаемый функцией результат не используется, то функцию используют ради побочного эффекта.

В примере 3 рассмотрены функции, у которых нет возвращаемого значения. Такие функции описываются с использованием ключевого слова `void` перед именем функции.

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

Каждая из них вызывается как отдельный оператор ради побочного эффекта.

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

Обе функции предназначены для выполнения одних тех же действий — обмена местами значения двух переменных. Отличие заключается только в типах переменных. Поэтому и названия функций удобно сделать одинаковыми. Говорят, что в этом случае имеет место перегрузка имени функции.

Перегрузка имени функции — определение нескольких функций с одним и тем же именем и с разными списками параметров. Различаться могут как количество параметров, так и их типы.

Компилятор для функций создаёт внутренние имена, в которых кроме собственно имени функции включается информация о типах параметров и способах их передачи. Внутренне имя называется *сигнатурой*. Функции могут быть перегружены, если они имеют разные сигнатуры. Тип возвращаемого значения не входит в сигнатуру.

Какую именно из перегруженных функций следует вызвать компилятор решает на основе анализа списка фактических параметров.

Поскольку переменные `k` и `m` целые, при вызове `func_swap(k,m)` компилятор выбирает версию со списком параметров целого типа. Для вещественных переменных `a` и `b` используется вариант со списком вещественных параметров.

Побочный эффект функций `func_swap` проявляется в изменении значений переменных, которые будут использованы в качестве фактических параметров. Поэтому необходимо использовать передачу параметров по ссылке.

В обоих перегруженных вариантах функции `func_swap` в примере 3 оба параметра должны быть изменяемыми, поэтому они оба передаются по ссылке.

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

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

Использование ссылок для передачи параметров позволяет сделать код функции и её вызова боле коротким, простым и удобно читаемым. В то время как указатели требуют использование операций разыменования внутри функции и взятия адреса переменных при вызове функции.

☺ Для изменяемых параметров рекомендуется всегда, когда это возможно, передавать параметры по ссылке, а не через указатели.

Пример 4. Организовать перегрузку функций для нахождения максимума для следующих вариантов параметров: два целых числа; три целых числа; два вещественных числа.

```
#include <iostream>
using namespace std;
inline int maximum(const int a, const int b) {
  return a>b?a:b;
}

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

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

int main() {
  cout<< maximum(3,2)<<endl;
  cout<< maximum(3,2,5)<<endl;
  cout<< maximum(3.2,5.0)<<endl;
  return 0;
}
```

Обычно рекомендуется структурировать программу с помощью функций, выделяя с их помощью даже небольшие подзадачи. Это делает программу легко читаемой. Но, с другой стороны, каждый вызов функций приводит к увеличению накладных расходов, что сказывается на снижении производительности программы.

Чтобы снижать накладные расходы на вызовы функций в C++ используется механизм *встраиваемых* (`inline`) *функций*. Их использование эффективно для небольших функций.

Спецификация `inline` перед указанием типа результата всего лишь является рекомендацией компилятору сделать функцию встраиваемой. Компилятор сам решает, будет ли выполнена рекомендация. В случае положительного решения компилятор вставляет копию кода функции в тех местах, где находятся вызовы этой функции. В результате сокращаются накладные расходы, но увеличивается размер компилируемого кода.

В случае игнорирования компилятором спецификации `inline` используется обычный механизм вызова функции.

 Поскольку `inline`-функция встраивается непосредственно в код файлов, которые содержат её «вызовы», любое изменение такой функции приводит к перекомпиляции соответствующих файлов.

1.5. Параметры (аргументы) функции по умолчанию

Бывают ситуации, когда у функции, имеющей много параметров, при вызове для части параметров приходится указывать одни и те же значения, что вызывает неудобства.

Для решения проблемы в языке C++, как и во многих других языках, используются *аргументы (или параметры) по умолчанию*. Аргументы по умолчанию — это параметры, значения которых при вызове указывать необязательно, поскольку компилятор автоматически подставит значения, указанные при объявлении такой функции.

 Существует строгий регламент использования аргументов по умолчанию. Они должны быть расположены только в конце списка параметров. Опускать можно только аргументы, расположенные в конце списка подряд.

Пример 5. Вычислить периметры двух треугольников, заданных координатами вершин на плоскости. Для этого использовать собственную функцию с аргументами по умолчанию, вычисляющую расстояние между двумя точками на плоскости.

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

using namespace std;

double dist(double x1, double y1=0, double x2=0, double y2=0);
double dist(double x1, double y1, double x2, double y2) {
  return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}

int main() {
  std::locale::global(std::locale(""));
  double x1,x2,y1,y2;

  //Первый треугольник
  //одна из вершин лежит в точке с координатами (0,0)
  //две другие – произвольные точки на плоскости
  cout<<"Введите координаты двух точек"<<endl;
  cin>>x1>>y1>>x2>>y2;
  cout<<"периметр треугольника равен ";
  cout<<dist(x1,y1)+dist(x2,y2)+dist(x1,y1,x2,y2)<<endl;

  //Второй треугольник
  //одна из вершин лежит в точке с координатами (0,0)
  //вторая – на оси Ox, третья – на оси Oy.
  cout<<"Введите координату x для точки на оси Ox"<<endl;
  cin>>x1;
  cout<<"Введите координату y для точки на оси Oy"<<endl;
  cin>>y2;
  cout<<"периметр треугольника равен ";
  cout<<dist(x1)+dist(0,y2)+dist(x1,0,0,y2)<<endl;

  return 0;
}
```

Объявление функции `dist` предполагает использование трёх параметров по умолчанию:

```
double dist(double x1, double y1=0, double x2=0, double y2=0);
```

В соответствии с правилами эту функцию можно вызывать с одним, двумя, тремя или четырьмя параметрами. Допустимы следующие варианты списка фактических параметров:

- ✓ только координата x первой точки;
- ✓ координаты x, y первой точки;
- ✓ координаты x, y первой точки и координата x второй точки;
- ✓ координаты x, y обеих точек.

Для однозначности определения значения по умолчанию следует указывать только в одном месте: или в объявлении функции или в её определении.

☺ Правила хорошего стиля рекомендуют указывать значения по умолчанию в объявлении функции.

Аргументами по умолчанию могут быть те параметры, которые имеют некоторые значения, используемые чаще других. Благодаря этому такие значения можно не указывать при вызове данной функции. В случае использования типичных значений параметров это позволяет упростить запись и чтение вызовов функций с большим списком параметров.

☠ Используйте аргументы по умолчанию только по назначению. Не рекомендуется использовать их как флаг или переключатель для выбора фрагмента кода.

В примере 5 продемонстрирован другой способ настройки корректного отображения символов национального алфавита — подключения локали (региональных настроек пользовательского интерфейса), которая задана в настройках операционной системы.

В C++ для этого используется объект типа `locale`. В нем хранятся свойства символов, настройки форматирования и прочая информация. Подключение выполняется следующим образом:

```
std::locale::global(std::locale(""));
```

1.6. Выражения и операции

При выполнении программы на C++ происходит вычисление различных выражений. Выражение состоит из одного или нескольких операндов, соединённых знаками операций. Операндами могут быть константы, переменные, вызовы функций и выражения. Любое выражение в C++ имеет результат и может иметь побочный эффект.

С точки зрения получаемого результата выражения бывают двух видов: именующие (`lvalue`) и значащие (`rvalue`).

Именующее выражение — это выражение, результатом которого является ссылка на объект.

К именующим выражениям относятся: переменные, ссылки на элементы массива по индексу, разыменованные указатели. Именующему выражению почти всегда можно присвоить какое-то значение, поскольку такое выражение всегда связано с областью памяти с известным адресом.

Значащие выражения — это все выражения, не относящиеся к именующим.

К значащим выражениям относятся выражения типа:

```
5
-2
7.3
"Привет, друг!"
3+2
a*5
sin(x)
a[i]+b[i]
firstname + "  " + lastname
```

В языке C++ понятия «именующее выражение» и «значащее выражение» трактуются шире, чем в языке C. Именующие выражения разделены на модифицируемые и константные.

Константные именующие выражения нельзя использовать в левой части оператора присваивания. К ним применима только операция взятия адреса &.

Модифицируемые именующие выражения могут являться левыми операндами в операциях присваивания, к ним применимы операция взятия адреса &, операции инкремента и декремента (++, --).

Во всех операциях, кроме присваивания, взятия адреса и инкремента/декремента требуются значащие выражения. Именующие выражения в этом случае будут неявно преобразованы в значащие.

Именующие выражения получаются в результате выполнения операций индексации [ ], разыменования *, присваивания = и составных операций присваивания (+= -= и т.п.). Все остальные операции возвращают значащие выражения.

Операции приведения типов имеют в качестве результата выражение значащего типа, за исключением операции приведения к ссылочному типу, которая возвращает именующее выражение.

Операция вызова функции может возвращать как именующее выражение, так и значащее. Если функция возвращает ссылку, то результат — именующее выражение, во всех остальных случаях результатом является значащее выражение. При необходимости именующее выражение неявно преобразуется в значащее, обратное преобразование невозможно.

Как и в других языках в C++ операции делятся на группы по количеству операндов: унарные операции, бинарные операции, тернарная операция и n-арная операция (вызов функции).

При написании выражений следует учитывать такие свойства операций, как приоритет и ассоциативность. Приоритет трактуется аналогично трактовке приоритета операций в математике и может быть изменён с использованием скобок. Ассоциативность определяет порядок выполнения нескольких одноприоритетных операций в бесскобочном выражении.

Ассоциативность всех унарных операций определена справа налево. К унарным операциям относятся следующие: арифметические операции изменения знака (+, -), логическая операция отрицания (!), побитовая операция отрицания (~), операции для работы с адресами (&, *), операции инкремента/декремента (++, --).

Ассоциативность бинарных операций может быть разной. Ассоциативность слева направо определена для арифметических, логических операций, операций сдвига, операции индексирования, операция запятая. Ассоциативность справа налево определена для побитовых логических операции и операции присваивания.

Рассмотрим особенности некоторых операций языка C++.

В выражениях операция *присваивания* используется для того, чтобы записать значение правого операнда в левый операнд. Поэтому левый операнд должен быть именующим модифицируемым выражением. Результат операции присваивания совпадает со значением левого операнда после выполнения присваивания. Он может быть использован в других выражениях, в частности, в выражении присваивания. Например, если нескольким переменным нужно присвоить одно и тоже значение:

```
x=y=z=0;
```

Это возможно вследствие того, что операция присваивания является правоассоциативной. У операции присваивания очень низкий приоритет, более низкий — только у операции запятая.

В программах часто встречаются ситуации, когда нужно изменить значение переменной с помощью операции присваивания. Для упрощения записи таких операций используют составные формы присваивания, которые также являются правоассоциативными. Например, операции +=, –=, *= и т. п.

Для изменения значений переменных также используются операции *инкремента* и *декремента.* Каждая из них имеет две формы — префиксную и постфиксную:

`--a, ++a` — префиксная форма;

`a--, a++` — постфиксная форма.

Каждая из форм операции инкремента `a++` `(++a)` является сокращённой формой операции присваивания вида: `a=a+1`. Однако при компиляции такой операции компилятор может использовать более эффективную реализацию, заменив, по возможности, операцию сложения машинной операцией увеличения (`inc`). Операция декремента (`--`) обрабатывается компилятором аналогично.

Обе формы операции инкремента и декремента применимы только к именующим модифицируемым выражениям. Эти операции могут использоваться как в самостоятельных выражениях только ради побочного эффекта (изменения операнда), так и в составе других выражений. В этом случае проявляются особенности префиксной и постфиксной форм. Для префиксной формы результатом выражения инкремента/декремента является значение именующего выражения *после его изменения*. Для постфиксной формы результатом является значение именующего выражения *перед его изменением*. Например:

постфиксная форма

```
aa=5;
bb=aa++;
```

префиксная форма

```
aa=5;
bb=++aa;
```

При использовании постфиксной формы после выполнения фрагмента кода значения переменных равны: `aa=6` и `bb=5`.

При использовании префиксной формы: `aa=6` и `bb=6`.

Операция с самым низким приоритетом — операция *запятая*. Она позволяет использовать два выражения в тех ситуациях, где синтаксис языка C++ требует только одно выражение.

Например, синтаксис оператора цикла `for` предполагает использование трёх выражений в заголовке: инициализации, условия продолжения и итерации. В выражении условия продолжения несколько логических выражений можно объединить в одно с помощью логических операций. В выражениях инициализации и итерации для объединения несколько выражений используется операция запятая. Например:

```
for (k=j=0,p=1; k<n; k++, j=k*2)
```

Операция запятая гарантирует *сериализацию* вычисления выражения, т. е. вычисление правого операнда начнётся только после того, как будет вычислен левый. Это позволяет использовать результат вычисления левого операнда при вычислении правого. Значение правого операнда для операции запятая всегда является результатом такого выражения.

В C++ операции `new` и `delete` предназначены для работы с динамической памятью (кучей). Операция `new` создаёт объект, размещая его в выделенной динамической памяти. Результатом выполнения такой операции является указатель на выделенную память (её адрес). Если выполнение операции завершилось неудачей, будет либо возвращён нулевой указатель, либо сгенерирована исключительная ситуация. Выражение с операцией `delete` ничего не возвращает, а выполняется ради побочного эффекта. В процессе выполнения операции `delete` уничтожается объект в динамической памяти и происходит освобождение памяти и возвращение её в кучу.

Для доступа к элементам массива используется бинарная операция индексации [ ], которую для *пользовательских классов* можно переопределять (перегружать), так же, как и большинство операций языка С++.

1.7. Управляющие конструкции (операторы)

Синтаксис языка С++ требует, чтобы любой оператор, кроме составного (блока), завершался точкой с запятой.

Как и во многих других языках программирования в С++ существует *пустой оператор*. Он предоставляет возможность опускать оператор там, где нет никакого действия. По правилам языка он тоже завершается точкой с запятой и часто используется в операторе цикла с пустым телом. Например, суммирование конечного ряда может быть записано в виде цикла с пустым телом. Ниже представлены две формы записи цикла с пустым телом:

```
for (i=1,s=0; i<n; s+=1.0/i, i++);
for (i=1,s=0; i<n; s+=1.0/i, i++){}
```

☠ Важно помнить, что точка с запятой после заголовка цикла завершает цикл оставляя его тело пустым, что может привести к логической ошибке в программе. В случае такой ошибки пустой оператор повторится много раз, а предполагаемое тело цикла выполнится один раз после цикла.

В языке С++ любое выражение, завершающееся точкой с запятой, является *оператором «выражение»*, который включается в программу только ради его побочных эффектов.

Примеры некоторых операторов «выражение» и их побочных эффектов приведены в таблице 2.

Таблица 2

Побочные эффекты некоторых выражений

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

Язык C++ позволяет использовать операторы цикла трёх видов: `for`, `while` и `do while`.

Циклы с проверяемым условие `while` и `do while` имеют между собой только одно отличие: оператор `while` проверяет выполнение условия перед каждой итерацией, оператор `do while` после выполнения каждой итерации. Обе формы используют условие для продолжения цикла.

```
//цикл с предусловием
while (условие)
  оператор
//цикл с постусловием
do
  оператор
while (условие)
```

Конструкция цикла `for` делает его достаточно универсальным, позволяя включать в заголовок цикла много разных действий. Синтаксис оператора цикла `for` определяется следующим образом:

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

Заголовок цикла `for` включает три блока, разделённых точками с запятыми: блок инициализации, блок условия и блок итерации, каждый из которых является выражением.

Выражение инициализации выполняется один раз, в самом начале работы цикла, и может быть опущено. Чаще всего используется для задания начальных значений переменных. В блоке условия обязательно должно

быть выражение, которое может быть преобразовано к логическому типу. Истинность выражения в блоке условия проверяется перед каждым выполнением итерации цикла. Если блок условия пуст, то считается, что выражение условия истинно. Блок итерации выполняется после каждого повторения тела цикла. Чаще всего используется для изменения переменных через операции инкремента/декремента или сокращённых операций присваивания.

Синтаксис допускает пустоту любого из блоков, в том числе и всех трёх. Например, бесконечный цикл можно записать следующим образом:

```
for (;;) оператор
```

Для организации ветвления в языке C++ используются условный оператор `if else` и оператор выбора `switch`:

```
if (условие)              switch (выражение выбора){
     оператор1               case констВыр1: операторы
[else                        case констВыр2: операторы
     оператор2]               . . .
                              [default     : операторы]
                          }
```

В операторе `switch` выражение выбора может быть любого целочисленного типа. Каждое значение `констВырN` после `case` является меткой строки, которая определяет, куда будет осуществлён переход при соответствующем значении выражения выбора. После чего будут выполнены операторы для всех последующих меток. Чтобы не допустить переход на операторы со следующей меткой, необходимо использовать оператор `break`.

```
switch (binary%2) {
  case 0: cout<<"even"; break;
  case 1: cout<<"odd"; break;
}
```

Необязательная ветвь с меткой `default` выполняется, если ни одна из других меток не подходит. Рекомендуется применять оператор `switch` в случае большого количества вариантов, проверяемых через равенство.

Пример 6. Дан набор символов, завершающийся символом $. Для каждого из символов цифр 0, 1, 3, 7 определить частоту встречаемости.

```
#include <iostream>
using namespace std;
int main() {
  setlocale(LC_ALL, "Rus");
  char ch;
  int nc0,nc1,nc3,nc7;
  nc0=nc1=nc3=nc7=0;
  cout<<" Введите строку, заканчивающуюся символом $";
  cin.get(ch);
  while (ch!='$') {
    switch (ch) {
      case '0': nc0++; break;
      case '1': nc1++; break;
      case '3': nc3++; break;
      case '7': nc7++; break;
      default: break;
    }
    cin.get(ch);
  }
  cout<<"digit 0-"<<nc0<<" digit 1-"<<nc1;
  cout<<" digit 3-"<<nc3<<" digit 7-"<<nc7<<"\n";
  return 0;
}
```

Для организации посимвольного ввода из входного потока используется функция класса `istream`

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

Класс `istream` объявлен в заголовочном файле<iostream>.

1.8. Многофайловые проекты

На языке C++ программа — это одна или несколько функций, среди которых обязательно присутствует функция с именем `main()`. Функция `main` является точкой входа в программу, поэтому она должна быть в единственном экземпляре. Кроме того, программа может включать объявления и определения глобальных объектов, подключения пространств имён и подключения заголовочных файлов.

Если в программе несколько функций, то их можно расположить как в одном файле (исходном модуле), так и в нескольких файлах. При разработке больших программ удобно использовать несколько исходных файлов, объединяемых в один проект. Поскольку языки С++ и С поддерживают раздельную компиляцию модулей, каждый из исходных файлов компилируется отдельно. При отсутствии ошибок компиляция завершается успешно и её результатом является объектный модуль. Каждому успешно откомпилированному исходному модулю соответствует объектный модуль. Программа линковщик собирает откомпилированные файлы, в том числе и библиотечные, в единый исполняемый файл.

Пример 7. Для целого числа посчитать количество и сумму его цифр в десятичной записи. Изменить число вычеркнув из него первую и последнюю цифры.

Для каждого из действий определим функцию:

- ✓ подсчёт количества цифр в десятичной записи числа;
- ✓ вычисление суммы цифр числа;
- ✓ вычёркивание из числа его первой цифры;
- ✓ вычёркивание из числа его последней цифры.

Рассмотрим первый вариант программы с одним файлом.

```
#include <iostream>

using namespace std;

//   объявления функций которые определены после main()
int count_dig(int a);// передача параметра по значению
int sum_dig(int a);
void del_last_dig(int& a);// передача параметра по ссылке
int del_first_dig(int& a);

int main() {
  int val;
  cin>> val;
  cout<<count_dig(val)<<endl;
  cout<<sum_dig(val)<<endl;
```

```
/*поскольку следующие две функции изменяют значение параметра,
  то сохраним исходное число во вспомогательной переменной
*/
  int r= val;
  del_last_dig(r);
  cout<< val <<' '<<r<<endl;
  cout<<del_first_dig(r)<<' '<<r;
}
int count_dig(int a) {
  if (a==0)
    return 1;
  int count=0;
  while (a!=0) {
    count ++;       //операция инкремента
    a/=10;     //составная операция присваивания
  }
  return count;
}

int sum_dig(int a) {
  int sum=0;
  a=abs(a);
  while (a!=0) {
    sum +=a%10;     //составная операция присваивания
    a/=10;          //составная операция присваивания
  }
  return sum;
}

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

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

Объявления (заголовки) пользовательских функций расположены перед функцией `main()`.

➢ В языке C++ в отличие от языка C рекомендуется всегда делать объявление функций.

Если в однофайловом проекте объявление функций является рекомендацией, то в многофайловом — это необходимость, поскольку определение функций могут располагаться не в тех файлах, где они используются.

В примере 7 функции для подсчёта количества и суммы цифр числа используют *передачу параметров по значению*.

```
int count_dig(int a);// передача параметра по значению
int sum_dig(int a);
```

При передаче фактического параметра по значению в описании формального параметра указываются только его тип и имя.

В функциях, которые изменяют число (вычёркивают первую или последнюю цифру) необходима *передача параметра по ссылке*.

```
void del_last_dig(int& a);// передача параметра по ссылке
int del_first_dig(int& a);
```

Для передачи параметра по ссылке в описании формального параметра после указания его типа добавляется модификатор ссылки `&`.

Хотя в обеих функциях параметр передаётся по ссылке (значение фактического параметра изменяется), в одной из этих функций тип результата void, а в другой — тип результата int. Первый способ соответствует по стилю процедурам, а второй — стилю описания функций, принятому в языках С и С++. Если у функции изменяется только один из параметров, то изменённое значение такого параметра возвращается в виде результата функции.

В первом случае вызов функции является самостоятельным оператором. Причём функция вызывается ради побочного эффекта.

```
del_last_dig(r);
cout<< val <<' '<<r<<endl;
```

Второй вариант допускает использование вызова функции либо отдельным оператором, либо внутри выражения. В примере 7 показан вызов

функции `del_first_dig()` внутри выражения для вывода результата в поток.

```
cout<<del_first_dig(r)<<' '<<r;
```

Для сравнения в поток выводятся и значение, возвращаемое функцией, и изменившееся значение фактического параметра.

В функциях примера 7 используются операторы: цикл с предусловием, оператор инкремента и составные операторы присваивания.

- ✓ цикл с предусловием

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

- ✓ инкремент

```
k++;
```

- ✓ составные операторы присваивания

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

Операция деления / для целых операндов определена как деление нацело, а операция % является операцией вычисления остатка от деления.

Продемонстрируем работу с многофайловым проектом на этом же примере. Перенесём определения функций `count_dig`, `sum_dig`, `del_last_dig`, `del_first_dig` в отдельный файл digit.cpp, который будет состоять только из определений функций.

```
int count_dig(int a) {
  if (a==0)
    return 1;
  int count=0;
  while (a!=0) {
    count ++;       //операция инкремента
    a/=10;     //составная операция присваивания
  }
  return count;
}
```

```
int sum_dig(int a) {
  int sum=0;
  a=abs(a);
  while (a!=0) {
    sum +=a%10;     //составная операция присваивания
    a/=10;          //составная операция присваивания
  }
  return sum;
}
void del_last_dig(int& a) {
  a/=10;  //составная операция присваивания
}

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

При попытке откомпилировать этот файл получим сообщение об ошибке компиляции, указывающее, что имя функции `abs()` неизвестно. Для успешной компиляции необходимо подключить заголовочный файл <cmath> с помощью директивы `#include`.

```
#include <cmath>
```

В заголовочном файле <cmath> находятся объявления математических функций и некоторые математические константы.

Для первоначального варианта программы подключение заголовочного файла <cmath> не требовалось, потому что многие заголовочные файлы внутри себя подключают другие заголовочные файлы. В примере 7 такие подключения выполняются в файле <iostream>.

Содержимое файла с функцией `main()` представлено ниже:

```
#include <iostream>
using namespace std;
//   объявления функций которые определены после main()
int count_dig(int a);// передача параметра по значению
int sum_dig(int a);
void del_last_dig(int& a);// передача параметра по ссылке
int del_first_dig(int& a);
```

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

Перед функцией `main()` расположены объявления функций, определённых в файле digit.cpp, потому что без них имена функций станут неизвестными в файле с функцией `main()` и компиляция завершится с ошибкой.

В больших проектах одни и те же функции могут вызываться в разных файлах. Чтобы в каждый файл не добавлять объявления таких функций создаются заголовочные файлы, содержащие объявления функций. В таком случае для добавления строк с объявлениями достаточно подключить заголовочный файл.

Для нашей задачи создадим заголовочный файл digit.h, содержащий объявления четырёх функций:

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

Чтобы подключить digit.h в основной файл используем строку

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

☺ Правило хорошего стиля рекомендуют для каждого cpp-файла, содержащего определения функций, создавать соответствующий ему заголовочный h-файл с объявлениями этих функций. Для облегчения работы с большими проектами желательно, чтобы имена соответствующих cpp- и h- файлов совпадали.

1.9. Заголовочные файлы и библиотеки в C++

Механизм заголовочных файлов в многофайловых проектах позволяет упростить использование функций из других файлов, в том числе и из библиотек.

Библиотечные cpp-файлы всегда сопровождаются одним или несколькими заголовочными файлами, подключаемыми препроцессорной директивой `#include`. Такой способ работы с библиотечными функциями уменьшает количество ошибок при использовании этих функций.

Директива `#include` указывает препроцессору, что необходимо подставить содержимое файла с указанным именем вместо директивы. Различают два способа указания имени подключаемого файла: в угловых скобках и в двойных кавычках.

При использовании угловых скобок препроцессор ищет файл, используя «пути поиска», которые настраиваются в переменной окружения или задаются в командной строке компилятора.

При использовании двойных кавычек препроцессор начинает поиск файла с каталога, где расположены исходные файлы проекта. Если файл не найден, то поиск продолжается, используя «пути поиска».

Первый вариант задания имён файлов в директиве `#include` обычно используется для подключения заголовочных файлов стандартной библиотеки, а второй — для заголовочных файлов, создаваемых программистом.

В языке C++ было принято решение для библиотечных заголовочных файлов не использовать расширения. В то же время язык C++ унаследовал от языка C много библиотечных файлов, в том числе и заголовочных, в которых сохранено традиционное расширение .h. Для соответствия современному стилю языка C++ рекомендуется использовать обновлённые имена заголовочных файлов без расширения .h, снабдив их префиксом c.

Рассмотрим следующий фрагмент:

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

В современном варианте он выглядит так:

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

Такое решение позволяет быстро определить, где используются библиотеки языка C, а где — библиотеки C++.

☺ Использование имени с расширением .h для стандартных заголовочных файлов библиотек считается устаревшим.

Пример 8. Написать функцию для проверки простоты числа и с её помощью вывести все простые числа среди первых N натуральных.

Листинг Prime.cpp содержит функцию `IsPrime` для проверки простоты числа.

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

bool IsPrime (int a) {
  if (a==1)
    return false;
  int sqr_a = int(sqrt((double)a));
  for (int i=2;i<= sqr_a; ++i) {
    if (a%i == 0)
      return false;
    }
  return true;
}
```

Листинг Prime.h содержит объявление функции `IsPrime`.

```
#pragma once
bool IsPrime (int a);
```

Листинг main.cpp содержит основную программу.

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

using namespace std;
```

```
int main() {
  int N;
  cout<<"N = ";
  cin>>N;
  for (int i=1;i<=N; ++i)
    if (IsPrime (i))
      cout<<i<<' ';
  cout<<endl;
  return 0;
}
```

Рекомендуется обратить внимание на условный оператор.

```
if (a == 1)
  return false;
```

Достаточно распространённой является опечатка, когда вместо операции == «сравнить на равенство» используется операция присваивания =. Эта ситуация приводит к сложно отслеживаемой ошибке, потому что синтаксически выражение почти всегда остаётся корректным, а результат выполнения кода может не соответствовать ожидаемому.

Если допустить такую ошибку в рассмотренном условном операторе функции `IsPrime`, то она любое число будет определять как непростое, потому что результат операции `a=1` равен 1, что соответствует истине, и не зависит от исходного значения `a`.

В функции есть ещё один условный оператор.

```
if (a%i == 0)
  return false;
```

В этой конструкции такая опечатка приведёт к ошибке компиляции, потому что слева от знака сравнения на равенство стоит выражение rvalue, а оно не может находиться слева от знака присваивания.

Следует обратить внимание на ещё одну особенность в этой программе. Для ограничений количества повторений цикла использована вспомогательная переменная `sqr_a`, значение которой вычисляется до цикла.

```
  int sqr_a = int(sqrt((double)a));
  for (int i=2;i<= sqr_a; ++i) {. . .}
```

➢Рекомендуется вычисляемые выражения, значения которых не меняются при выполнении цикла, выносить из условия продолжения цикла, поскольку в цикле `for` выражение во втором блоке заголовка вычисляется на каждой итерации перед выполнением тела цикла.

Пример 9. Даны два целых числа. Найти максимум из этих чисел, максимум из сумм цифр этих чисел, максимальную цифру в каждом из чисел и минимум из максимальных цифр этих двух чисел. Для нахождения максимума и минимума из двух чисел написать свои собственные функции.

Для нахождения суммы цифр числа воспользуемся готовой функцией из примера 7, для этого будем использовать уже созданные нами файлы digits.cpp и digits.h.

Функции для определения максимума и минимума из двух целых чисел имеют компактный код, поэтому сделаем их inline-функциями и разместим в заголовочном файле maxmin.h.

Листинг файла maxmin.h

```
#pragma once

inline int my_max(int a, int b) {
  return a > b ? a : b;
}

inline int my_min(int a, int b) {
  return a < b ? a : b;
}
```

Поскольку inline-функции не компилируются, а копии кода функций только лишь размещаются в местах вызова этих функций, в многофайловых проектах следует размещать inline-функции в заголовочных файлах.

📖 Важно помнить, что в заголовочном файле не должно быть определений функций за исключением inline-функций.

Функцию нахождения максимальной цифры в числе поместим в отдельный файл digit1.cpp, поскольку не рекомендуется вносить изменения в готовые файлы из других проектов. Создадим для него соответствующий заголовочный файл digit1.h.

Листинг файла digit1.h

```
#pragma once

int max_digit(int a);
```

Листинг файла digit1.cpp

```
#include <cmath>
#include "maxmin.h"
int midnight(int a) {
  a = abs(a);
  int max_d = 0;
  while (a > 0) {
    max_d = my_max(a % 10, max_d);
    a /= 10;
  }
  return max_d;
}
```

Функция `max_digit` использует функцию `my_max`, определённую в заголовочном файле maxmin.h.

Следует обратить внимание, что при разработке проекты мы не стремились минимизировать количество файлов. Во-первых, не стали менять уже созданный ранее файл из другого проекта. Во-вторых, постарались сгруппировать функции по их назначению.

☺ В больших проектах рекомендуется объединять в cpp-файлы группы взаимосвязанных функций. Для каждого cpp-файла необходимо создавать отдельный заголовочный файл.

Листинг основной программы

```
#include <iostream>
#include "digits.h"
#include "digit1.h"
#include "maxmin.h"
using namespace std;
```

```
int main() {
  int a,b;
  cout << "a = ";
  cin >> a;
  cout << "b = ";
  cin >> b;
  cout << my_max(a,b) << endl;
  cout << my_max(sum_dig(a), sum_dig(b)) << endl;
  cout << max_digit(a) << max_digit(b) << endl;
  cout << my_min(max_digit(a), max_digit(b)) << endl;
  return 0;
}
```

Обратите внимание, что все заголовочные файлы в наших примерах начинаются с директивы препроцессора `#pragma once`. Рассмотрим, зачем она нужна.

В сложных многофайловых проектах можно не уследить, что какой-нибудь заголовочный файл несколько раз подключается в один и тот же файл, вследствие чего возникнут ошибки компиляции. Директива `#pragma once` позволяет не допускать такие ситуации.

Директива препроцессора `#pragma once` очень удобна в использовании и в настоящее время применяют преимущественно её, но она является нестандартной директивой и поддерживается не всеми компиляторами.

Существует стандартный способ недопущения многократного подключения заголовочных файлов — стражи включения, которые реализуются через специальные директивы препроцессора (`ifndef`, `endif`, `define`).

Для каждого заголовочного файла создаётся собственный флаг, с помощью которого не допускается повторное включение. Заголовочный файл следует начинать с директивы `#ifndef имя_флага`, которая проверяет, не был ли определён флаг. Если флаг не установлен, то код заголовочного файла ещё не включался. После директивы `#ifndef` следует устанавливающая флаг директива `#define имя_флага`. Потом следуют

различные объявления, помещаемые в этот заголовочный файл. Конструкция стражей включения завершается директивой `#endif`. Если флаг уже был установлен, то весь код между директивами `#ifndef` и `#endif` игнорируется.

Примерный формат заголовочного файла:

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

Имя флага должно быть уникальным именем в рамках проекта.

☺ Для обеспечения уникальности имени препроцессорного флага рекомендуется использовать имя заголовочного файла, записанное символами в верхнем регистре, а точку заменить символом подчёркивания.

Тогда файл с именем digit1.h будет выглядеть следующим образом.

```
#ifndef DIGIT1_H
#define DIGIT1_H
int max_digit(int a);
#endif    //DIGIT1_H
```

Директива `#pragma once` не только требует меньше кода, но и не допускает коллизии имён препроцессорных флагов.

1.10. Способы обработки ошибок

Ошибки можно разделить на синтаксические ошибки (времени компиляции), ошибки линковки (времени сборки) и ошибки времени выполнения. Синтаксические ошибки находит компилятор. Программа не может быть запущена на выполнение, пока в ней присутствуют ошибки компиляции или ошибки времени сборки.

Отсутствие ошибок на этапе компиляции и линковки не гарантирует корректность написанной программы. В процессе выполнения также могут возникнуть ошибки, называемые ошибками времени выполнения.

Причиной таких ошибок могут быть ошибочные данные, несоблюдение правил работы с динамической памятью или файлами.

Особенность таких ошибок в том, что их трудно искать, поскольку они могут возникать не при каждом запуске программы. Чтобы избегать таких ситуаций, рекомендуется включать в код программы обработку потенциальных ошибок.

Реакция на ошибку может быть нескольких видов:

- ✓ аварийное завершение с выдачей диагностики или без;
- ✓ возврат функцией признака ошибки.

Разберем варианты аварийного завершения на простейшем примере деления на ноль. При отсутствии проверок возможных ошибочных ситуаций программа выглядит следующим образом.

```
int main() {
  int x, y;
  cin >> x >> y;
  cout << x / y;
}
```

При вводе нуля в переменную `y` во время выполнения программы произойдёт аварийное завершение работы программы с выдачей сообщения типа «Integer division by zero».

Зная суть проблемы, можно было перед выполнением деления поставить проверку на равенство нулю переменной `y`, выполнив собственную диагностику. Для аварийного завершения можно использовать одну из функций `exit(code)` или `abort()`.

```
int main() {
  int x, y;
  cin >> x >> y;
  if (y == 0) {
    cout << "error";
    exit(-1);
  }
  cout << x / y;
  return 0;
}
```

Функция `exit(code)` позволяет вернуть в операционную систему код завершения, отличный от нуля, как сигнал ошибки.

В случае y==0 в окне вывода появятся следующие сообщения:

- ✓ «error» — сообщение, выдаваемое самой программой;
- ✓ «(процесс XXXX) завершил работу с кодом -1» — сообщение среды выполнения.

Если заменить функцию `exit(code)` на функцию `abort()`, то в случае ввода ошибочных данных запрашивается выход из программы в результате неустранимой ошибки.

☺ Следует помнить, что функция `exit(code)` завершает выполнение программы более корректно, чем функция `abort()`.

Условный оператор с проверкой ошибочной ситуации можно заменить на `assert(выражение)` — макрос, определённый в заголовочном файле `cassert`.

```
int main() {
  int x, y;
  cin >> x >> y;
  assert(y && "division by zero");
  cout << x / y;
  return 0;
}
```

Макрос `assert(выражение)` применяется для проверки условия корректного выполнения программы. В нашем примере в качестве выражения используется утверждение что переменная `y` не равна нулю.

Если выражение ложно, то само выражение, имя исполняемого файла номер строки, содержащий `assert(выражение)`, направляются в стандартный поток ошибок, а затем вызывается функция `abort()`.

Обычно для диагностики ошибок в выражение включают не только само условие, но и текст, поясняющий ошибку, соединяя их операцией конъюнкции `&&`.

☺ Рекомендуется использовать макрос `assert(выражение)` на этапе отладки программы. В отличие от условного оператора `assert` легко отключается директивой `#define NDEBUG`.

Если код с проверкой ошибочных ситуаций находится в функции, то помимо перечисленных возможностей можно использовать возврат функцией кода ошибки.

Оформим наш пример в виде функции и её вызова в `main()`.

```
bool calc (int a, int e, int &res) {
  if (e) {
    res=a/e;
    return true;
  }
  return false;
}

int main() {
  int x, y,r;
  cin >> x >> y;
  if (calc(x,y,r))
    cout<<r;
  else
    cout<<"error";
  return 0;
}
```

Язык C++ позволяет вызывать функцию, игнорируя возвращаемое значение, поэтому возвращаемый признак ошибки может быть потерян, как в следующем вызове.

```
calc(x,y,r);
cout<<r;
```

Поскольку проверки при вызове каждой функции на наличие возвращаемого признака ошибки приводят к существенному увеличению объёма кода, программисты часто игнорируют эту информацию. В этом случае обнаруженная, но не обработанная ошибка обычно приводит к непредсказуемым последствиям.

Существует более новый способ обработки ошибок времени выполнения, основанный на исключениях. Обработка исключений является неотъемлемой частью языка С++. Чтобы предотвратить ошибки времени выполнения используется *механизм обработки исключений*, который состоит в следующем. В случае потенциальной ошибки создаётся *объект исключения* (*выбрасывается исключение*). В этот момент прерывается выполнение фрагмента кода и управление передаётся *обработчику исключений*, который может находиться в другой части программы. Это позволяет выносить обработку ошибок в отдельные блоки.

Для разных ошибок объекты исключения могут быть разных типов, как встроенных типов, так и специальных классов. Тип исключения никак не связан с типом результата, возвращаемого функцией. Обычно для каждого типа исключения предусматривается свой обработчик. Объект исключения может передать для обработчика дополнительную информацию помимо своего типа.

Для выбрасывания исключения используется оператор

```
throw <исключение>;
```

Генерация исключений чаще всего используется внутри функций. Если обработчик исключения находится за пределами функции, которая выбросила исключение, то происходит досрочное завершение функции.

```
int calc (int a, int e) {
  if (e) {
    return a/e;
  }
  throw " division by zero ";
  //тип исключения const char *
}
```

Чтобы обеспечить обработку исключений, фрагмент кода, в котором могут возникать исключения, помещается в блок `try`.

```
try {
// фрагмент кода, в котором могут выбрасываться исключения
}
```

Блок обработки исключений располагается непосредственно после блока `try`. Обработчиков может быть несколько в соответствии с различными типами исключений, и они следуют друг за другом. Каждый обработчик начинается с ключевого слова `catch`:

```
catch (type1 id1){
// Обработка исключений типа type1
}
catch (type1 id2){
// Обработка исключений типа type2
}
…
catch (type1 idN){
// Обработка исключений типа typeN
}
//Здесь продолжится нормальное выполнение программы...
```

Блоку `catch` можно передать объект исключение и как обычный параметр и как анонимный. В случае анонимного параметра обработчиком используется только тип исключения, в случае обычного параметра можно использовать дополнительную информацию из объекта исключения.

```
int main (){
  int a,e;
  cin>>a>>e;
  try {
    cout << calc(a,e);
  }
  catch (char *s) {
    cout << s<<endl;
    //обработка исключения
  }
  return 0;
}
```

В большинстве случаев обработчик исключительных ситуаций должен обеспечить корректное завершение фрагмента кода, вызвавшего исключение. При этом может выдаваться поясняющая информация о возникшей проблеме.

Рассмотрим пример, в котором переменные `x` и `y` размещены в динамической памяти:

```
int main (){
  int *x,*y;
  x=new int;
  y=new int;
  cin>>*x>>*y;
  try {
    cout << calc(*x,*y);
  }
  catch (char *s) {
    cout << s<<endl;
    delete x;
    delete y;
    return -1;
  }
  delete x;
  delete y;
  return 0;
}
```

В обработчике выдаётся диагностика ошибки, освобождается динамическая память, выделенная для переменных в функции `main` и выполняется завершение выполнения программы с кодом возврата «-1».

☺ При написании обработчиков исключений следует помнить о корректном освобождении динамической памяти и закрытии файлов.

В некоторых ситуациях, например при вводе некорректных данных, обработчик может предложить исправить ошибочную ситуацию (ввести корректные данные) и продолжить выполнение программы.

```
int main (){
  int x, y;
  while (cin >> x>> y){
    try {
      cout << calc(x, y);
    }
    catch (const char * s) {
      //можно ввести новые x, y и вернуться к вычислению
      cout << s<<endl;
      continue;
    }
  }
  return 0;
}
```

Внутри блока `try` может быть оператор генерации исключения, или вызов функции, внутри которой генерируется исключение, или вызов функции, внутри которой вызывается другая функция, которая генерирует исключение. Такая вложенность вызовов функций может иметь любую глубину.

Исключение передаётся в функцию, содержащую блок `try` и блоки `catch`. Если среди обработчиков блока `try` не находится обработчик соответствующего типа, то исключение передаётся на следующий уровень обработки.

При возврате из функции происходит освобождение стека вызова функции, при этом удаляются все локальные объекты. При нормальном завершении происходит передача управления на команду, следующую за вызовом функции, а при возникновении исключений, управление передаётся в соответствующий обработчик.

 Автоматическое уничтожение локальных объектов называют «раскруткой стека».

Рассмотрим цепочку вызовов функций `main()→f1()→f2()→f3()`, где функция `f3()` выбрасывает (генерирует) исключение. При этом обработчик исключения находится в функции `main`. Различие в последовательности действий при нормальном завершении цепочки вызовов и при генерации исключения в функции `f3()` показано на рисунках 1.1 и 1.2.

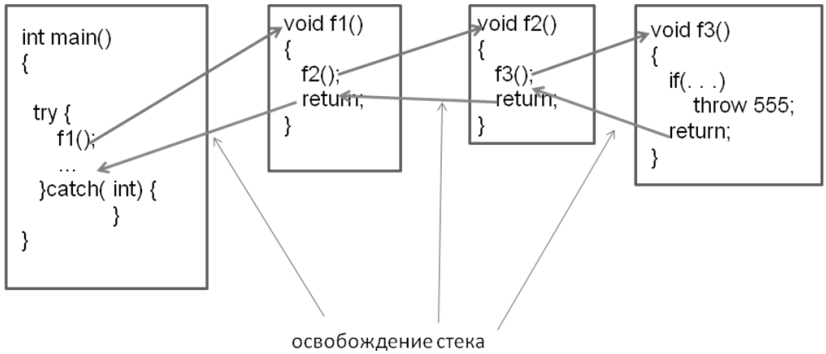

Рис.1.1. Нормальное завершение цепочки вызовов функций

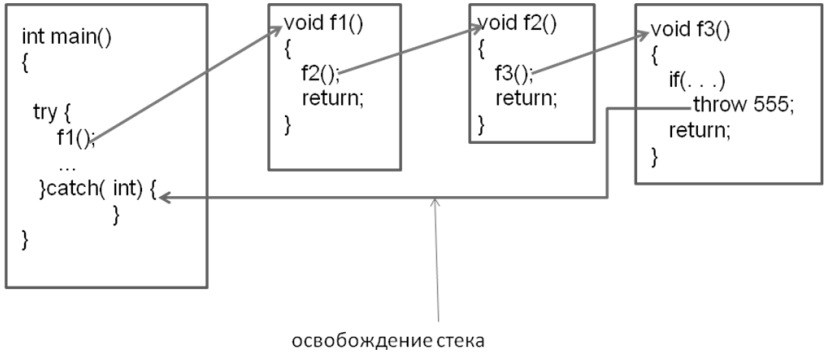

Рис.1.2. Завершение цепочки вызовов при наличии исключения

1.11. Рекурсивные функции

Рекурсивные функции удобно применять для реализации рекурсивных алгоритмов, либо для обработки структур данных, имеющих рекурсивное определение.

Пример 10. Реализовать алгоритм эффективного возведения вещественного числа в целую степень через рекурсивную функцию, используя формулу рекуррентного соотношения.

$$a^n = \begin{cases} 1, \text{ если } n = 0, \\ 1/a^n, \text{ если } n < 0, \\ a \cdot a^{n-1}, \text{ если } n > 0,\ n - \text{нечетное}, \\ (a^{n/2})^2, \text{ если } n > 0,\ n - \text{четное}. \end{cases}$$

Формула рекуррентного соотношения позволяет легко записать рекурсивную функцию.

```
inline bool odd(unsigned int a) {
  return a&1;
}

inline double sqr(double a){
  return a*a;
}

double power(double a, int n) {
  if (n==0)
    return 1;
  if (n<0)
    return 1/power(a,-n);
  if (odd(n))
    return a*power(a,n-1);
  return sqr(power(a, n/2));
}
```

Пример 11. Дана последовательность ненулевых целых чисел, признаком завершения является число 0. Вывести сначала все отрицательные числа, сохраняя их исходный порядок, а затем все положительные — также сохраняя их порядок. Не использовать структуры данных для хранения вводимых значений.

```
#include <iostream>
using namespace std;
void print(){
  int x;
  cin>>x;
  if (x)  {
    if (x<0)
      cout<<x<<' ';
    print();
    if (x>0)
      cout<<x<<' ';
   }
}
```

```
int main(){
  std::locale::global(std::locale(""));
  cout<<"Вводите ненулевые целые числа через пробел"<<endl;
  cout<<"Чтобы завершить ввод введите 0"<<endl;
  print();
  cout<<endl;
  return 0;
}
```

В рекурсивной функции `print()` отрицательные числа выводятся на рекурсивном спуске, а положительные — на рекурсивном подъёме.

Чтобы вводимые и выводимые значения не смешивались, ввод нужно организовать в виде одной строки чисел, разделённых пробелом.

Пример 12. Дана строка, содержащая правильно записанную формулу, т.е. содержит только цифры и знаки, перечисленные в правилах, а также круглые скобки. Правила записи формулы определяются следующим образом:

```
<формула>::=<цифра>|(<формула><знак><формула>)
<знак>::=+|-|*|/
<цифра>::=0|1|2|3|4|5|6|7|8|9
```

Для задания строки использовать посимвольный ввод. Вывести результат вычисления по формуле. Для обработки возможной семантической ошибки (деления на ноль) использовать механизм исключительных ситуаций.

Рекурсивное определение формулы предполагает рекурсивную реализацию.

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

int formula() {
  char c, op;
  int x,y;
  cin>>c;
  if (c>='0' && c<='9') //проверяем, что символ является
цифрой
    return c-'0';
  x=formula();//начало формулы вида (x op y)
  cin>>op;
  y=formula();
```

```
  cin>>c; // пропускаем закрывающуюся скобку
  switch (op){
    case '+': return x+y;
    case '-': return x-y;
    case '*': return x*y;
    case '/': if (y!=0)
                return x/y;
              else
                throw (-55);
    }
    throw(-1);
}

int main() {
  std::locale::global(std::locale(""));

  for (int i=0;i<6;++i)//введём 6 тестовых формул
    try{
      cout<<formula()<<endl;
    }
    catch (int err) {
      if (err==-55)
        cout<<"Ошибка деления на ноль"<<endl;
      else
        cout<<"Ошибка в знаке"<<endl;//недопустимый символ
    }
  return 0;
}
```

На рисунке 1.3 представлена демонстрация работы программы на шести тестовых формулах.

Рис. 1.3. Результаты работы программы

В функции предусмотрены две исключительные ситуации: деление на ноль и возникновение символа, недопустимого правилами записи формулы.

В случае наличия ветвления в функции она обязана возвращать результат по каждой из веток. В операторе выбора `switch(op)` результат формируется только для 4 знаков операций, поскольку по условию вводится правильная формула. Тем не менее компилятор требует возврата результата и в случае других символов. Поэтому после оператора выбора предусмотрен выброс исключения `throw(-1)` вместо оператора `return`.

В обоих случаях в качестве типа исключительной ситуации используется целый тип. Чтобы обработчик мог определить, какая из ситуаций возникла, внутри обработчика используется условный оператор.

```
catch (int err) {
    if (err==-55)
      cout<<"Ошибка деления на ноль"<<endl;
    else
      cout<<"Ошибка в знаке"<<endl;//недопустимый символ
  }
```

ГЛАВА 2. ВСТРОЕННЫЕ ТИПЫ ДАННЫХ

Язык C++, как и многие другие языки программирования, относится к языкам со статической типизацией.

> ➢ При статической типизации типы определяются на этапе компиляции. Некоторые языки, например, Python, являются языками с динамической типизацией. При динамической типизации типы определяются на этапе выполнения программы.

Статическая типизация требует в тексте программы либо явного указания типа переменной, либо компилятор будет автоматически выводить тип по имеющемуся значения переменной.

Встроенные типы данных языка C++ относятся или к фундаментальным (fundamental), или к составным (compound). Фундаментальные иногда называют базовыми.

> ➢ В языке C++, так же, как и в языке C, встроенные типы — это эквиваленты машинных типов данных. В некоторых языках, например, PascalABC.NET встроенные типы данных могут быть как эквивалентами машинных типов, так и отражением математических понятий.

Для представления целочисленных значений и значений с плавающей точкой используются несколько фундаментальных типов. Целочисленные значения и значения с плавающей точкой имеют разные форматы представления данных в памяти. При этом для хранения может выделяться разное количество байтов. В этом и проявляется связь с машинными типами данных.

Представление данных разных типов в памяти определяет допустимый диапазон значений (размерность) для этих типов.

2.1. Целочисленные типы данных

К целочисленным типам языка C++ относятся: `char`, `short int`, `int`, `long int`, и `long long int`. Они перечислены в порядке неубывания их размерности. Эти типы являются основными, но каждый из них, в зависимости от формата представления в памяти, может подразделяться на два вида: знаковые (`signed`) и беззнаковые (`unsigned`). Ключевые слова `signed` и `unsigned` указываются перед основным типом, например, `unsigned int`.

В языке C++, так же, как и в C, действуют правила умолчания при указании типа. В названиях типов `short int`, `long int`, и `long long int` ключевое слово `int` можно опускать (`short`, `long`, и `long long`). По этим же правилам умолчания ключевое слово `signed` также можно опускать. Поэтому типы `short`, `int` и `long`, если не указано `unsigned`, являются знаковыми.

Поскольку в языках C и C++ тип `char` относится к целочисленным типам, он может быть как знаковым, так и беззнаковым.

Для хранения стандартных символов ASCII-кода следовало бы использовать беззнаковый тип (`unsigned char`). Но в стандарте языка это явно не указано и оставлено на усмотрение разработчиков компиляторов. Поэтому в большинстве компиляторов тип `char` определяется как знаковый тип. Если же необходимо использовать тип `char` для числовых значений, то следует указать тип `signed char` для диапазона от –128 до +127, а тип `unsigned char` — для диапазона от 0 до 255.

С появлением двухбайтовых кодовых таблиц, например, UTF-16 или ISO.10646, в языке C++ появился целочисленный тип `wchar_t` для представления расширенного набора символов.

Чтобы в языке C++ узнать размер типа в байтах используется операция `sizeof (имя типа)`, а для переменных вызов операции выглядит следующим образом `sizeof объект`. Операция `sizeof` возвращает беззнаковое целое.

Для определения диапазона допустимых значений можно использовать константы, определённые в заголовочном файле climits, например, для типа `int` используются константы `INT_MIN` и `INT_MAX`.

К целочисленным типам также относится и тип `bool`. Тип `bool` используется для определения переменных логического типа. Значениями типа `bool` являются предопределённые константы `true` и `false`.

> Ранее в языках C и C++ логический тип отсутствовал. При этом любое ненулевое значение интерпретировалось как «истина» (`true`), а нулевое — как «ложь» (`false`).

Следует учитывать, что, несмотря на введение типа `bool`, любое ненулевое значение по-прежнему представляет `true`, а нулевое — `false`, т. е. выполняется неявное приведение типа. Оно выполняется для любых числовых значений и для значений указателя. Существует обратное неявное преобразование из типа `bool` в целый тип, при котором `true` преобразуется в единицу, а `false` — в ноль.

Пример 13. Найти наибольший общий делитель (НОД) двух целых чисел A и B, используя алгоритм Евклида.

Существуют два варианта алгоритма Евклида. Геометрический основан на следующем 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;

// алгоритм можно использовать и для отрицательных чисел
int gcd(int a, int b){
//избавляемся от знаков
  a=abs(a);
  b=abs(b);//
  while (a!=b) {
    if (a>b) a-=b;
    else b-=a;
  }
  return a;
}

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

Второй вариант алгоритма Евклида использует соотношение:

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

Рассмотрим только изменения в функции, вычисляющей НОД.

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

2.2. Поразрядные операции над целочисленными типами

В C++ для целочисленных типов данных, кроме типа `bool`, определены поразрядные операции (операции над двоичными разрядами). Эти операции позволяют работать с отдельными битами внутри байтов. С их помощью можно тестировать, устанавливать или сдвигать отдельных биты.

В поразрядных операциях &, | и ^, несмотря на их коммутативность, традиционно разделяют семантику левого и правого операндов. Левый принято считать проверяемым или подвергаемым изменению, а значение

правого операнда, называемое *маской*, задаёт принцип проверки или изменения левого операнда. Операция сдвига не является коммутативной, её левый операнд представляет значение, подвергаемое сдвигу, а правый операнд определяет величину этого сдвига в двоичных разрядах.

В таблице 3 приведено краткое описание семантики поразрядных (побитовых) операций в предположении, что правый операнд является маской.

Таблица 3

Семантика поразрядных (побитовых) операций

Операция	Значение	Для чего применяется
&	поразрядная конъюнкция (`and`)	0 в разряде маски гарантирует установку 0 в соответствующем разряде результата; 1 в разряде маски используется для проверки того, что в соответствующем разряде левого операнда стоит 1
|	поразрядная дизъюнкция (`or`)	1 в разряде маски гарантирует установку 1 в соответствующем разряде результата; 0 в разряде маски используется для проверки того, что в соответствующем разряде левого операнда стоит 0
^	поразрядное исключающее ИЛИ (`xor`)	дважды применённая операция с одной и той же маской восстанавливает значение левого операнда
>>	побитовый сдвиг вправо	левый операнд подвергается сдвигу, правый операнд задаёт величину сдвига в битах. Для беззнаковых значений при сдвиге освобождающиеся слева разряды заполняются нулями. Для значений со знаком значение знакового разряда (0 или 1) сохраняется
<<	побитовый сдвиг влево	левый операнд подвергается сдвигу, правый операнд задаёт величину сдвига в битах. При сдвиге освобождающиеся справа разряды заполняются нулями
~	поразрядное отрицание (`not`)	унарная операция, инвертирует все биты своего операнда, результат — дополнение операнда до 1

Пример 14. Дано целое число в диапазоне от 0 до 255. Получить двоичное представление этого числа, используя поразрядные операции.

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

void dispBinary (unsigned char u);

int main() {
  unsigned char u;
  cout<<" введите число в диапазоне от 0 до 255:";
  cin >> u;
  cout <<"двоичное представление числа:";
  dispBinary(u);
  cout << " дополнение до единицы: ";
  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";
}
```

Для хранения целого числа в диапазоне от 0 до 255 используется тип `unsigned char` — однобайтовое целое без знака.

В функции `dispBinary()` в цикле `for` в выражении итерации вместо традиционного инкремента используется операция сдвига вправо >>, которая изменяет маску, хранящуюся в переменной `t`.

В маске в одном разряде устанавливается единица, в остальных — нули. Двоичное представление числа 128 определяет маску 10000000. Операция сдвига вправо >> последовательно перемещает единицу от самого левого разряда до самого правого. После последнего сдвига все разряды становятся нулевыми.

Значения маски, последовательно формируемые в цикле, позволяют проверить наличие единицы в каждом двоичном разряде. Если в разряде,

определяемом маской t, стоит 1, поразрядная конъюнкция числа с маской даст ненулевой результат.

Если в тело цикла вставить промежуточную печать значения параметра цикла t, то можно убедиться, что побитовый сдвиг вправо соответствует операции целочисленного деления на два.

Пример 15. Для целого числа реализовать эффективные операции умножения на 2, целочисленного деления на 2 и проверку числа на нечётность.

```
inline int mul_2(unsigned int x){
  return x<<1;
}

inline int div_2(unsigned int x){
  return x>>1;
}

inline bool odd (unsigned int x){
  return x&1;
}
```

Побитовый сдвиг влево на один разряд соответствует умножению числа на два, вправо — целочисленному делению на два. Значение младшего (самого правого) бита позволяет определить чётность числа: у чётного числа младший бит равен нулю, у нечётного —единице.

Пример 16. Для беззнакового целого числа выделить: бит с заданным номером и байт с заданным номером. Биты и байты нумеруются справа налево, начиная с единицы.

```
inline int get_bit(unsigned int x, unsigned int n){
  return (x>>(n-1))&1;
}
inline int get_byte(unsigned int x, unsigned int n){
  return (x>>((n-1)*8))&255;
}
```

Функция get_bit() сдвигает бит с номером n в самый правый разряд, а затем выделяет его с помощью побитовой конъюнкции с единицей.

Функция `get_byte()` сдвигает байт с номером `n` вправо на место младшего байта, а затем выделяет с помощью побитовой конъюнкции с числом 255, которое в двоичном представлении соответствует восьми единичным разрядам (11111111).

Пример 17. Для латинских букв реализовать функции преобразования в нижний и верхний регистры.

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

Решение основано на том, что в таблице кодировок коды прописных (заглавных) и строчных латинских букв отличаются только в пятом разряде (разряды нумеруются справа налево, начиная с нуля). У заглавных букв в этом разряде стоит ноль, а у прописных — единица.

Для того чтобы преобразовать букву нижнего регистра в букву верхнего регистра, нужно установить в пятом разряде единицу, сохранив значение остальных разрядов. Для установки пятого разряда в единицу используется побитовая дизъюнкция с двоичной маской 00100000, которая соответствует десятичному числу 32.

Для того чтобы преобразовать букву верхнего регистра в букву нижнего регистра, нужно установить в пятом разряде ноль, сохранив значение остальных разрядов. В этом случае используется побитовая конъюнкция с двоичной маской 11011111, которая соответствует десятичному числу 223.

Пример 18. Даны два целых числа. Первое число является сообщением, второе — ключом шифрования. Выполнить простейшее

шифрование сообщения (закодировать) и затем расшифровать его (раскодировать).

```
inline int code (int origin, int key) {
  return origin^key;}
int main() {
  int key = 55555;
  int val = 98765;
  val = code(val,key); //кодирование сообщения
  cout<<val<<endl;
  val= code(val,key); //декодирование сообщения
  cout<<val<<endl;
  return 0;
}
```

В данном примере продемонстрировано применение обратимой операции «исключающее или» (^). Повторное применение этой операции с тем же самым правым операндом восстанавливает исходное значение левого операнда (x^y) ^ y = x. Поэтому операция нашла применение в криптографии как простейшая реализация идеального шифра (шифра Вернама).

2.3. Вещественные типы данных

Вещественные числа представляются в формате с плавающей точкой. Для них в языке C++ предусмотрены три типа: `float`, `double` и `long double`. Основным типом считается тип `double`.

Для вещественных типов данных определены следующие арифметические операции: сложение, вычитание, умножение и деление. В языке C++ они являются полиморфными, т. е. тип операции и её результата зависит от типа операндов. В случае операндов разных типов выбирается операция, соответствующая типу с большим диапазоном значений.

Пример 19. Для целого числа n вычислить сумму $\sum_{i=1}^{n} \frac{1}{i}$ двумя способами, перебирая слагаемые в прямом и обратном порядке. Сравнить полученные суммы.

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

double sum_1(int n);
double sum_2(int n);

int main() {
  int n;
  cout<<" введите n "<<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;
}
```

Перебор слагаемых в прямом порядке выполняется в функции `sum_1(int n)`. Функция `sum_2(int n)` перебирает слагаемые в обратном порядке. Для большого значения `n` (например, 2000) результаты суммирования начинают отличаться (рис. 2.1). Это связано с накоплением ошибки округления.

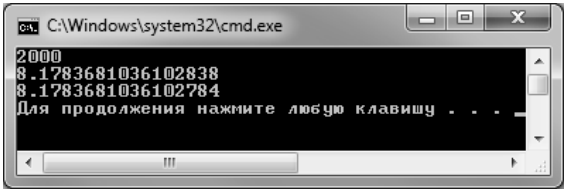

Рис. 2.1. Результаты суммирования для $n = 2000$

Реализация функций `sum_1` и `sum_2` отличается только заголовками цикла `for`.

```
for (int i=1; i<n+1; ++i) s+= 1.0/i;
for (int i=n; i; --i) s+= 1.0/i;
```

В блоке инициализации обоих циклов содержится объявление параметра цикла. Переменные, объявленные внутри цикла, в том числе и в заголовке, являются локальными переменными цикла. Время жизни и область видимости таких переменных ограничено оператором цикла.

Второй оператор цикла `for` демонстрирует использование значения переменной в качестве логического выражения, поскольку ненулевое значение преобразуется к логическому значению `true`, а нулевое, соответственно, к `false`.

Хотя значение переменной `n` не изменяется, в заголовке первого цикла `for` выражение `n+1` вычисляется на каждом шаге, поскольку логическое выражение `i<n+1` проверяется на каждом шаге. Это неэффективно.

☺ Не рекомендуется включать в условия продолжения цикла вычисляемые выражения, значение которых не изменяется.

Поскольку числитель и знаменатель слагаемого в сумме $\sum_{i=1}^{n}\frac{1}{i}$ являются целыми, то операция деления соответствовала бы делению нацело. Тогда слагаемое равнялось бы нулю для всех i>1. Поэтому для получения корректного значения слагаемого числитель записывается в виде вещественной константы (1.0).

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

Чтобы разница в результатах суммирования стала заметной, необходимо увеличить количество выводимых знаков в дробной части. По умолчанию выводится всего 6 цифр. Чтобы изменить количество цифр,

отображаемых после десятичной точки, следует использовать функцию `precision` объекта `cout`.

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

Действие функции `precision(n)` распространяется на все выводимые вещественные значения до следующего вызова `precision`. При этом происходит округление, а не отбрасывание дробной части. Кроме функции `precision` для влияния на способ представления данных в потоке вывода можно использовать функции `width(int)` и `fill(char)`.

Пример 20. Построить таблицу значений функции `sin(x)` на отрезке [a, b] с шагом h. Для вычисления значений функции в точке использовать разложение в ряд $\sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{(2i+1)!}$.

```
#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-2*M_PI : x+2*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;
}
```

Таблицу формируем, выводя последовательно: значение аргумента x, вычисленную сумму бесконечного ряда, аппроксимирующего функцию sin(x), и значение функции x из библиотеки cmath.

Чтобы пройтись по отрезку [a, b] с шагом h использован цикл for с параметром вещественного типа, что удобнее, чем использование цикла while:

```
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;
```

Вместо функций precision(n) и width(n) использованы дополнительные манипуляторы для потока вывода (объекта cout). Каждый манипулятор отправляется непосредственно в поток и действует на следующий за ним элемент вывода. Для использования дополнительных манипуляторов требуется подключать заголовочный файл iomanip.

Манипулятор setw(n) позволяет установить количество позиций для вывода значения следующего за ним элемента вывода. Действие манипулятора setprecision(n) аналогично действию функции precision(n).

Набор манипуляторов шире, чем набор функций объектов ввода/вывода. Манипуляторы предназначены для настройки различных параметров и флагов: форматирования, пропуска пробельных символов при вводе, сброса буфера вывода и т. п.

Вычисление суммы бесконечного ряда реализовано в функции m_sin(double). Суммирование ряда продолжается пока выполняется условие abs(a)>eps, где eps является именованной константой вещественного типа:

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

Чтобы исключить влияние накопления ошибки округления при больших значениях модуля аргумента x, можно воспользоваться периодичностью функции sin(x).

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


Обратите внимание, что строка в коде программы закомментирована. Прежде чем убрать комментарии с этой строки, рекомендуется провести вычислительные эксперименты. Для этого необходимо построить таблицу значений функции при больших значениях аргумента, например, 200 и более.

В данном цикле использована тернарная операция (операция, требующая три операнда), результат которой присваивается переменной x.

```
(x>0) ? x - 2*M_PI : x + 2*M_PI
```

Операнды тернарной операции разделены знаками «вопрос» и «двоеточие». Значение первого операнда приводится к типу bool, если оно равно true, то вычисляется второй операнд, иначе вычисляется третий операнд. Результатом тернарной операции является результат вычисления второго или третьего операнда, в зависимости от того, какой именно из них был вычислен.

2.4. Указатели

Указатель содержит адрес объекта программы и информацию о его типе, т. е. способе интерпретации данных объекта при доступе к памяти.

Объявление указателя:

```
<тип> * <имя>;
```

Указателю можно присвоить: значение адреса переменной соответствующего типа с помощью операции взятия адреса (&), значение другого указателя или адрес нового безымянного объекта, размещаемого в динамической памяти с помощью операции new:

```
int *p, *q, *t;
int a=0;
p=&a;// взятие адреса
t=p;//копирование указателя
q=new int;//создание объекта в динамической памяти
```

Для доступа к данным через указатель используется операция разыменования (*):

```
*q=5;
int b=*q;
cout<<b<<" "<<*q;// Использование b и *q равнозначно
```

Иногда в программе требуется подчеркнуть, что указатель не хранит в данный момент адрес ни одного из объектов программы. Для этого, начиная со стандарта С++11, используется значение `nullptr`. Ранее для этих целей использовалась константа 0, а до стандарта1998 года — `NULL`.

```
p=nullptr; q=nullptr;     //p=NULL; q=0;
```

Нужно понимать, что адресом переменной является целое число, поэтому его значение можно увидеть, так же, как и значение переменной любого базового типа.

Пример 21. В приведённом коде сделать дополнения так, чтобы выводилась информация о расположении в памяти всех переменных программы.

```
#include <iostream>
using namespace std;
double a=2.5, b=0.9;

void mswap(double &a, double &b) {
  double r = a;
  a=b;
  b=r;
}
int mmax(int a, int b) {
  return (a>b)? a:b;
}
int main() {
  int k = 5, m= 7, n;
  n=mmax(k,m);
  mswap(a,b);
  return 0;
}
```

Ниже приведён вариант решения, в котором продемонстрированы:

- ✓ вывод адреса переменной непосредственно с помощью операции & (взятие адреса);
- ✓ сохранение значения адреса в переменную-указатель;
- ✓ многократное использование переменной-указателя для хранения адресов переменных в памяти.

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

double a=2.5, b=0.9;

void mswap(double &a, double &b){
  double r = a;
  double *t = &a;
  cout<<"    &a="<<t<<"  &b="<<&b<<"  &r="<<&r<<endl;
  a=b;
  b=r;
}

int mmax(int a, int b){
    cout<<"    &a="<<&a<<"  &b="<<&b<<endl;
    return (a>b)? a:b;
}

int main(){
  cout << "in main->"<<endl;
  double *q = &a;
  cout<<"  &a="<<q;
  q = &b;
  cout<<"   &b="<<q<<endl;
  int k=7,m=5,n;
  int *p = &k;
  cout <<"  &k="<<p<<"  &m="<<&m<<"  &n="<<&n<<endl;
  cout<<endl<<"in max-> "<<endl;
  n=mmax(k,m);
  cout<<endl<<"in swap ->"<<endl;
  mswap(a,b);
  return 0;
}
```

При каждом запуске программа, а соответственно и её объекты, могут располагаться в разных областях памяти, результаты выполнения при

каждом запуске могут быть разными. Результаты одного из запусков представлены на рисунке 2.2.

Обратите внимание, что адреса представлены в шестнадцатеричном формате. Сравнив значения адресов, можно заметить, что глобальные и локальные переменные располагаются в разных областях памяти.

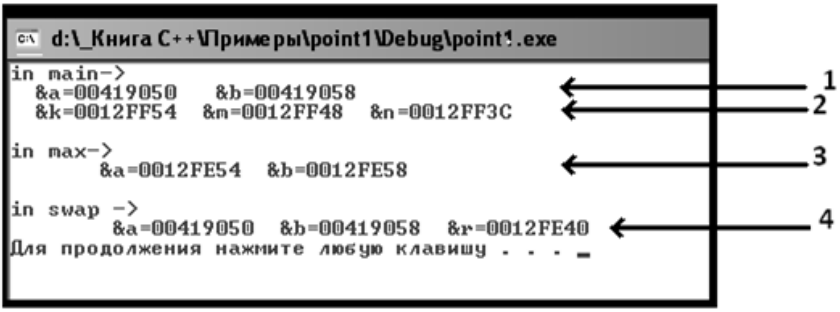

Рис. 2.2. Протокол одного из запусков

Программе для выполнения выделяется некоторая область памяти, которая содержит области для кода программы, для глобальных переменных и для стека вызовов подпрограмм.

Область расположения глобальных переменных относится к статической области памяти программы. Переменные в ней располагаются в порядке описания их в программе (рис. 2.2 — стрелка 1).

```
double a=2.5,b=0.9;
int main() {
  double *q = &a;
  cout<<"  &a="<<q;
  q = &b;
  cout<<"   &b="<<q<<endl;
```

Область размещения локальных переменных относится к автоматической памяти, которая организована в виде стека вызовов подпрограмм. Стек растёт в сторону уменьшения адресов от большего значения к меньшему (рис. 2.2 — стрелка 2).

```
  int k=7,m=5,n;
  int *p = &k;
  cout <<"  &k="<<p<<"  &m="<<&m<<"  &n="<<&n<<endl;
```

При передаче в функцию параметров по значению создаются локальные переменные — копии фактических параметров, которые размещаются на стеке (рис. 2.2 — стрелка 3).

```
int mmax(int a, int b) {
    cout<<"    &a="<<&a<<"  &b="<<&b<<endl;
    return (a>b)? a:b;
}
```

При передаче параметров по ссылке функция получает адрес переменной (фактического параметра), при этом адрес тоже размещается на стеке (рис. 2.2 — стрелка 4).

```
void mswap(double &a, double &b) {
  double r = a;
  double *t = &a;
  cout<<"    &a="<<t<<"  &b="<<&b<<"  &r="<<endl;
  a=b;
  b=r;
}
```

Указатели, как и переменные других типов могут быть константными. Важно понимать различие между константным указателем и указателем на константу. Это различие определяется положением ключевого слова `const` при объявлении указателя.

Указатель на константу	Константный указатель
`const <тип> * <имя>` `<тип> const * <имя>`	`<тип> * const <имя>`

Продемонстрируем корректное использование константных указателей и указателей на константу, а также типичные ошибки, возникающие при некорректном использовании указателей.

```
int aa=100;
int bb=222;

int *const P1=&aa;  //Константный указатель
*P1=987;            //Менять значение разрешено,
//P1=&bb;           //но изменять адрес не разрешается
```

```
const int *P2=&aa;  //Указатель на константу
//*P2=110;          //Менять значение нельзя,
P2=&bb;             //но менять адрес разрешено

const int *const P3=&aa; //Константный указатель на константу
//*P3=155;               //Изменять нельзя ни значение
//P3=&bb;                //ни адрес, к которому такой
                         //указатель привязан
```

В коде закомментированы те фрагменты, которые содержат некорректное использование или константных указателей, или указателей на константу. Попытка убрать комментарии приведёт к ошибке компиляции.

При передаче в функцию параметров через указатель модификатор `const` чаще всего применяется для защиты от изменений либо самого указателя, либо данных, на которые он ссылается.

```
void func(int*a,const int*b,int*const c,const int*const d);
```

Рассмотрим объявление параметров функции `func`. Объявление `int*a` означает, что в теле функции можно изменить как указатель, так и значение, на которое он указывает. Для указателя `b` возможно изменение только его значения (изменение адреса), для `c` — значения, на которое указывает данный указатель. Для указателя `d` недопустимы никакие изменения.

2.5. Указатели на функции

В языках C и C++, как и в других языках программирования, существует тип указатель на функцию, который называют функциональным типом.

В общем виде объявление указателя на функцию выглядит следующим образом:

```
тип_результата (*указатель_на_функцию) (список_параметров);
```

Например,

```
int (*funcPtr)( int,double);
```

В этом примере `funcPtr` — это указатель на функцию с двумя параметрами `int` и `double` и возвращающую результат целого типа.

Значением указателя на функцию может быть адрес любой функции, имеющей соответствующую сигнатуру.

Пусть имеется функция `int func(int,double)`, сигнатура которой соответствует сигнатуре указателя `funcPtr`. Чтобы проинициализировать указатель `funcPtr` адресом функции `func`, можно использовать один из двух вариантов:

```
funcPtr= &func(); // используется взятие адреса функции
funcPtr= func; //используется имя функции
```

В первом случае используется очевидный синтаксис взятия адреса функции с помощью операции `&`. Во втором варианте используется более простая форма записи — имя функции.

> Обратите внимание, что отсутствие круглых скобок в объявлении:
> `int *funcPtr(int,double);`
> приведёт к тому, что будет объявлена не переменная-указатель на функцию, а функция, возвращающая значение типа `int *`. Это связано с приоритетом операций и с правилом интерпретации сложных объявлений.

Довольно часто встречаются ситуации, когда функциональный тип используется для передачи параметров в другую функцию. Поскольку описание функционального типа может оказаться громоздким или сложно читаемым, то рекомендуется использовать создание новых типов.

Для создания новых типов в языке C++ используется конструкция

```
using имя_нового_типа = определение_типа;
```

Например, для `unsigned int` можно создать синоним `UInt`, который будет новым типом, созданным пользователем.

```
using Uint = unsigned int;
```

В результате чего тип `UInt` можно применять точно так же, как тип `unsigned int`.

До стандарта C++11 использовалась конструкция `typedef`

```
typedef определение_типа имя_нового_типа;
typedef unsigned int UInt;
typedef double (*fType)(double x); // старый стиль
using fType = double (*)(double x); // новый стиль
```

Для функционального типа объявление будет выглядеть так:

```
typedef double (*fType)(double x); // старый стиль
using fType = double (*)(double x); // новый стиль
```

Пример 22. Описать функцию построения таблицы значений функции одной вещественной переменной на заданном отрезке с заданным шагом. Используя функцию построить таблицы значений нескольких вещественных функций.

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

const double eps=0.00001;
using fType = double (*)(double a);

double f1(double a){return exp(a)*sin(a);}
double f2(double a){return a*a+0.5*a-7;}
double f3(double a){
  if (abs(a)<eps) throw -101;
  return 1/a;
}

void tabfunc(double a,double b,double h,fType f){
  string line = "-----------------------------------";
  cout<<"Таблица значений функций на отрезке"<<endl;
  cout<<line<<endl;
  cout<<"|      x         |       f(x)      |"<<endl;
  cout<<line<<endl;
  double x = a;
  while (x<b+h/3){
    cout<<"|"<<setw(16)<<setprecision(5)<<x<<"| ";
    try{
      cout<<setw(16)<<setprecision(5)<<f(x)<<"|"<<endl;
    }
```

```
    catch(int){cout<<"функция не определена"<<endl;}
    x+=h;
  }
  cout<<line<<endl;
}

int main() {
  std::locale::global(std::locale(""));
  cout<<"f(x)=exp(x)*sin(x)"<<endl;
  tabfunc (-2,2,0.5,f1);
  cout<<endl<<endl;
  cout<<"f(x)=x*x+0.5*x-7"<<endl;
  tabfunc (-2,2,0.5,f2);
  cout<<endl<<endl;
  cout<<"f(x)=1/x"<<endl;
  tabfunc (-2,2,0.5,f3);
  return 0;
}
```

Для передачи параметра `f` в функцию `tabfunc` определён функциональный тип `fType`:

```
using fType = double (*)(double a);
void tabfunc (double a,double b,double h,fType f)
```

Функции `f1`, `f2`, `f3` соответствуют функциональному типу `fType` и используются как фактические параметры при вызовах функции `tabfunc`.

Результат выполнения программы представлен на рисунке 2.3.

Для выполнения табулирования функций с особенностями потребовалось использование механизма исключений. При этом соответствующие функции в точках, где они не определены, должны выбрасывать исключение типа `int`.

```
double f3(double a){
  if (abs(a)<eps) throw -101;
  return 1/a;
}
```

Пример 23. Продемонстрировать определение, инициализацию, разыменование указателя на функцию и упрощённую форму вызова

```
#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
  fp();          //Упрощённая форма вызова
  void (*fp2)() = func;  //Определение и инициализация
  (*fp2)();      //Разыменование означает вызов
  fp2();         //Упрощённая форма вызова
}
```

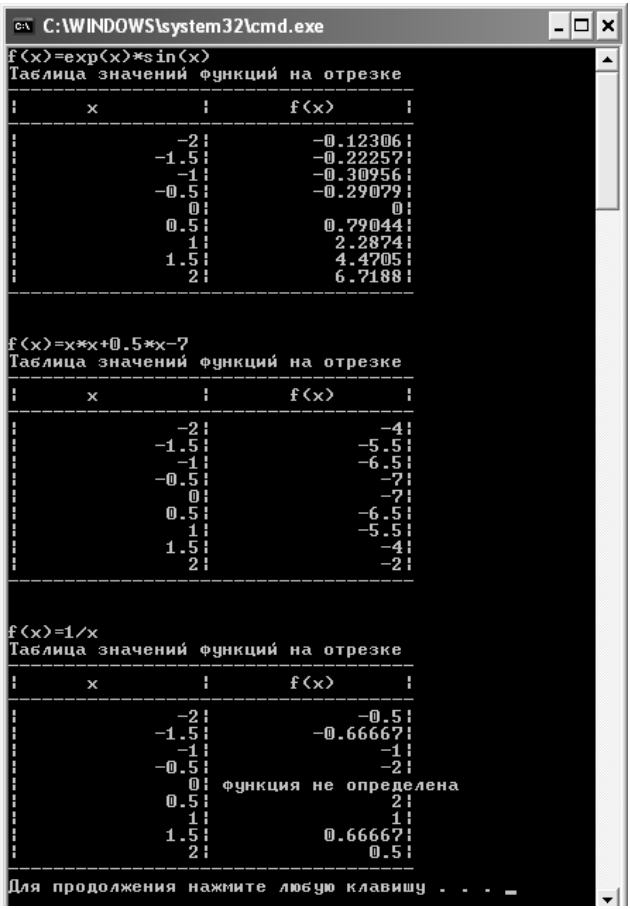

Рис. 2.3. Результат выполнения программы

ГЛАВА 3. МАССИВЫ И СТРОКИ

3.1. Одномерные массивы

При объявлении массива необходимо указать тип элементов в массиве, его имя и количество элементов (размер массива):

```
<тип> <имя>[<размер>];
```

При таком объявлении размер массива остаётся фиксированным до конца выполнения программы. Поэтому такие массивы называют статическими. Как и в других языках для работы с элементами массива используется операция индексации [ ].

Следует обратить внимание на особенность массивов в языках С и С++, вытекающую из стремления к быстродействию и эффективности. Имя массива трактуется как константный указатель на адрес его размещения в памяти (адрес первого элемента).

Пример 24. Написать функцию определения номера первого нулевого элемента в целочисленном массиве. Продемонстрировать использование для массива, содержащего не более 10 элементов.

```
#include <iostream>

using namespace std;

int find_zero(int a[], int n);

int main() {
  std::locale::global(std::locale(""));

  int a[10], n,k;
  do {
    cout << "размер массива" << endl;
    cin>> n;
  } while (n<1 ||n>10); //проверка на допустимость размера

  for(int i=0; i<n; ++i) {
    cout<<i<<"-й элемент массива";
    cin>>a[i];
  }
```

```
  if ( (k= find_zero (a,n))<0)
    cout<<"нет нулей\n";
  else
    cout<<"номер первого нуля = "<<k;
  return 0;
}

int find_zero (int a[], int n) {
  int i=0;
  while (i<n && a[i]!=0)
    i++;
  return i<n ? i : -1;
}
```

Последний элемент массива имеет индекс на единицу меньше, чем размер массива, потому что, как и во многих языках, нумерация элементов начинается с 0.

☠ Необходимо обратить внимание на потенциальный источник ошибок при работе с массивами в языке C++, поскольку в нём отсутствует проверка выхода за границы массива.

В примере продемонстрировано описание параметра функции для передачи одномерного массива:

```
int find_zero (int a[], int n);
```

Поскольку при передаче одномерного массива в функцию компилятор игнорирует значение, указанное в квадратных скобках, то это значение обычно опускают, оставляя скобки пустыми. Размер массива передаётся отдельным параметром.

 Поскольку в языках C и C++ при передаче массивов в качестве параметров в функцию передаётся адрес массива (первого элемента), то элементы массива являются изменяемыми.

Чтобы не допустить изменения элементов массива в функции, к параметру-массиву применяют спецификатор `const`.

```
int find_zero (const int a[], int n);
```

📖 Следует помнить, что в языке C++ действует принцип наименьших привилегий. Поэтому функции не должны модифицировать массивы без крайней необходимости.

Существует ещё один способ описания параметров-массивов с использованием указателей. Он является более универсальным.

```
int find_zero (const int *a, int n);
```

При любом из способов описания параметров-массивов при вызове функции в качестве фактического параметра используется имя массива.

```
k= find_zero (a,n);
```

Изменить в примере 24 описание параметра-массива функции так, чтобы использовался указатель.

Пример 25. Дана последовательность символов. Признаком окончания последовательности является символ «*». Организовать подсчёт количеств символов, встречающихся в последовательности: по отдельности каждой цифры; суммарно всех пробельных литер (пробелов, табуляций и новых строк), а также суммарно всех остальных литер, кроме символа «*».

Рассмотрим одно из возможных решений:

```
#include <iostream>

using namespace std;

void main () {
  int nwhite=0, nother=0;

  int ndigit [10];
  for (int i = 0; i < 10; ++i)
    ndigit [i] = 0;

  char c;
  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 (int i = 0; i < 10; ++i)
      cout<<i<<'-'<<ndigit[i]<<' '<<endl;
  cout<<" white = "<<nwhite<<" other = "<<nother<<endl;
}
```

В задаче можно выделить двенадцать категорий вводимых символов. При этом удобно счётчики для цифр хранить в массиве, а не в отдельных переменных. Для оставшихся категорий символов используются два отдельных счётчика.

Чтобы по символу цифры определить значение индекса в массиве счётчиков цифр находим разность кодов текущего символа и символа нуля. Поскольку тип `char` является целочисленным и не требует преобразования типов, то вычисление индекса выглядит следующим образом `ndigit [c-'0'].`

3.2. Массивы в динамической памяти

Рассмотренный в предыдущем разделе способ объявления массива означает, что память под него будет отведена перед началом выполнения программы или функции и будет закреплена за массивом до окончания времени жизни массива. Такой способ хорошо подходит для ситуаций, когда размер массива заранее известен (к моменту написания кода).

Существует возможность выделять память под массив во время выполнения программы, указывая его размер не «по максимуму», а необходимый при конкретном выполнении программы. Такой массив будет располагаться в динамической памяти, для работы с которой требуются операторы выделения памяти (`new`) и освобождения памяти (`delete`).

Выделение памяти под целочисленный массив a размера n выглядит следующим образом.

```
int *a; int n;
cin>>n;//определение размера массива возможно и другим
способом
a=new int[n];
```

Для освобождения динамической памяти, занимаемой массивом необходимо применить специальную форму оператора delete, которая указывает, что освобождается память, занимаемая массивом:

```
delete[] a;
```

Пример 26. Описать функцию вычисления среднего арифметического элементов вещественного массива. Продемонстрировать её использование для массива произвольного размера.

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

int main() {
  std::locale::global(std::locale(""));
  int n;
  cout <<"Введите размер массива";
  cin>>n;
  double *a;
  a= new double[n];
  for (int i=0; i<n; ++i) {
    cout<<i<<"-й эелмент";
    cin>>a[i];
  }
  cout<<"среднее арифметическое = "<< average(a,n);
  delete[] a;
  return 0;
}

double average(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;
}
```

3.3. Связь массивов и указателей

И для статических, и для динамических массивов имя массива — это адрес массива, который является одновременно и адресом первого элемента массива (`&a[0]`). Адрес — это указатель, а над указателями определены операции адресной арифметики, операции присваивания и сравнения.

Операция присваивания позволяет копировать значения указателей, например, в дополнительные (вспомогательные) переменные-указатели.

К указателям одного типа применимы операции сравнения на равенство/неравенство. Остальные операции сравнения имеют смысл только, если это указатели на один и тог же массив.

К операциям адресной арифметики относятся: +, –, ++, ––. Они используются при работе со структурами данных, в которых элементы расположены в памяти последовательно, например, с массивами.

Операции ++и –– являются унарными, применимы к указателям на элементы массива, если это имеет смысл. Они сдвигают указатель вправо или влево на количество байтов, занимаемых одним элементом массива (на один элемент массива).

Бинарные операции +, –, у которых правый операнд — целое число, предназначены для смещения указателя на количество элементов, задаваемое правым операндом. При этом указатель смещается на количество байтов, кратное размеру адресуемого элемента данных.

Бинарная операция «минус» может иметь также в качестве двух операндов указатели на элементы одного массива. В этом случае разность указателей равна количеству элементов массива, расположенных между ними.

Рассмотрим ситуацию, когда переменная `a` является указателем на начало массива. Увеличение значения переменной `a` на 1 приводит к тому, что указатель смещается к следующему элементу, т. е. к элементу с

индексом равным 1. Отсюда вытекает эквивалентность следующих выражений.

```
a   ~   &a[0]
a+1 ~   &a[1]
a+i ~   &a[i]
```

Поскольку некоторые компиляторы выполняют операцию индексации `a[i]` дольше, чем обращение к элементу массива через указатель `*(a+i)`, желательно в задачах обработки массивов уметь использовать арифметику указателей.

Поскольку выражения `a++` и `a--` изменяют значения переменой `a`, то их нельзя применять к имени массива. Для перемещения по элементам массива с помощью указателей используются дополнительные (вспомогательные) переменные-указатели.

Пример 27. Написать функцию проверки массив целых чисел на симметричность. Использовать адресную арифметику.

```
#include <iostream>
using namespace std;
bool simm_arr(int *a, int n);
int main() {
  int *a, n;
  cout <<"Введите размер массива";
  cin>>n;
  a=new int[n];
  for (int i=0; i<n; ++i) {
    cout<<i<<"-й элемент ";
    cin>>a[i];
  }
  cout<<simm_arr(a,n)<<endl;
  delete []a;
  return 0;
}
bool simm_arr(int *a, int n) {
  int *b=a+n-1;
  while (a<=b)
    if (*a++!=*b--)
      return false;
  return true;
}
```

В функции `simm_arr` для проверки симметричности массива используются два указателя `a` и `b`. Указатель `a`, являющийся параметром, а следовательно, локальной переменной, перемещается от начала массива к середине. На каждой итерации цикла выполняется `a++`. Указатель `b` инициализируется адресом последнего элемента `b=a+n-1`. Он движется к середине массива `b--`. Смещение указателей к центру продолжается пока истинно условие `a<=b` и пока элементы в парах совпадают. Если обнаружено несовпадение `*a++!=*b--`, функция завершается со значением `false`.

Организация работы с массивом через указатели позволяет работать с массивом и его фрагментами через два указателя, которые определяют диапазон элементов. Традиционно диапазон задаётся полуоткрытым справа интервалом `[*begin;*end)`. Первый указатель определяет начало диапазона элементов массива, второй — указатель, значение которого ограничивает диапазон справа.

 Важно помнить, что указатель, ограничивающий диапазон справа, всегда находится за пределами диапазона обрабатываемых элементов.

Пример 28. Для функции, вычисляющей сумму элементов диапазона массива, продемонстрировать её использование для разных диапазонов.

```
#include <iostream>
using namespace std;
int sum_elem(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<<"sum of all ="<<sum_elem(array, array+n)<<endl;
  cout<<"sum of the 3 first="<<sum_elem(array,array+3)<<endl;
  cout<<"sum of the 4 last="
      <<sum_elem(array+n-4,array+n)<<endl;
  cout<<"sum from 3 to 7 ="<<sum_elem(array+2,array+7) <<endl;
  return 0;
}
```

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

В функции `sum_elem` диапазон элементов задаётся двумя параметрами `begin` и `end`.

```
int sum_elem(const int *begin, const int *end);
```

Перебор элементов диапазона выполняется с помощью цикла `for`.

```
for (pt=begin; pt!=end; ++pt)
```

Условие продолжения цикла `pt!=end` отражает тот факт, что правая граница находится за последним элементом диапазона.

Весь массив также можно рассматривать как диапазон. Тогда левая граница совпадает с адресом массива (его именем), а правую можно определить, добавив к адресу массива количество его элементов.

```
cout<<"sum of all ="<<sum_elem(array, array+n)<<endl;
```

Для диапазона с третьего по седьмой элементы используется следующий вызов.

```
cout<<"sum from 3 to 7 ="<<sum_elem(array+2, array+7) <<endl;
```

Реализация функции `sum_elem` предполагает, что диапазон задан корректно (`begin<=end`). Дополните функцию проверкой условия корректности. В случае ошибочных параметров выбрасывать исключение.

3.4. Массивы и рекурсия

Массивы, как и многие другие структуры данных, можно определить рекурсивно. Поэтому большинство алгоритмов по их обработке легко реализуется через рекурсию. Функцию обработки массива можно представить как обработку одного элемента (первого или последнего) и вызов функции обработки для остальных элементов.

Пример 29. Описать две рекурсивные функции вычисления суммы элементов массива. Использовать оба варианта передачи массива как параметра функции. Продемонстрировать вызовы на разных диапазонах.

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

int summ (int* a, int n) {
  if (n==0)
    return 0;
    return summ(a, n-1)+ a[n-1];
}

int summ(int*bg, int*end) {
  if ( bg == end)
    return 0;
    return summ(bg+1,end) + *bg;
}

int main() {
  int a1[]={3,5,6,2,8,-5};
  int a2[]={9,5,-6,7,3,-3,8,6,2,4};
  cout<<summ(a1,6)<<endl; //все элементы массива
  cout<<summ(a2,9)<<endl; //все элементы массива
  cout<<summ(a1,3)<<endl; //первые три элемента массива
  cout<<summ(a1+3,3)<<endl; //последние три элемента массива

  cout<<summ(a1,a1+6)<<endl;
  cout<<summ(a1,&a1[6])<<endl;
  cout<<summ(a2,a2+9)<<endl;
  cout<<summ(a1+2,a1+2)<<endl;
  return 0;
}
```

В первой реализации перегруженной функции `summ` массив передаётся указателем на начало массива и количеством элементов.

```
int summ (int* a, int n) {
  if (n==0)
    return 0;
    return summ(a, n-1)+ a[n-1];
}
```

Эта реализация демонстрирует вычисление суммы через суммирование последнего элемента с результатом рекурсивного вызова функции для первых `n-1` элементов.

Соответствующие вызовы:

```
cout<<summ(a1,6)<<endl; //все элементы массива
cout<<summ(a2,9)<<endl; //все элементы массива
cout<<summ(a1,3)<<endl; //первые три элемента массива
cout<<summ(a1+3,3)<<endl; //последние три элемента массива
```

Во второй реализации перегруженной функции `summ` массив передаётся двумя указателями, которые определяют начало и конец сегмента.

```
int summ(int*bg, int*end) {
  if ( bg >= end) return 0;
    return summ(bg+1,end) + *bg;
}
```

В этой реализации демонстрируется вычисление суммы через суммирование первого элемента с результатом рекурсивного вызова для оставшихся элементов массива.

Соответствующие вызовы:

```
cout<<summ(a1,a1+6)<<endl; //все элементы массива
cout<<summ(a1,&a1[6])<<endl; //все элементы массива
cout<<summ(a2+3,a2+9)<<endl; //эл-ты с индексами с 3-го по 8-ой
cout<<summ(a1+2,a1+2)<<endl; //вырожденный диапазон, сумма = 0
```

Пример 30. Написать две рекурсивные функции для обработки элементов массива. Первая должна находить максимальный элемент, вторая — считать количество нулевых элементов массива.

```
int max_elem(int* aa, int n) {
  if ( n==1 )
    return aa[0];
  int m = max_elem(aa,n-1);
  return aa[n-1]>m ? aa[n-1] : m;
}

int count0 (int *aa, int n) {
  if ( n==0 )
    return 0;
  if (aa[n-1]==0)
    return 1+count0(aa,n-1);
  return count0(aa,n-1);
}
```

Стоит обратить внимание, что в реализации функции `max_elem` используется локальная переменная `m` для сохранения результата

рекурсивного вызова. Это связано с тем, что попытка обойтись без вспомогательной переменной приведёт к существенному росту количества рекурсивных вызовов.

```
//return aa[n-1]>max_elem(aa,n-1)? aa[n-1]: max_elem(aa,n-1);
```

Хотя в функции `count0` также присутствуют два рекурсивных вызова, но они находятся в разных ветках условного оператора, поэтому увеличения количества рекурсивных вызовов не происходит.

3.5. Статическое определение двумерных массивов

В языках C и C++ многомерный массив определяется как массив массивов. Таким образом определяется массив любой размерности, где каждый элемент, в свою очередь, является массивом. Двумерный массив определяется следующим образом:

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

Следующее объявление означает, что переменная `matrix` — это массив из 3 элементов, каждый из которых — это массив из 2 целых чисел:

```
int matrix[3][2];
```

Для многомерных массивов, так же, как и для одномерных, можно выполнить начальную инициализацию:

```
int matr[3][4]={{1,2,3,4},{5,6,7,8},{9,0,1,2}};
```

При инициализации многомерных массивов можно опускать первую размерность, она определяется автоматически:

```
int arr[][4]={{4,3,2,1},{52,6,17,8},{9,10,1,2}};
```

При описании многомерных массивов как параметров функции также можно первую размерность оставить неопределённой. Остальные размерности должны быть заданы:

```
void ff (int matr[][10], int nn, int mn);
```

Аналогично одномерным массивам объявление `int matr[][10]` можно заменить на объявление `int (*matr)[10]`, что соответствует

указателю на массив, каждый элемент которого — массив из 10 элементов целого типа:

```
void ff (int (*matr)[10], int nn, int mm);
```

Обратите внимание на необходимость скобок, поскольку объявление `int * matr[10]` означало бы массив из 10 указателей типа `int`.

Пример 31. Организовать построчный вывод матрицы целых чисел, с максимально возможным количеством столбцов равным 10. Решение оформить в виде функции.

```
void print_matr (int matr[][10], int nn, int mm) {
  int j;
  for (int i=0; i<nn; ++i) {
    for (j=0; j<mm; ++j)
      cout<< matr[i][j]<<"  ";
    cout<<endl;
  }
}
```

Обязательное требование указания количества столбцов при объявлении статического двумерного массива, в том числе и при описании параметров функции, делает код программы негибким.

Статические массивы любой размерности занимают в памяти непрерывный участок. Размер выделяемого участка вычисляется компилятором исходя из величины каждого измерения. При обращении к элементу массива по индексу его положение в памяти определяется с использованием значений размерностей. При этом имя многомерного массива по-прежнему является адресом массива, а также адресом его первого элемента.

Пример 32. Вывести все элементы матрицы размера `nx2`. Организовать доступ к элементам через указатель (не использовать индексы). Решение оформить в виде функции. Привести примеры вызова для разных матриц.

```
#include <iostream>
using namespace std;
void print_matr_n_2(int b[][2],int n){
  int *c=&b[0][0];
  for (int i=0;i<n*2;++i)
    cout<<*c++<<' ';
  cout<<endl;
}
int main() {
  int a[2][2]= {{1,2},{3,4}};
  int b[3][2]= {{1,2},{3,4},{5,6}};
  print_matr_n_2(a,2);
  print_matr_n_2(b,3);
  return 0;
}
```

Обращение к элементам матрицы организовано через указатель, который смещается на каждом шаге к следующему элементу. Это возможно вследствие линейного расположения матрицы по строкам на непрерывном участке памяти. В этом примере элементы выводятся без разбиения на строки. Количество столбцов учитывается только при вычислении общего количества элементов матрицы.

Пример 33. Написать функцию, находящую в целочисленной матрице индексы первого вхождения элемента с заданным значением. Если элемент есть, функция возвращает «истину», иначе — «ложь».

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

bool findIndices(int a[][5],int n,int m,int Y,int&ki,int&kj){
  int i=0;
  int j=0;
  while (i<n && a[i][j]!=Y) {
    j++;
    if (j==m) {
      j=0;
      i++;
    }
  }
  if (i<n) {
    ki=i;
    kj=j;
  }
  return i<n;
}
```

```
void input(int a[][5], int n, int m) {
  for (int i=0; i<n; ++i)
    for (int j=0;j<m;++j)
      cin>>a[i][j];
}

int main() {
  int matr[5][5];
  int ki,kj,n,m,elem;
  cin>>n>>m;
  input(matr,n,m);
  cin>>elem;
  bool f= findIndices(matr,n,m,elem,ki,kj);
  if (f)
    cout<<ki<<' '<<kj<<endl;
  else
    cout<<"элемент не найден"<<endl;
  return 0;
}
```

В основу алгоритма положено решение, предложенное Д. Грисом [20]. Его особенностью является использование одного цикла с условием. В условии проверяется нахождение элемента с заданным значением и выход за границу матрицы, поскольку элемент с заданным значением может отсутствовать.

```
while (i<n && a[i][j]!=Y)
```

В цикле используется два индекса. Индекс по столбцам увеличивается на каждой итерации, а индекс по строкам увеличивается только при достижении индексом по столбцам максимального значения.

```
if (j==m) {
  j=0;
  i++;
}
```

Для проверки того, что элемент найден, используется условие `i<n` после завершения цикла.

3.6. Двумерные массивы в динамической памяти

Выделение динамической памяти одним оператором `new` возможно только для непрерывного линейного участка. При этом невозможно

передать компилятору информацию о том, что эта память выделяется под двумерный массив. В этом случае, чтобы обратиться к элементу через указание строки и столбца придётся пересчитать линейное смещение элемента относительно начала массива.

Чтобы смоделировать в динамической памяти структуру данных «двумерный массив» можно использовать указатель на массив указателей. Для случая матрицы целых чисел описание указателя будет выглядеть следующим образом:

```
int **a;
```

Выделение динамической памяти при таком описании производится поэтапно. На первом этапе выделяется память под массив указателей на одномерные массивы, соответствующие строкам матрицы:

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

На втором этапе происходит выделение памяти под каждую строку:

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

Модель представления двумерного массива в динамической памяти показана на рисунке 3.1.

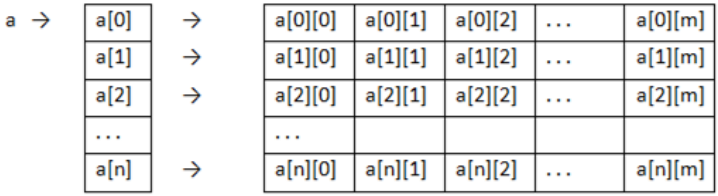

Рис. 3.1. Модель представления двумерного массива в динамической памяти

Освобождение динамической памяти, занятой двумерным массивом, также выполняется в два этапа, но в обратной последовательности:

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

Такой способ представления двумерного массива позволяет создавать массив как со строками одинаковой длины, так и различающейся. Это позволяет экономно использовать память в некоторых случаях, например, для треугольных матриц.

Пример 34. Создать и вывести на печать матрицу следующего вида:

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

Решение позволяет продемонстрировать экономное использование памяти. Будем выделять память только для элементов, расположенных на главной диагонали и выше.

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

В памяти формируется нерегулярный массив, в котором длина каждой следующей строки уменьшается на единицу.

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

 Создать и распечатать нижнюю треугольную матрицу, заполнив её по аналогии с матрицей из примера 34.

Пример 35. Создать набор функций для работы с матрицей в динамической памяти: создание матрицы заданных размеров и заполнение её с клавиатуры; печать матрицы; освобождение динамической памяти, занимаемой матрицей. Продемонстрировать использование функций для решения задачи проверки квадратной матрицы на симметричность.

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

int ** create(int n=5, int m=5);
void create(int **& a,int n=5, int m=5);

void input(int ** a,int n=5, int m=5);
void print(int ** a,int n=5, int m=5);

bool isSymm(int ** a,int n=5);
void free(int **&a,int n=5);

void create(int **& a,int n, int m){
  a=new int *[n];
  for (int i=0; i<n; ++i)
    a[i]=new int[m];
}

int ** create(int n, int m){
  int ** a=new int *[n];
  for (int i=0; i<n; ++i)
    a[i]=new int[m];
  return a;
}
```

```
void input(int ** a,int n, int m){
  for (int j, i=0; i<n; ++i)
    for (j=0; j<m; ++j) cin>>a[i][j];
}

void print(int ** a,int n, int m){
  int j;
  for (int i=0; i<n; ++i) {
    for (j=0; j<m; ++j)
      cout<<a[i][j]<<"  ";
    cout<<endl;
  }
}

bool isSymm(int ** a,int n){
  bool fl=true;
  for (int j, i=0; i<n && fl; ++i){
    for (j=0; j<i && fl; ++j)
      fl=a[i][j]==a[j][i];
  }
  return fl;
}

void free(int **&a,int n){
  for (int i=0; i<n; ++i)
    delete []a[i];
  delete []a;
  a=nullptr;
}


int main() {
  int **a;
  int n;
  cin>>n;
  create(a,n,n);
  input(a,n,n);
  print(a,n,n);
  cout<<isSymm(a,n)<<endl;
  free(a,n);
  a=create();
  input(a);
  print(a);
  cout<<isSymm(a)<<endl;
  free(a);
  return 0;
}
```

Рассмотренная модель двумерного массива в динамической памяти позволяет при передаче его в функцию использовать в качестве параметра указатель на указатель.

```
void create(int **& a,int n=5, int m=5);
void input(int ** a,int n=5, int m=5);
void print(int ** a,int n=5, int m=5);
bool isSymm(int ** a,int n=5);
void free(int **& a,int n=5);
```

Такой способ описания параметра позволяет разрабатывать универсальные функции работы с двумерными массивами. При вызове функции фактическим параметром тоже должен быть указатель на указатель.

```
int **a;
int n;
cin>>n;
…
print(a,n,n);
```

В отличие от одномерных массивов, статические двумерные массивы нельзя передавать в функцию с таким описанием параметра. Чтобы иметь возможность использовать такие функции для статических двумерных массивов необходимы дополнительные действия. Нужно создать вспомогательный массив указателей на строки матрицы. Например, чтобы применить функцию `print` к двумерному статическому массиву `aa[5][3]` выполняем следующие действия:

```
int aa[5][3];
int *p[5];
for (int i=0; i<5; ++i)
     p[i]=&aa[i][0];
print (p,5,3);
```

Рассмотрим функции, в которых параметр `a` (указатель на указатель) передаётся по ссылке:

```
void create(int **& a,int n=5, int m=5);
void free(int **&a,int n=5);
```

Необходимость передачи по ссылке вызвана тем, что значение указателя при выделении/освобождении памяти в функции меняется.

В функции `free` при освобождении памяти, занимаемой двумерным массивом, требуется информация только о количестве строк, что позволяет передавать в функцию только одну размерность.

```
void free(int **& a,int n){
  for (int i=0; i<n; ++i)
    delete []a[i];
  delete []a;
  a=nullptr;
}
```

Язык C++ позволяет возвращать адрес динамически создаваемого массива, что продемонстрировано в перегруженной функции `create`

```
int ** create(int n, int m){
  int ** a=new int *[n];
  for (int i=0; i<n; ++i)
    a[i]=new int[m];
  return a;
}
```

Для второго варианта вызов функции `create` выглядит проще и семантически более понятно.

```
int **ab; int nn;
cin>>nn;
ab=create(nn,nn);
```

☺ В языке C++ рекомендуется вместо передачи по ссылке параметра типа указатель использовать возвращение функцией значения типа указатель.

3.7. Алгоритмы сортировок для массивов

Существует несколько критериев, по которым оценивается эффективность алгоритмов сортировок. К ним относятся экономия памяти и быстродействие алгоритма. Требование экономии памяти выполняется в случае переупорядочивания массива на месте, т. е. без использования вспомогательного. На быстродействие алгоритма влияет количество выполняемых операций, а именно, сравнений элементов и их перестановок.

Особенно важно учитывать эффективность алгоритмов сортировок для массивов больших размеров.

С точки зрения быстродействия различают базовые и улучшенные алгоритмы сортировок [13]. К базовым относят три метода, требующих порядка n^2 операций. Оценка быстродействия улучшенных методов стремится к $O(n \cdot \log_2 n)$.

Пример 36. Для каждого из базовых алгоритмов сортировки (выбором, обменом, включением) реализовать соответствующие функции.

Рассмотрим алгоритм сортировки выбором, в котором производится многократный выбор экстремального элемента (либо всегда максимального, либо всегда минимального). Выбранный элемент перемещается в ту позицию, которую он будет занимать в упорядоченном массиве.

```
void select_Sort (double * aa, int size) {
  int min;
  for (int i =0; i < size-1; ++i) {
    min = i ;
    for (int j  = i+1; j < size; ++j){
      if (aa[ j ] < aa[min])
         min = j;
    }
    swap (aa[ i ], aa[ min ]);
  }
}
```

Алгоритм сортировки обменом называют ещё методом «пузырька». В нём производится обмен значениями всех пар рядом стоящих элементов, которые нарушают условие упорядоченности. Процесс обмена прекращается, когда все пары окажутся упорядоченными.

```
void bubble_Sort (double * aa, int size) {
  bool flag = true;
  int i = size-1;
  int j;
  while (flag) {
    flag = false;
    for (j  = 0; j < i; ++j)
      if (aa[ j ] > aa[j+1]) {
```

```
        swap (aa[ j ], aa[j+1]);
        flag = true;
      }
      i = j;
    }
}
```

Сортировка включением возможна как с использованием вспомогательного массива, так и без. Мы рассмотрим второй вариант. Обе версии алгоритма используют включение одного элемента в упорядоченный массив, сохраняя упорядоченность массива. Массив, состоящий из одного элемента, является упорядоченным.

```
void insert_Sort (double * aa, int size) {
  double xx; int j;
  for (int i = 1; i <size; ++i) {
    xx = aa[i];
    j = i-1;
    while (j > =0 && aa[ j ] > xx) {
      aa[j+1] = aa[j];
      j --;
    }
    aa[j+1] = xx;
  }
}
```

Реализуем перегрузку функции `insert_Sort` для случая задания массива через диапазон.

```
void insert_Sort (double * beg, double * end) {
  double xx; double *dd;
  for (double *cc=beg+1; cc<end; ++cc) {
    xx =*cc;
    dd = cc-1;
    while (dd >=beg && *dd > xx) {
      *(dd+1) = *dd;
      dd--;
    }
    *(dd+1) = xx;
  }
}
```

Пример 37. Реализовать быструю сортировку (сортировку Хоара).

Быстрая сортировка относится к улучшенным методам сортировки. Она реализуется на базе сортировки обменом. Для повышения эффективности А. Хоар предложил выполнять обмены элементов,

отстоящих друг от друга на больших расстояниях [13]. Цель таких обменов — разделить массив на две части таким образом, чтобы все элементы левой части не превышали элементы правой. В алгоритме перестановки происходят относительно опорного элемента, который обычно соответствует центральному элементу массива. Алгоритм рекурсивно применяется к каждой из частей до тех пор, пока её размер больше единицы.

```
void quickSort(double*aa, int low, int high) {
  int i = low, j = high;
  double xx = aa[(low+high)/2];  //xx - опорный
  do {
    while(aa[i] < xx) ++i;  //поиск элемента слева для обмена
    while(aa[j] > xx) --j;  //поиск элемента справа для обмена
    if (i <= j){
      swap(aa[i],aa[j]);   //обмен элементов местами
      ++i; --j;            //переход к следующим элементам:
    }
  } while(i < j);
  if (low < j) quickSort(aa, low, j);
  if (i < high) quickSort(aa, i, high);
}
```

Реализуем перегрузку функции `quickSort` для случая задания массива через диапазон.

```
void quickSort(double*beg, double *end) {
  double xx = *(beg+(end-beg)/2);  // xx - опорный
  double *p=beg;
  double *q=end-1;
  do {
    while(*p < xx) ++p;  //поиск элемента слева для обмена
    while(*q > xx) --q;  //поиск элемента справа для обмена
    if (p <= q){
      swap(*p,*q);      //обмен элементов местами
      ++p; --q;         //переход к следующим элементам:
    }
  } while(p < q);
  if (beg < q)  quickSort(beg, q+1);
  if (p < end-1) quickSort(p, end);
}
```

Рекурсивная реализация увеличивает накладные расходы на вызовы функции. Поэтому для массивов небольших размеров она будет проигрывать в скорости по сравнению с реализацией базовых методов. Хоар

разработал алгоритм именно для перестановок на больших расстояниях, т. е. для массивов больших размеров.

Поэтому на практике рекомендуется комбинировать быструю сортировку с одной из базовых. Если размер сортируемого сегмента становится меньше некоторой границы, то для этого сегмента рекурсия останавливается и для него выполняется одна из базовых сортировок. Например, в реализации функции `qSortWCut` используется сортировка включением.

```
void qSortWCut(double*beg, double *end) {
  if (end-beg<CUTOFF) {
    insertionSort(beg,end);
    return;
  }

  double xx = *(beg+(end-beg)/2);  // xx - опорный
  double *p=beg;
  double *q=end-1;
  do {
    while(*p < xx) ++p;  //поиск элемента слева для обмена
    while(*q > xx) --q;  //поиск элемента справа для обмена
    if (p < q){
      swap(*p,*q);       //обмен элементов местами
      ++p; --q;          //переход к следующим элементам:
    }
  } while(p < q);
  if (beg < q) qSortWCut(beg, q+1);
  if (p < end-1) qSortWCut(p, end);
}
```

Условие `(end-beg<CUTOFF)` используется для прекращения рекурсии и перехода к простой сортировке.

Пример 38. Упорядочить строки матрицы по возрастанию сумм их элементов с использованием индексной сортировки.

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

void indexSort(int** matr, int n, int m, int* ind);

int main() {
  setlocale(LC_ALL, "RU");
```

```
  int **matr;
  int n, m;
  int* index;

  cout << "Введите размер матрицы ";
  cin >> n >> m;

  matr = new int* [n];
  for (int i = 0; i < n; ++i) {
    cout << "Введите строку с номером " << i << " ";
    matr[i] = new int[m];
    for (int j = 0; j < m; ++j) {
      cin >> matr[i][j];
    }
  }

  index = new int[n];
  indexSort(matr, n, m, index);

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

  delete[] index;
  for (int i = 0; i < n; ++i)
    delete[] matr[i];
  delete[] matr;
  return 0;
}

void indexSort(int** aa, int n, int m, int* ind) {
  int* sum = new int[n]
  for (int i = 0; i < n; ++i) {
    ind[i] = i;
    sum[i] = 0;
    for (int j = 0; j < m; ++j)
      sum[i] += aa[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--;
      }
}
```

Использование вспомогательного массива индексов позволяет повысить эффективность алгоритма за счёт уменьшения количества обменов. Вместо того, чтобы менять местами строки матрицы, меняем соответствующие элементы в массиве индексов.

Полученный в результате сортировки массив индексов отражает размещение строк в отсортированной матрице. Такая сортировка называется индексной.

Обычно для индексной сортировки вводится не только массив индексов, но и массив ключей. Он содержит значения, по которым выполняется сортировка. Массив ключей может отсутствовать в том случае, если ключи хранятся в исходном массиве. Алгоритм сортировки основывается на сравнении ключей, доступ к которым происходит через индекс, и перестановке индексов. Доступ к отсортированным элементам массива возможен только через массив индексов.

Индексная сортировка реализована в виде функции

```
void indexSort(int** aa, int n, int m, int* ind)
```

В качестве параметров передаются матрица, размещённая в динамической памяти, её размеры и выходной параметр `int*ind` для массива индексов. Внутри функции создаётся массив ключей.

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

Поскольку сортировка выполняется по суммам элементов строк, то они используются в качестве ключей индексной сортировки.

По окончании работы функции `indexSort` исходная матрица и массив ключей остаются неизменными, меняется только массив индексов.

Чтобы воспользоваться результатами сортировки, необходимо обращаться к элементам матрицы по индексам, в качестве которых используются элементы изменённого индексного массива:

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

В приведённом примере для форматированного вывода элементов матрицы используется манипулятор `setw`, устанавливающий ширину поля вывода.

Пример 39. 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--;
  }
}
```

В данном примере также используется индексная сортировка. При этом массив ключей не создаётся, так как ключами являются элементы матрицы. Отметим, что при решении задачи используется особенность представления двумерного массива как массива указателей на одномерные массивы. В этом случае роль индексного массива играет массив указателей на строки (рис.3.2).

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

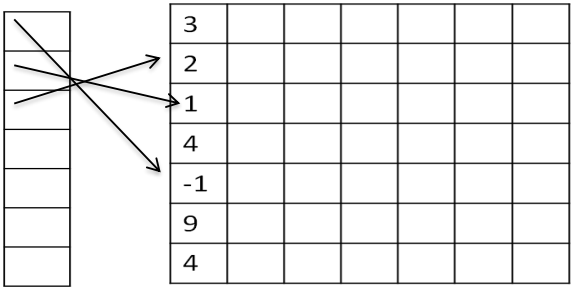

Рис.3.2. Перестановка указателей на строки матрицы.

Такой приём значительно ускоряет время работы алгоритма и упрощает обращение к элементам отсортированной матрицы.

3.8. Сложные объявления

Объявления в C++ могут быть перенасыщены скобками и их сложно читать. Например, в следующем объявлении `pArrPChar` — это функция, возвращающая указатель на массив из указателей на функции возвращающие `char`:

```
char (*(*pArrPChar ())[])();
```

Столкнувшись с подобным сложным объявлением, лучше всего его разбор начинать с середины и постепенно двигаться наружу, анализируя по очереди символы, то справа, то слева [7]. Рассмотрим вначале в качестве примера более простое объявление:

```
int (*pf)(); // *pf; указатель на функцию, возвращающую int
```

«Серединой» в данном случае является имя переменной, то есть `pf`. Сначала нужно посмотреть направо, где ничего нет (закрывающая круглая скобка завершает выражение). Затем смотрим налево, где находятся символ «звёздочка» (признак указателя) и завершающая скобка; затем снова направо (пустой список аргументов обозначает функцию, которая вызывается без аргументов) и снова налево (ключевое слово `int` указывает на то, что функция возвращает значение целого типа).

☺ Используйте для прочтения сложных объявлений приём «движения изнутри наружу чередуя направо и налево» [7].

Например, следующие объявления можно прочесть так:

```
void *(*(*fp1)(int))[10];
```

`fp1` — указатель на функцию, которая получает аргумент типа `int` и возвращает указатель на массив из 10 указателей типа `void`;

```
float (*(*fp2)(int,int,float))(int);
```

fp2 — указатель на функцию, которая получает три аргумента (int, int и float) и возвращает указатель на функцию, получающую аргумент int и возвращающую float;

```
int (*(*fp3())[10])();
```

fp3 — функция, которая возвращает указатель на массив из 10 указателей на функции, возвращающие значения типа int.

☺ Если необходимо создавать много сложных объявлений, рекомендуется использовать конструкцию using или typedef.

Например, объявление

```
using fp4 = double (*(*(*)())[10])();
fp4 a,b,c;
```

означает следующее: fp4 — тип указателя на функцию, которая вызывается без аргументов и возвращает указатель на массив из 10 указателей на функции, вызываемые без аргументов и возвращающие double. Переменные a,b,c относятся к типу fp4.

Проанализировать следующие объявления:

```
char ** argv;
//argv: указатель на указатель на char
int (* daytab)[13];
//daytab: указатель на массив [13] из int
void *comp();
//comp: функция, возвращающая указатель на void
void (*comp)();
//comp: указатель на функцию, возвращающую void
char (*(*x[3])())[5];
//x: массив [3] из указателей на функцию, возвращающую
указатель на массив [5] из char
```

3.9. 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.

Размер массива должен быть хотя бы на единицу больше количества символов в строке, определяемой этим массивом, потому что 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";
```

При этом безымянная константная строка `"String"` используется для извлечения из неё символов при инициализации массива. К самой этой строке доступа нет.

Следующий пример демонстрирует типичную ошибку, возникающую при объявлении строки:

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

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

Чтобы избежать этой ошибки, рекомендуется не указывать размер массива при объявлении строки:

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

При таком способе объявления размер массива определяется по фактическим данным, которыми он инициализируется.

Эквивалентным объявлением, демонстрирующим внутреннее представление строки, является явная инициализация элементов массива символами:

```
char s3_1[] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
```

Под каждый из массивов `s3` и `s3_1` при этом отводится память из семи элементов типа `char`.

> Поскольку такие строки представляются массивами, имя строки является указателем на первый элемент и правила относительно указателей справедливы.

Иногда в реальных программах можно встретить не вполне корректную инициализацию:

```
char* s4 = "String";
```

Если в случае `s3` объявляется константный указатель на массив, то объявленный указатель `s4` ссылается на константную строку.

Значения элементов массива `s3` можно изменять, а значение указателя `s3` — нет. Значение указателя `s4` изменять можно, а значения элементов массива `s4` — нет. Эти особенности продемонстрированы в 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;
}
```

Если в случае ошибки компиляции

```
//cout<<*(++s3)<<endl; – ошибка компиляции
```

необходимо исправить её для получения работающего кода, то в случае попытки некорректной работы с указателем s4 ошибки компиляции не возникает. А значит мы получаем откомпилированную неработающую программу.

```
//s4[1]='A'; – ошибка времени выполнения
```

Чтобы защитить программу от таких ошибок, рекомендуется использовать следующее объявление

```
const char * s4 = "String";
```

При таком объявлении оператор

```
s4[1]='A';
```

будет вызывать ошибку компиляции

```
's4': you cannot assign to a variable that is const
```

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, а значит не содержит символа конца строки.

Распространённая 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 распространённые 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 приведённом 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 ним 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 использования класса `string`:

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

int main() {
  string s1,s2;  //Пустые строки
  string s3 = "Hello!";  //Инициализированная строка
  string s4("I am");  //Ещё один пример инициализации
  string s5(s3);  //И ещё 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 существующим объектом `s3`.

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

```
s2 = "Ву";
```

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++ присваивает строки, а в стиле С — только указатели.

Стандартный класс С++ `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 С++ существенно снижают вероятность самых распространённых и опасных ошибок программирования в стиле языке С: 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 С 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 начнёт изменяться. Тогда для него будет выделен новый блок памяти. Такая возможность реализуется посредством подсчёта 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 неё заголовочный файл C++ string. Класс `string` определён в пространстве имён `std`, поэтому в программу включается директива `using`.

3.10. Обработка строк в стиле языка С

Поскольку строка в стиле С завершается нулевым символом, её обработка имеет определённую специфику. Рассмотрим её на примере копирования строк.

Пример 40. Реализовать функцию копирования заданной строки `s2` в строку `s1`.

Рассмотрим вначале несколько способов копирования без привлечения библиотечных функций.

Воспользуемся тем, что строка — это массив символов. Следующий цикл производит посимвольное копирование из строки `s1` в строку `s2` до

того момента, как в строке встретится нулевой символ. После цикла нулевой символ дописывается в конец строки `s2`.

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

Учитывая, что число `0` неявно преобразуется в символ `'\0'`, можно заменить символ `'\0'` нулём.

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

Поскольку в языке С тип `char` относится к целочисленным типам, для которых любое ненулевое значение считается истиной (`true`), выражение `s1[i]!=0` для проверки условия в цикле можно упростить:

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

Посимвольное копирование можно реализовать, 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 приведём полный текст программы с использованием этой функции.

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

char * copy(char *p, const char *q) {
  char *pp=p;
  while (*pp++=*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;
  cout <<copy(s3,s1) << endl;
  cout <<"s3=" << s3 << endl;
}
```

Две последние строки функции `main` выводят на экран один и тот же результат — значение переменной `s3`.

Все рассмотренные варианты 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, начиная со стандарта 1998 г., в cstring).

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

☠ Распространённая ошибка: применение в качестве параметров 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 41. Реализовать функцию для нахождения максимальной из двух строк.

```
#include <iostream>
#include <cstring>
using namespace std;
char* maxs(char* a,  char* b) {
  if (strcmp(a, b)>0)
    return a;
  else
    return b;
}
int main() {
  char s1[] = "qwerty";
  char s2[] = "asd";
  char * sMax;
  cout << maxs("Hello", "By") << endl;
  cout << maxs(s1, s2) << endl;
  sMax = maxs(s1, s2);
  return 0;
}
```

Функция `maxs` использует функцию `strcmp` из стандартной библиотеки cstring:

```
strcmp(const char*string1, const char*string2)
```

Строки сравниваются в *лексикографическом* порядке. Результат сравнения определяется по правилам, приведённым в таблице 4. Для использования данной функции подключается заголовочный файл cstring (string.h).

Таблица 4

Результат сравнения строк

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

Пример 42. Вычислить количество вхождений строки `s2` в строку `s1`, используя стандартные функции `strstr`, `strlen` библиотеки cstring.

```
int count(char *s1,char *s2){
  int c = 0; int len = strlen(s2);
  char *p = strstr(s1, s2);
  while (p){
    c++;
    p = strstr(p + len, s2);
  }
  return c;
}
```

Функция `strstr(s1, s2)` возвращает указатель на первое вхождение строки `s2` в строку `s1` или нулевой указатель, если вхождение строки не обнаружено. Чтобы каждый раз искать новое вхождение `s2`, необходимо вызывать функцию `strstr` не для `s1`, а я для её части. При этом в конце каждого шага цикла `p` необходимо сдвигать на длину строки `s2`, чтобы не происходило зацикливания:

```
p = strstr(p + len, s2);
```

Пример 43. Найти индекс последнего вхождения символа `ch` в строке `s`, используя функцию `strchr`.

```
int last_ind(char *s, char ch){
  char *p = strchr(s,ch);
  int ind = -1;
  while (p) {
    ind = p - s;
    p = strchr(p+1, ch);
  }
  return ind;
}
```

В функции `last_ind` использована возможность определения индекса через разность указатели. Этот приём можно использовать для любых массивов.

Пример 44. С помощью рекурсивной функции выдать символы строки в обратном порядке.

```
void reverse (const char* s) {
  if (*s) {
    reverse(s+1);
```

```
      cout <<*s;
  }
}
int main() {
  char s1[] = "qwerty";
  reverse("Hello");cout << endl;
  reverse(s1);
  cout << endl << s1 << endl;
}
```

Пример 45. Поменять порядок символов в строке на противоположный. Для этого определить указатель на последний символ (оформить в виде отдельной функции), и, перемещая два указателя от начала и конца строки к середине, менять местами соответствующие символы.

```
char* findLastChar(char* str){
  if (!*str) return 0;
  while (*str)
    str++;
  return str-1;
}
void reverseString(char* str) {
  if (!*str) return;
  char *last = findLastChar(str);
  while (str < last) {
    swap(*str,*last);
    str++;
    last--;
  }
}
```

Если строка пустая, то функция `findLastChar` возвращает нулевой указатель. Аналогичная проверка выполняется и в начале функции `reverseString`, чтобы избежать выполнения лишних действий.

Сравнение указателей `str<last` является допустимым, т. к. они указывают на область, соответствующую одной и той же строке.

Пример 46. Дана строка, состоящая из слов, разделённых одним или несколькими пробелами. В начале и в конце строки могут находиться начальные и конечные пробелы. Требуется удалить лишние пробелы

(оставить только по одному между словами), поменять местами чётные и нечётные слова и записать результат в новую строку.

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

В данном алгоритме будут 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) {
  const char *p1;
  while (*p == ' ') p++; //пропустить начальные пробелы
  do {
    p1= p;//запомнить адрес нечётного слова
    while (*p != ' ' && *p) p++;   //пропустить нечётное слово
    while (*p == ' ') p++;    //пропустить пробелы
    if (*p) {  //если есть чётное слово
      while (*p != ' ' && *p)
        *ps++=*p++; //копировать чётное слово
      *ps++ = ' ';  //добавить пробел
    }
    while (*p1 != ' ' && *p1)
      *ps++=*p1++;  //копировать нечётное слово
    *ps++ = ' ';    //добавить пробел
    while (*p == ' ') p++;    //пропустить пробелы
  } while (*p);
  *--ps=0;     //удалить лишний пробел и добавить 0
}

int main() {
  char s1[100], char *s3=new char [100];
  s2[100]="  Hello!   1111111111    2222222222222  ";
  swap_even(s2,s1);
  cout <<"s1=" << s1 << endl;
  cout <<"s2=" << s2 << endl;
  swap_even(s1,s3);
  cout <<"s3=" << s3 << endl;
}
```

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

Пример 47. Описать функции `TrimLeftC(s,c)` и `TrimRightC(s,s)`, удаляющие в строке `s` соответственно начальные и концевые символы, совпадающие с символом `c`. Поскольку очень часто требуется удалить начальные пробелы, то параметр `c` сделать параметром по умолчанию (по умолчанию значение равно символу пробела).

Так как строка — это массив символов, то задачи удаления/вставки символов могут быть решены или с помощью сдвига, или созданием новой строки, содержащей изменённые данные.

В силу особенностей представления строк в стиле C, возможны другие варианты решения. Например, удаление концевых (правых) символов решается переносом признака конца строки.

```
void TrimRightC(char* s, char c=' ') {
  char *p=s;
  while (*p!=0)
    p++;
  if (p==s) return;
  --p;
  while (p!=s && *p==c)
    p--;
  p++;
  *p=0;
  return;
}
```

Алгоритм решения состоит из трех частей. Вначале выполняется поиск конца строки. Далее от найденной позиции конца строки в обратном направлении ищется первый символ, отличный от символа `c`. Затем признак конца строки устанавливается в позицию за найденным символом.

После нахождения позиции конца строки выполняется проверка на пустоту строки.

```
  if (p==s) return;
```

На этапе поиска в обратном направлении дополнительно проверяется выход за левую границу строки. Это вызвано тем, что строка может состоять только из одних символов `c`.

```
  while (p!=s && *p==c)
    p--;
```

Удаление начальных (левых) символов решается переносом указателя на начало строки. При этом указатель на исходную строку сохраняется, т. е. сама строка остаётся без изменений. Функция возвращает новый указатель на часть строки, не содержащей начальных символов `c`.

```
char* TrimLeftC(char* s, char c=' ') {
    while (*s!=0 && *s==c)
    s++;
  return s;
}
```

Такая реализация удаления начальных символов является быстрой и эффективной, но допустима только в случае, если дальнейшая работа с исходной и результирующей строками не требует ни изменения содержимого, ни удаления строки с данными. Это связано с тем, что в памяти размещена одна строка данных, а работаем с двумя указателями на эту строку. Для иллюстрации рассмотрим фрагмент программы, содержащий инициализацию данных, вызов функции и вывод результатов.

```
  char *str=new char[101];
  char *res;
  cout<<endl<<"input string ";
  cin.getline(str,100);
  res=TrimLeftC(str);
  cout<<endl<<"origin string ="<<str<<endl;
  cout<<"result string ="<<res<<endl;
```

На рисунке 3.3 продемонстрирован результат вызова функции `TrimLeftC` для входной строки «□□HELLO□□» с точки зрения размещения в памяти.

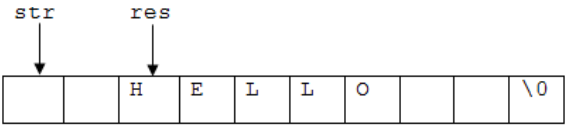

Рис. 3.3. Результат выполнения

Так как исходная строка размещается в динамической памяти, то в какой-то момент память необходимо освободить:

```
delete[] str;
```

После освобождения памяти использование `res` становится некорректным. Указатель `res` ссылался на ту же самую область памяти, которая теперь является недоступной.

Функция `TrimLeftCInPlace` реализует алгоритм удаления ведущих символов путём сдвига влево. Этот вариант работает дольше, но решение полностью соответствует постановке задачи.

```
void TrimLeftCInPlace(char* S, char C = ' '){
  char *p = S;
  while (*p != 0 && *p == C)
    p++;
  while (*S++ = *p++);
}
```

3.11. Обработка строк в стиле языка C++

Класс `string` имеет обширный набор методов для работы со строками, позволяющих выполнять присваивание, конкатенацию, сравнение строк, доступ к отдельным элементам, поиск, замену, удаление и т. п.

Для строк в стиле C и C++ реализованы простые механизмы преобразования из одного типа в другой. Любой строковый литерал является строкой в стиле C. Поэтому в случае присваивания литерала переменной типа `string` происходит неявное преобразование:

```
string s0="Привет!";
char s1[]="Hello!";
string s2=s1;
```

Преобразовать строку в стиле C в тип `string` можно также явно вызвав конструктор:

```
string s3(s1);
string s4("world");
```

Во всех методах и операциях класса `string` в качестве параметров для строк можно использовать оба вида строк и в стиле C и в стиле C++:

```
string ident1 ("max");
string ident2 ("min");
char ident3[]="sum";
…
if (ident1<ident2) ident1.append("less");
if (ident1==ident3) ident1.append("equal");
if ("avg"!=ident2) ident2.append("not equal");
```

Для преобразования строк из типа `string` в тип `char*` используется метод `c_str()` класса `string`, который возвращает указатель типа `const char *` на строку, содержащую те же символы, что и строка типа `string`.

Необходимость такого преобразования возникает крайне редко только в тех случаях, когда требуется использовать для обработки строк функции библиотеки cstring.

```
string s1="C++";
const char *s2;
s2 = s1.c_str();
```

☺ Рекомендуется не смешивать использование строк в стиле C и строк в стиле C++.

Пример 48. 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
#pragma once
#include <string>

void replaceAll (string &context, const string &from,
                 const string & to);

//ReplaceAll.cpp
#include "ReplaceAll.h"
using namespace std;
void replaceAll (string & context, const string & from,
                 const 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();
  }
}
```

Версия `find()`, использованная в этой программе, получает во втором аргументе начальную позицию поиска. Если вхождение подстроки не найдено, то возвращается значение `string::npos`. Переменная `npos` является статическим элементом класса `string`. Её значение равно максимально возможному количеству символов в строковом объекте. Это значение используется как признак неудачного завершения поиска.

Поиск следующего вхождения `from` в `context` необходимо начинать с позиции, следующей за позицией произведённой 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 надёжна, из неё практически полностью устраняются «нетворческие» операции — отслеживание границ памяти, в которой хранятся строковые данные.

Пример 49. Описать функцию, классифицирующую строку как идентификатор, целое число, число с плавающей точкой или неопределённую лексему.

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

enum kind{ empty, ident, integer, floating, error };

const string lower("abcdefghijklmnopqrstuvwxyz");
const string upper("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
const string letters = lower + upper + "_";
const string digits = "0123456789";
const string identchars = letters + digits;

kind classify(const string& s) {
  if (s.empty())
    return empty;
  if (letters.find_first_of(s[0]) != string::npos) {
  //буква
  // проверяем идентификатор
    if (s.find_first_not_of(identchars, 1) == string::npos)
      //только допустимые символы
      return ident;
    else
      return error;
  }
```

```
  // остались числа
  string::size_type pos; //позиция цифр
  if (s[0] == '+' || s[0] == '-')
    pos = 1;
  else
    pos = 0;
  if (pos == s.length())
    return error;
  if (digits.find_first_of(s[pos]) == string::npos)
    //число должно начинаться с цифры
    return error;
  //конец цепочки цифр
  pos = s.find_first_not_of(digits, pos);
  if (pos == string::npos)
    //одни цифры - целое число
    return integer;
  else if (s[pos] == '.') {
    //конец цепочки цифр
    pos = s.find_first_not_of(digits, pos + 1);
    if (pos == string::npos)
      return floating;
  }

  //может быть экспонента
  if (s[pos] == 'e' || s[pos] == 'E') {
    if (pos == string::npos)
      // символ E последний
      return error;
    if (s[pos+1] == '+' || s[pos+1] == '-')
      //пропускаем
      ++pos;
    if (pos == s.length() - 1)
      //знак - последний символ строки
      return error;
    pos = s.find_first_not_of(digits, pos + 1);
    if (pos == string::npos)
      //конец цепочки цифр
      return floating;
  }
  return error;
}

void print(kind t) {
  switch (t) {
  case empty: cout << "empty"; break;
  case integer: cout << "integer"; break;
  case floating: cout << "float"; break;
  case ident: cout << "ident"; break;
  case error: cout << "error"; break;
  }
```

```
  cout << endl;
}

int main(){
  print(classify("1234"));
  print(classify("12.34"));
  print(classify("-1234"));
  print(classify("asd1234"));
  print(classify("12e34"));
  print(classify("-12e+34"));
  print(classify("12e--34"));
  print(classify("1234asd"));
  return 0;
}
```

В программе использованы функции `find_first_of` и `find_first_not_of`. Первым параметром каждой из них является строка, задающая множество символов. Функция `find_first_of` ищет в исходной строке первый символ, принадлежащий множеству, а функция `find_first_not_of` ищет первый символ, не принадлежащий множеству. Если поиск завершается неудачно, возвращается `string::npos`.

ГЛАВА 4. ПАРАМЕТРЫ И ТИПЫ

Зачастую при реализации какого-либо алгоритма хочется, чтобы один и тот же код работал для разных типов элементов и в целом не зависел от их типа. А иногда желательно, чтобы функция могла применять к элементам массива или элементам других наборов данных различные функции. Для этого функция должна принимать в качестве параметра или параметров указатель на функцию, то есть параметр функционального типа.

4.1. Параметры функционального типа

С параметрами функционального типа мы уже встречались в разделе 2.5 на примере построения таблицы значений для произвольной вещественной функции `f(x)`. Такой подход оказывается очень удобным для простейшей реализации обобщённого подхода к обработке элементов массива или других наборов данных.

Пример 50. Описать функции вычисления суммы и максимума значений тех элементов целочисленного массива, которые удовлетворяют заданному условию.

```
#include <climits>
#include <cmath>
#include <iostream>
using namespace std;
typedef bool (* pred1)(int);
typedef bool (* pred2)(int,int);

bool pos (int a) {
  return a>0;
}
bool isSimple (int x) {
  bool f=true;
  for (int i=2; i<sqrt((double)abs(x)) && f;++i)
    f= x%i != 0;
  return f;
}
bool count_dig(int a,int b) {
  int c = a==0 ? 1 : 0;
  while (a) {
```

```
    c++;
    a/=10;
  }
  return c==b;
}
bool last_dig(int a,int b) {
  return abs(a%10)==b;
}

int sum (int *a, int n, pred1 f) {
  int s=0;
  for (int i=0;i<n;i++)
    if (f(a[i]))
      s+=a[i];
  return s;
}

int sum (int *a, int n, pred2 f, int b) {
  int s=0;
  for (int i=0;i<n;i++)
    if (f(a[i],b))
      s+=a[i];
  return s;
}
int maxx(int *a, int n, pred1 f) {
  int m=INT_MIN;
  for (int i=0;i<n;i++)
    if (f(a[i]) && a[i]>m)
      m=a[i];
  return m;
}

int main() {
  int a[]={-5,-95,-75,1,4,2,-5,-9,17,-15,24,2};
  cout<<"sum last_dig "<<sum(a,12,last_dig,5)<<endl;
  cout<<"sum count_dig "<<sum(a,12,count_dig,1)<<endl;
  cout<<"sum isSimple "<<sum(a,12,isSimple)<<endl;
  cout<<"sum pos "<<sum(a,12,pos)<<endl;
  cout<<"max isSimple "<<maxx(a,12,isSimple)<<endl;
  cout<<"max pos "<<maxx(a,12,pos)<<endl;
  return 0;
}
```

Два введённых функциональных типа `pred1` и `pred2` определяют типы для функций-предикатов с одним или двумя параметрами соответственно.

```
typedef bool (* pred1)(int);
typedef bool (* pred2)(int,int);
```

Первому типу соответствуют функции-предикаты `pos` и `isSimple`, второму — `count_dig` и `last_dig`.

Функция `maxx` реализует алгоритм нахождения максимального элемента в массиве, удовлетворяющего заданному условию. Поскольку условие определяется как параметр-предикат, то при вызове функции можно использовать любые условия, оформленные как соответствующие функции.

```
cout<<"max isSimple "<<maxx(a,12,isSimple)<<endl;
cout<<"max pos "<<maxx(a,12,pos)<<endl;
```

Алгоритм суммирования по условию реализован двумя перегруженными функциями с одноместным или двуместным предикатом соответственно.

```
cout<<"sum last_dig "<<sum(a,12,last_dig,5)<<endl;
cout<<"sum count_dig "<<sum(a,12,count_dig,1)<<endl;
cout<<"sum isSimple "<<sum(a,12,isSimple)<<endl;
cout<<"sum pos "<<sum(a,12,pos)<<endl;
```

Использование указателей на функции-предикаты демонстрирует пример функционального программирования.

Пример 51. 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)` передаётся 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

```
#pragma once

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);
using pf= int (*) (int[], int);
```

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

В заголовочном файле используется определение типа указателя на функцию.

```
using pf= int (*) (int[], int);
```

Параметр такого типа передаётся 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;
```

```
}
```

Использование объявленных функций в качестве параметров — файл main.cpp

```
#include "funcs.h"
#include <iostream>
using namespace std;
int main(){
  const int nmax = 100;
  int a[nmax], n;
  input(a, n, nmax);
  int s;
  pf pointf;

  Accumulate(min, s, a, n);
  cout << "min = " << s << endl;
  pointf = min;
  Accumulate(pointf, s, a, n);
  cout << "min = " << s << endl;

  Accumulate(mult, s, a, n);
  cout << "mult = " << s << endl;
  pointf = mult;
  Accumulate(pointf, s, a, n);
  cout << " mult = " << s << endl;

  Accumulate(sum, s, a, n);
  cout << "sum = " << s << endl;
  pointf = sum;
  Accumulate(pointf, s, a, n);
  cout << " sum = " << 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()`.

4.2. Массивы указателей на функции

Иногда возникают задачи, в которых необходимо в качестве фактических параметров функционального типа использовать большое

количество функций. Поскольку указатель на функцию является типом, можно организовать массив элементов такого типа.

Стоит особо остановиться на такой конструкции, как *массив указателей на функции*. Этот механизм воплощает концепцию *кода, управляемого таблицами*. Определение какая из функций будет выполняться осуществляется через переменную состояния (например, индекс функции в массиве) без использования конструкции выбора. Такой приём особенно удобен при частом добавлении или удалении функций из таблицы, а также при динамическом создании или изменении таблицы.

Пример 52. Продемонстрировать использование массива указателей на функции, реализованные в примере 71.

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

int main(){
  const int nmax = 100;
  int a[nmax], n;
  input(a, n, nmax);
  int s;
  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;
}
```

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

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

4.3. Шаблоны функций

Перегруженные функции обычно используются для выполнения похожих по синтаксису, 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 C 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 серьёзные побочные эффекты. Шаблоны же позволяют полностью контролировать соответствие типов.

Пример 53. Реализовать и использовать шаблон функции для вывода элементов массива.

```
#include <iostream>
using namespace std;
template <typename T>    //можно 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>    //можно template <class 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» [16].

В шаблоне функции `printArray` объявляется один параметр шаблона `T` для типа элементов массива, выводимого функцией `printArray`.

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

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

 Каждый параметр типа 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 создаёт 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` идёт объявление шаблона функции, в котором его аргументы указаны явно.

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

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

Пример 54. Реализовать функции нахождения максимума из двух параметров следующих типов данных: `int`, `double`, `char*`, `date`.

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

template <typename T>
T maxx(const T a, const T b) {
  return a>b ? a : b;
}

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

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

date maxx(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 << maxx(3, 2) << endl;
  cout << maxx(3.5, 20.4) << endl;
  cout << maxx("Hello", "By") << endl;
  date a = { 27, 03, 2015 }, b = { 31, 01, 2008 }, c;
  c = maxx(a, b);
  cout << c.day << '.' << c.month << '.' << c.year << endl;
  return 0;
}
```

Шаблон функции необходим, когда описываемый алгоритм может быть применён к данным различных типов. Поэтому шаблон функции `T maxx(const T a, const T b)` можно использовать для создания шаблонных функций для типов `int` и `double`.

```
cout << maxx(3, 2) << endl;
cout << maxx(3.5, 20.4) << endl;
```

Иногда общее определение, предоставляемое шаблоном, для некоторых типов вообще не работает, а для некоторых работает неэффективно. В этом случае необходимо предоставить специализированное определение для конкретизации шаблона (*специализации*) или использовать перегруженную функцию с нужным списком параметров.

```
template <>    //специализация шаблона
const char* maxx(const char* a, const char* b) {
  if (strcmp(a, b)>0)
    return a;
  else
    return b;
}

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

//перегруженная функция
date maxx(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;
}
```

Порядок определения, какой экземпляр функции соответствует данному вызову, следующий [16].

- ✓ 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 учётом автоматического преобразования типов.

4.4. Приведение типов данных

Приведение типов данных в стиле языка C доступно и в языке C++ и до сих пор применяется в силу лаконичности записи. Приведение в стиле C выполняется с использованием одного или нескольких преобразований. Оно может быть использовано для преобразования любого типа в любой другой тип, при этом не учитывается, что это преобразование может быть небезопасным. Например, преобразование целого числа в указатель на тип

int. С точки здравого смысла, такое преобразование некорректно, однако компилятор выполнит это приведение.

Чтобы выполнить явное приведение типа в стиле С, 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 (таблица 5), способных полностью заменить операции приведения в стиле С.

Таблица 5

Операторы 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 надёжного 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.

Новый синтаксис приведения типа сразу бросается в глаза и отличается от других элементов программы.

```
xxx_cast< type_to >( expression_from )
```

Например:

```
int b = 200;
unsigned long a = static_cast< unsigned long >( b );
string s = static_cast< string >("Hello!");
ofstream fout("nameF",ios::app |ios::binary);
fout.write(reinterpret_cast <char*>(&b),sizeof b);
```

Если неправильно используются операторы приведения в стиле C++, то компилятор сообщит об ошибке. Приведение в стиле C не обладает такой возможностью. Если в программе встречается приведение типа, то следует воспринимать его как предупреждение, что происходит что-то необычное. При этом операторы в стиле C++ несут больше информации.

4.5. Структуры

Структуры относятся к составным типам данных. Структура хранит набор объектов, которые могут быть разных типов. Поэтому для структур операция индексации не определена. Каждый объект, входящий в структуру, имеет собственное имя и является её членом. Это имя используется для доступа с помощью операции точка (.).

Структуры можно рассматривать [16] как пользовательские типы данных. Поэтому такой тип нужно определить перед тем, как его использовать. При определении типа структуры используется ключевое слово `struct`, за которым следует имя определяемого типа. Далее в фигурных скобках перечисляются через точку с запятой определения членов. Определение структуры должно завершаться точкой с запятой.

Например, определим структуру для хранения даты:

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

После определения имя типа можно использовать аналогично встроенным типам данных для определения переменных, параметров и возвращаемых значений функций. Например:

```
date birthday;
date *holiday;
date vacation[28];
date max(const date &a, const date &b);
```

Переменную типа структуры можно инициализировать при объявлении. Значения полей заключаются в фигурные скобки и перечисляются через запятую, аналогично инициализации массивов. Например:

```
date a={9,5,2008,"may"}, b={1,1,2008,"январь"};
```

Доступ к членам структуры выполняется посредством операции точка, например:

```
cout<<a.monthName<<endl;
```

Пример 55. Для описанной структуры `date` реализовать функции: определения большей из двух дат, вывода даты в формате DD.MM.YYYY, вывода в формате DD–Month–YYYY.

```
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;
struct date {
  int day, month, year;
  string monthName;
};

void print1(const date& x){
  cout.fill('0');
  cout<<setw(2)<<x.day<<'.'<<setw(2)<<x.month<<'.'<<x.year;
}

void print2(const date& x){
  cout<<x.day <<'-'<<x.monthName<<'-'<<x.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(){
  setlocale(0, "Russian");
  date a = { 31, 5, 2008, "may" },
       b = { 1, 1, 2008, "январь" };
  print1(b);
  cout << endl;
  print1(a);
  cout << endl;
  date c = max(a,b);
  print2(c);
  return 0;
}
```

Рассмотрим заголовок функции для определения наибольшей даты:

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

Так как структуры относятся к составным типам данных и могут иметь большой размер, то эффективная передача их в функции в качестве параметров выполняется по ссылке. Для защиты параметров от изменений добавляется квалификатор `const`.

☺ Рекомендуется для структур и других составных типов данных (строк типа `string`, классов) использовать передачу параметров по ссылке. В случае необходимости запрета на изменение параметров внутри тела функции следует использовать квалификатор `const`.

Часто возникает необходимость в объявлении указателя на структуру и получении доступа к членам структуры через указатель. Например:

```
date *holiday=&b;
date *workday=new date;
```

В этом случае можно получить доступ к названию месяца посредством `(*holiday).monthName`. Скобки здесь необходимы, так как операция точка является постфиксной и имеет более высокий приоритет, чем любая префиксная операция, в том числе операция разыменования `*`. Для упрощения записи используется постфиксная операция стрелка (`->`), которая позволяет обращаться к членам структуры через указатель. Таким образом, обозначение `holiday->monthName` равносильно приведённому выше.

☺ Для упрощения записи рекомендуется при доступе к членам структуры через указатель использовать операцию стрелка.

Структуры в языке C++ получили существенное развитие по сравнению с их аналогами в языке C, приобретя возможность включать функции в качестве своих членов.

Пример 56. Для описанной структуры `date` реализовать нахождение следующей даты как функцию — член структуры.

```
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;
struct date {
  int day;
  int month;
  int year;
  string monthName;
  int daysInMonth();
  void nextDate();
};
void print1(const date& x){
  cout.fill('0');
  cout<<setw(2)<<x.day<<'.'<<setw(2)<<x.month<<'.'<<x.year;
}
void print2(const date& x){
  cout<<x.day <<'-'<<x.monthName<<'-'<<x.year;
}
int date::daysInMonth(){
  int days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
  //невисокосный год
```

```
  if (month == 2){
    if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
      return 29;
    return 28;
  }
  return days[month - 1];
}
void date::nextDate() {
  day++;
  if (day > daysInMonth()){
    day = 1;
    month++;
    if (month > 12){
      month = 1;
      year++;
    }
  }
}
int main() {
  date a = { 31, 5, 2008, "may"};
  print1(a);
  cout << endl;
  a.nextDate();
  print1(a);
  return 0;
}
```

Рассмотрим подробнее вызов функции-члена:

```
a.nextDate();
```

Имя функции уточняется через точку именем переменной типа `date`, аналогично обращению к полям структур. Указатель на переменную, для которой вызывается функция-член, передаётся в неё в качестве неявного параметра. Поэтому в данной функции отсутствует список параметров. Доступ к полям структуры, вызвавшей функцию, в теле функции-члена осуществляется напрямую без уточнения именем переменной типа `date`. Аналогичное правило действует и для доступа к другим функциям-членам этой же структуры, например, для вызова вспомогательной функции `daysInMonth()`.

```
void date::nextDate() {
  day++;
  if (day > daysInMonth()){
    day = 1;  month++;
```

```
    if (month > 12){
      month = 1; year++;
    }
  }
}
```

В данном примере объявления и определения функций-членов разделены. Объявления расположены внутри структуры, а определения вынесены за её пределы. При этом определение функций-членов за пределами структуры требует уточнения имени функции именем структуры с помощью операции разрешения области видимости.

Использование структур предполагает введение новых пользовательских типов. Как встроенный, так и пользовательский тип определяется набором допустимых значений и операций. В языке С реализация операций для пользовательского типа возможна только в виде внешних функций. Возможность включения функций в качестве членов структуры, появившаяся в языке С++, усилила связь между типом данных и определённым над ним набором операций [16].

ГЛАВА 5. ФАЙЛЫ И СПИСКИ

5.1. 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, 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 также описаны 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` соотнесён 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, определённых 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 с клавиатурой. Аналогично, `cout` является объектом класса `ostream` и соответствует стандартному потоку вывода, который по умолчанию связан с монитором. Ещё двумя стандартно определёнными 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 не буферизованным, поэтому вся направляемая в этот поток информация сохранится в нем, даже если программа завершится аварийно.

Заголовочный файл fstream определяет классы `ifstream` и `ofstream`, позволяющие работать с файлами как с потоками.

И средства языка С, и средства языка С++ предоставляют возможность работать с файлами в текстовом и двоичном режимах.

С точки зрения операционной системы любой файл — это набор двоичных данных. Определением как интерпретировать эти данные занимается прикладная программа. Принято различать два режима интерпретации — текстовый и двоичный. Одно из отличий заключается в том, что в текстовом режиме некоторые символы (их двоичные коды) обрабатываются особым образом – это символы конца строки, перехода на новую строку, пробелы, символ табуляции. В двоичном режиме все символы равнозначны. Кроме того, ввод/вывод данных в текстовом режиме может сопровождаться операцией преобразования из/в символьное представление. В двоичном режиме преобразований не происходит.

Для простоты файлы, обрабатываемые в текстовом режиме, будем называть текстовыми, а в двоичном — бинарными.

5.2. Работа с текстовыми файлами в стиле С++

Пример 57. 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 создаётся объект класса `ofstream`. Используемый конструктор с параметром (именем файла) создаёт 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 (являющиеся служебными в текстовом режиме) 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 58. Написать программу, подсчитывающую общее количество символов в нескольких текстовых файлах. Список имён файлов, для которых нужно выполнить подсчёт, задаётся в списке аргументов командной строки при запуске программы.

```
#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 приём ввода имён обрабатываемых файлов через аргументы командной строки. Для рассматриваемой программы, например, после её 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 параметры в заголовке функции `main()`:

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

Параметр `argc` передаёт 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 передаётся 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]);
```

Метод 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 защищённый 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, но метод `close()` может не сбрасывать биты в маске состояния потока.

Как и в предыдущем примере, для подсчёта 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`.

При последовательной обработке файлов использовался только один объект типа `ifstream`. В данном примере это удобно, потому что все файлы обрабатываются в цикле.

На каждый объект файлового типа выделяются ресурсы, которые будут освобождены только тогда, когда завершится время жизни этого объекта. Поэтому данный приём экономит ресурсы.

☺ Всегда, когда алгоритм позволяет обрабатывать файлы последовательно, рекомендуется использовать один объект файлового типа, по очереди связывая его с файлами.

Пример 59. Написать программу для обработки текстовых файлов: вывод содержимого на экран, создание копии файла, выдача содержимого

файла по словам. Выбор режима обработки организовать посредством простого 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;
}
```

Приведённые 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 вперёд, чтобы определить конец слова или конец строки. Для пропуска пробелов между словами внутри строки используется функция `skipBlank()`.

Пример 60. Написать программу, позволяющую вводить строки и дописывать их в текстовый файл. Добавляемые строки переформатировать так, чтобы их длины не превышали 80 символов.

```
#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) {
  const int limit = 80;
  const int size = limit + 1;
  ofstream fout(filename, 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[size];
  cin.getline(line, size);
  while (line[0] != '\0') {
    fout << line << "\n";
    cin.clear();
    cin.getline(line, size);
  }
  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 задаётся 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 `ios::app` означает, что файл должен быть открыт для записи с сохранением имеющейся в нем информации и добавлением новых данных в конец файла. При этом если файл не существует, то он создаётся.

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

Признаком окончания ввода является пустая строка:

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

При использовании функции `cin.getline(line,size)` ввод строки в переменную `line` завершается или при вводе символа новой строки, или после ввода количества символов, ограниченного константой `limit=size-1`. Если прочитан `size-1` символ и при этом не достигнут символ конца строки, то устанавливается признак ошибки `failbit`. В этом состоянии дальнейшее использование потока `cin` вызывает исключение. Чтобы продолжить ввод, необходимо сбросить `failbit`:

```
cin.clear();
```

Пример 61. Написать программу, позволяющую создать «сжатую» копию текстового файла, удалив из него все пустые строки. При просмотре текстового файла в текстовом редакторе строка, содержащая только пробельные символы, воспринимается как пустая. Поэтому пустыми

строками считать строки, содержащие символ конца строки и, возможно, пробелы.

```
#include <iostream>
#include <fstream>
using namespace std;
int seekEoln(ifstream &in, char &c){
  c = in.get();
  int k = 0;
  while (!in.eof() && c == ' '){
    k++;
    c = in.get();
  }
  if (in.eof() || c == '\n')
    return -1;
  return k;
}
int main() {
  char filename[30] = "text.txt";
  char filename2[30] = "text2.txt";
  ifstream inFile(filename);
  if (!inFile.is_open()) {
    cerr << "Can't open " << filename << "\n";
    return -1;
  }
  ofstream outFile(filename2);
  if (!outFile.is_open()) {
    cerr << "Can't open " << filename2 << "\n";
    return -1;
  }
  int k;
  char c;
  while (!inFile.eof()){
    k = seekEoln(inFile, c);
    if (k > -1){
      for (int i = 0; i < k; ++i)
        outFile << ' ';
      while (!inFile.eof() && c != '\n'){
        outFile << c;
        c = inFile.get();
      }
      if (!inFile.eof())
        outFile << '\n';
    }
  }
  outFile.close();
  inFile.close();
  return 0;
}
```

Функция `int seekEoln(ifstream &in, char &c)` пропускает начальные пробелы в текущей строке файла. Функция возвращает -1, если строка пустая, или количество пропущенных пробелов, при этом параметр `c` содержит первый непробельный символ.

```
k = seekEoln(inFile, c);
```

Полученные значения `k` и `c` используются для формирования непустых строк в выходном файле.

```
if (k > -1){
  for (int i = 0; i < k; ++i)
    outFile << ' ';
  while (!inFile.eof() && c != '\n'){
    outFile << c;
    c = inFile.get();
  }
  if (!inFile.eof())
    outFile << '\n';
}
```

☠ Следует помнить, что каждая операция чтения из файла может сгенерировать исключительную ситуацию конца файла. Поэтому перед каждой следующей операцией чтения необходимо проверять состояние потока, например с помощью функции `eof()`.

5.3. Работа с бинарными файлами в стиле C++

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

Пример 62. Написать программу для организации работы с таблицей, хранящей данные о студентах (имя, год рождения, средний балл по итогам последней сессии). Для этого реализовать следующие функции: создание файла, вывод его содержимого на экран, выдача записи по номеру.

```
#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;
  ifstream fin(nameF, ios::in | ios::binary);
  streampos place = n * sizeof p;
  fin.seekg(place);
  if (fin.fail())
    exit(1);
```

```
  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();
}

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 (или открытии) потокового объекта 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 | (поразрядное 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 определённое число байтов из памяти в файл. Количество байтов, которое должно быть скопировано, задаётся 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 определённым номером. Для этого используются методы: `seekg()` — с объектами классов `ifstream` и `fstream`, и `seekp()` — с объектами классов `ofstream` и `fstream`.

Для обоих методов существуют два варианта перегрузок с одним параметром и с двумя. В первой версии функции позиция задаётся абсолютным значением, во второй — через смещение от одной из позиций (начало, текущая, конец). В рассматриваемом примере использована первая версия:

```
fin.seekg(place);
```

Пример 63. Дан файл вещественных чисел. Обнулить элементы файла, значения которых меньше среднего арифметического всех чисел в файле, хранящихся в файле.

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

int t=sizeof(double);

void toFile(char* nameF) {
  ofstream fout(nameF, ios::out | ios::binary);
  if (!fout.is_open()) {
    cerr << "Can't open " << nameF << "\n";
    exit(1);
  }
  double val;
```

```
  cout << "Enter value ";
  while (cin >> val){
    fout.write(reinterpret_cast <char*>(&val), t);
  }
  fout.close();
}

void echoFile(char* nameF) {
  double val;
  ifstream fin(nameF, ios::in | ios::binary);
  if (fin.is_open()) {
    while (fin.read(reinterpret_cast <char*>(&val),t))
      cout << val << " ";
    cout << endl;
  }
  fin.close();
}

double average(fstream &f) {
  double p,s=0;
  int k=0;
  streampos posg = f.tellg();
  f.seekg(0, ios_base::beg);
  f.read(reinterpret_cast <char*>(&p), t);
  while (!f.eof()){
    s += p;
    k++;
    f.read(reinterpret_cast <char*>(&p), t);
  }
  f.clear();
  f.seekg(posg, ios_base::beg);
  if (k) return s / k;
  return 0;
}

void smooth(fstream &f, double avg){
  double p;
  streampos posg = f.tellg();
  streampos posp = f.tellp();
  f.seekg(0, ios_base::beg);
  streampos pos=f.tellg();
  f.read(reinterpret_cast <char*>(&p), t);
  while (!f.eof()){
    if (p < avg){
      f.seekp(pos, ios_base::beg);
      p = 0.0;
      f.write(reinterpret_cast <char*>(&p), t);
    }
    pos = f.tellg();
    f.seekg(pos, ios_base::beg);
```

```
    f.read(reinterpret_cast <char*>(&p), t);
  }
  f.clear();
  f.seekg(posg, ios_base::beg);
  f.seekp(posp, ios_base::beg);
}

int main() {
  char nameF[30] = "inf.dat";
  toFile("inf.dat");
  //если файл inf.dat существует,
  //вызов функции toFile не нужен
  echoFile("inf.dat");
  fstream finout(nameF, ios::in | ios::out | ios::binary);
  if (!finout.is_open()) {
    cerr << "Can't open " << nameF << "\n";
    exit(1);
  }
  double d= average(finout);
  smooth(finout, d);
  finout.close();
  echoFile("inf.dat");
  return 0;
}
```

Функция `void toFile(char* nameF)` создаёт бинарный файл вещественных чисел. Ввод данных с клавиатуры организован с помощью цикла `while`.

```
while (cin >> val){…}
```

Условие, при котором цикл продолжается, — корректный ввод вещественного числа. При вводе данных в неподходящем формате (ввод символа вместо ожидаемого числа) значением выражения `cin>>val` является `false`.

Обратите внимание на то, что функции `average` и `smooth` в качестве параметра получают потоковую переменную, передаваемую по ссылке.

```
double average(fstream &f)
void smooth(fstream &f, double avg)
```

 Потоковые переменные всегда передаются в функции по ссылке.

В этом случае операции открытия и закрытия файловых потоков выполняются вне этих функций. Поскольку неизвестно, где расположен указатель файлового потока при вызове функции, необходимо установить его в нужную позицию. В данном примере используется определение позиции через смещение относительно начала файла:

```
f.seekg(0, ios_base::beg);
```

Чтобы после выхода из функции указатель файлового потока находился в той же позиции, что и до вызова функции, применяется следующий приём: перед первым изменением запоминаем позицию указателя, перед завершением функции возвращаем указатель в запомненную позицию.

```
streampos posg = f.tellg();
…
f.seekg(posg, ios_base::beg);
```

Позиция указателя является объектом класса `streampos`.

☺ В функциях, в которых файловый поток является параметром, рекомендуется в начале функции сохранять позицию указателя, а в конце — её восстанавливать.

Для запоминания позиции потокового файл типа `fstream` можно использовать одну из функций `tellg/tellp`, для установки позиции — `seekg/seekp`.

```
while (!f.eof()){
  if (p < avg){
    f.seekp(pos, ios_base::beg);
     p = 0.0;
    f.write(reinterpret_cast <char*>(&p), t);
  }
  pos = f.tellg();
  f.seekg(pos, ios_base::beg);
  f.read(reinterpret_cast <char*>(&p), t);
}
```

Для операций чтения и записи у классов, работающих с потоками, имеются разные указатели. Во избежание ошибок перед каждой такой операцией в примере явно устанавливается позиция указателя.

☠ Если предполагается использовать потоковый файл типа `fstream` и для выполнения операции чтения (`read`) и для выполнения операции записи (`write`), то перед каждой операцией чтения/записи следует явно устанавливать указатель файлового потока в нужную позицию.

Поскольку организация цикла использует проверку состояния файлового потока `eof`, то после завершения цикла необходимо очистить битовую маску состояния потока:

```
f.clear();
```

5.4. Работа с текстовыми файлами в стиле языка С

Пример 64. 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 как функции, так и типа структуры `FILE` необходимо подключить файл cstdio.

Параметр для установки режима открытия является строкой. Возможные режимы открытия приведены в таблице 6.

Таблица 6

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 создаётся.
"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 создаётся.

По умолчанию файл открывается в текстовом режиме. В случае открытия файла в бинарном режиме в строку определения режима необходимо добавить букву «b». При наличии любой ошибки чтения-записи `fopen` возвращает константу `NULL`. Открытие файла удобно совмещать с проверкой корректности данной операции:

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

Функция `fclose(fp)` закрывает файл. Она разрывает связь между файловым указателем и внешним именем и освобождает буфер, в котором могли остаться предназначенные для вывода данные.

Функция `void filecopy(FILE *ifp,FILE *ofp)` предназначена для посимвольного копирования содержимого одного файла в другой. В ней используются две стандартные функции для посимвольного ввода/вывода в текстовом режиме.

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

Функция `getc` возвращает следующую литеру из файла (потока), определяемого указателем `*ifp`. В случае исчерпания файла или ошибки она возвращает значение, определяемое константой `EOF`:

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

Функция `putc` пишет символ в файл и возвращает успешно записанный символ или `EOF` в случае ошибки:

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

Если в командной строке присутствуют аргументы, то они рассматриваются как имена последовательно обрабатываемых файлов. Если аргументов нет, то обработке подвергается 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);
  }
```

Программа сигнализирует о возможных 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 код возврата в инструкции:

```
return выражение;
```

Функция `ferror(stdout)` выдаёт ненулевое значение, если при записи в файл `stdout` возникает ошибка. Хотя при выводе данных без преобразования редко происходят 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. Единственное отличие состоит в том, что её первым аргументом является указатель, ссылающийся на файл, для которого осуществляется вывод. Формат вывода указывается вторым аргументом, далее следует список выводимых данных:

```
int fprintf(FILE *fp, char *format, ...)
```

Пример 65. Написать программу для демонстрации использования функции `fprintf` для различных типов данных. Вывод данных необходимо осуществить в файл с именем 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';
  stream = fopen("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`, позволяющей вызывать команды операционной системы. Команда передаётся в виде строки. В данном примере строка содержит команду `type` (для Windows) и имя выводимого файла. Для использования функции `system` необходимо подключить файл cprocess.

Для форматированного вывода в языке С используется несколько функций. Функция `printf` выводит свои аргументы в форматированном виде в стандартный поток вывода, `sprintf` — в строку, `fprintf` — в текстовый файл. Для каждой из них формат описывается в параметре для строки формата. Все они возвращают количество выведенных символов.

Строка формата содержит обычные символы, которые копируются в поток вывода, и спецификации формата. Каждая спецификация начинается с символа %, за которым следует один или несколько символов, определяющих формат выводимых данных.

Более подробную информацию о форматированном выводе можно найти в справочниках и книгах по языку С. Например, Б. Керниган и Д. Ритчи «Язык программирования Си» [7]. Некоторые часто используемые спецификации форматов приведены в таблице 7.

Таблица 7

Основные спецификации формата

Символ	Тип аргумента и способ вывода
d, i	десятичное целое
c	отдельный символ
s	выводятся символы из строки char *, пока не встретится '\0', или в количестве, заданном параметром точности
f	формат с плавающей точкой

Пример 66. Удалить из текстового файла пустые строки.

```
#include <cstdio>

int empty(char *s){
while (*s && *s==' ')
    s++;
  if (!*s)
    return 1;
  if (*s!='\n')
    return 0;
  return 1;
}

int main() {
  FILE *file, *temp;
  char buf[255];
  if ((file = fopen("file.txt", "r")) == NULL){
    fprintf(stderr,"Can not open file %s\n", "file.txt");
    return(1);
  }
  else {
    temp = fopen("temp.txt", "w");
    while (fgets(buf, 255, file))
    {
      if (!empty(buf))
        fputs(buf, temp);
    }
    fclose(file);
    fclose(temp);
    remove("file.txt");
    rename("temp.txt", "file.txt");
  }
  return 0;
}
```

Функция `fgets` считывает строку текста из файла, задаваемого переменной `file` и записывает её в строку в стиле C, на которую указывает переменная `buf`. Чтение заканчивается, если прочитано 254 символа или введён символ новой строки. Символ новой строки, если он был встречен, тоже записывается в `buf`. В случае успеха возвращается прочитанная строка `buf`, в случае достижения конца файла — нулевой указатель.

Значение 254 в этой задаче определяет максимальную длину строки в памяти. Для корректной работы длина строк в файле также не должна превышать 254.

Функция `remove` удаляет файл file.txt. Функция `rename` переименовывает файл temp.txt в file.txt. Обе функции вызывают команды операционной системы. Они применимы только к закрытым файлам.

```
remove("file.txt");
rename("temp.txt", "file.txt");
```

Для классов `fstream`, `istream` и `ostream` методов, аналогичных функциям `remove` и `rename`, не существует. Поэтому в задачах с использованием потоковых классов приём с удалением и переименованием файлов обычно не используется.

☺ При обработке файлов не рекомендуется смешивать использование потоковых классов и функций библиотеки cstdio.

5.5. Работа с бинарными файлами в стиле языка С

Для операций чтения и записи в бинарных файлах используются функции `fread` и `fwrite`:

```
size_t fread(void* ptr,size_t size, size_t count, FILE* stream)
size_t fwrite(const void * ptr,size_t size, size_t count,
              FILE* stream)
```

Функция `fread` читает не более `count` элементов длиной `size` байт и записывает их в область памяти, на которую указывает `ptr`. Функция возвращает количество успешно считанных байтов. Функция `fwrite` имеет аналогичные параметры.

Пример 67. Дан бинарный файл, содержащий целые числа. Проверить, что эти числа расположены симметрично относительно середины файла.

```
#include <cstdio>
#include <iostream>
```

```
using namespace std;
int t = sizeof(int);
void createFile(char *name){
  FILE *file;
  file = fopen(name, "wb");
  int val;
  cout << "Enter value ";
  int t = sizeof val;
  while (cin >> val){
    fwrite(&val, t,1,file);
  }
  fclose(file);
}
bool simm(FILE *file){
  if (feof(file))
    return true;
  long int beg, end;
  beg = ftell(file);
  fseek(file, -t, SEEK_END);
  end = ftell(file);
  int x, y;
  while (beg < end) {
    fseek(file, beg, SEEK_SET);
    fread(&x, t, 1, file);
    fseek(file, end, SEEK_SET);
    fread(&y, t, 1, file);
    if (x != y)
      return false;
    beg += t;
    end -= t;
  }
  return true;
}

int main(int argc,char * argv[]) {
  FILE *file;
  createFile("file.dat");
  if ((file = fopen("file.dat", "rb")) == NULL){
    fprintf(stderr, "I can not open file %s\n", "file.dat");
    return(1);
  }
  else {
    if (simm(file))
      cout << "OK!";
    else
      cout << "NOT OK";
  }
  fclose(file);
  return 0;
}
```

При открытии файла в бинарном режиме, режим нужно указывать явно:

```
file = fopen(name, "wb");
file = fopen("file.dat", "rb");
```

Если требуется произвольный доступ к файлу, необходимо использовать функции `ftell` и `fseek`.

Функция `ftell(FILE*)` возвращает текущую позицию в файле:

```
beg = ftell(file);
```

Функция `fseek(FILE*, long int, int)` перемещает указатель на заданную позицию в файле:

```
fseek(file, -t, SEEK_END);
fseek(file, beg, SEEK_SET);
```

Для третьего параметра есть макросы `SEEK_SET`, `SEEK_CUR`, `SEEK_END`. В случае ошибки функция `fseek` возвращает ненулевое значение.

Указатели на позицию в файле — целые числа (`long int`). Поэтому в функции используется сравнение значений указателей (`beg<end`) и их изменение на величину размера записи в файле. При этом указатель `beg` увеличивается, а `end` уменьшается. Выход из цикла произойдёт, когда `beg` станет больше либо равен `end`.

```
while (beg < end) {
  fseek(file, beg, SEEK_SET);
  fread(&x, t, 1, file);
  fseek(file, end, SEEK_SET);
  fread(&y, t, 1, file);

  if (x != y)
    return false;
  beg += t;
  end -= t;
}
```

5.6. Динамические структуры данных. Односвязные списки

Динамические структуры данных — это такие структуры, которые в ходе выполнения программы могут менять свой размер. Удобным средством реализации динамических структур являются списочные структуры (списки).

Списки — это программно-реализуемые структуры данных, элементы которых хранят информацию и связи с другими элементами. В качестве связи используется указатель на элемент. По количеству связей и направленности выделяют различные типы списков (односвязные, двусвязные и многосвязные; линейные, нелинейные, кольцевые и т. д.).

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

Поскольку все элементы списка размещаются в динамической памяти, нужно иметь указатель хотя бы на один элемент списка. Такой указатель (указатели) располагается в автоматической области памяти.

Для линейного односвязного списка указатель должен ссылаться на первый элемент, который называется головой списка. Иногда при реализации алгоритмов удобно иметь указатель на последний элемент списка, который можно назвать хвостом списка.

 В общем случае «хвостом» называется весь список без головы. Такая терминология активно используется в функциональных языках программирования.

Пример 63. Создать линейный односвязный список целых чисел, добавлением элементов в конец списка. Операции создания списка, удаления и вывода на экран оформить в виде функций.

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

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

list* sp_create_to_tail();
void print_sp(list* L);
void erase(list*&L);

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

list* sp_create_to_tail() {
  list* head, *p, *tail;
  int n;
  cout << "size list->";
  cin >> n; head = tail = nullptr;
  if (n > 0) {
    tail=head = new list;
    cout << "list item ->";
    cin >> head->inf;
    head->next = nullptr;
  }
  for (int i = 1; i<n; ++i) {
    p = new list;
    cout << "list item ->";
    cin >> p->inf;
    p->next = nullptr;
    tail->next = p;
    tail = p;
  }
  return head;
}

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

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

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

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

Функция `sp_create_to_tail` создаёт список и возвращает указатель на его начало. Поэтому функция не имеет параметров:

```
list* sp_create_to_tail();
```

Параметром функции `print_sp` является указатель на структуру, передаваемый по значению:

```
void print_sp(list* L);
```

Параметром функции `erase` также является указатель на структуру. Но этот параметр передаётся по ссылке, так как его значение в функции меняется:

```
void erase(list*&L);
```

В функцию удаления списка передаётся 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;
```

Тогда объявления соответствующих функций будут выглядеть так:

```
sp_ptr sp_create();
void print_sp(sp_ptr L);
void erase(sp_ptr &L);
```

Пример 69. 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;
};

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

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

sp_ptr sp_create() {
  sp_ptr L,p;
  int n;
  cout <<"size list->" ;
  cin >> n; L= nullptr;
  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(sp_ptr L) {
  sp_ptr p;
```

```
  p=L;
  while (p!= nullptr) {
    cout << p->inf<< " ";
    p=p->next;
  }
  cout <<"\n";
}

void compress(sp_ptr L) {
  sp_ptr p, q, t, d;
  p=L;
  while (p!= nullptr) {
  q=p->next;
    while (q!= nullptr && 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(sp_ptr&L) {
  sp_ptr t;
  t=L;
  while(t!= nullptr){
    L=t->next;
    delete t;
    t=L;
  }
}
```

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.

В функции compress указатели p и q определяют диапазон (открытый справа полуинтервал) подряд идущих одинаковых элементов. Если он содержит более одного элемента, то выполняется удаление всех элементов, кроме того, на который указывает p.

```
t=p->next;
if (t!=q) {
  p->next=q;
  while (t!=q) {
    d=t;
    t=t->next;
    delete d;
  }
}
```

Значение указателя p сохраняется до следующей итерации внешнего цикла. При такой организации циклов p никогда не перемещается на следующий элемент списка, равный предыдущему. Каждый указатель, который будет разыменован в теле цикла, необходимо проверять на неравенство nullptr.

```
while (p!= nullptr) {
q=p->next;
  while (q!= nullptr && p->inf==q->inf)
  …
}
```

Пример 70. Реализовать набор рекурсивных функций для обработки элементов линейного односвязного списка: печать списка в обратном порядке; определение количества элементов, удовлетворяющих предикату; проверка выполнения условия, задаваемого двуместным предикатом, для всех соседних пар элементов списка.

```
void reversePrintSp(spPtr L) {
  if (L) {
    reversePrintSp(L->next);
    cout <<L->inf << " ";
  }
  else
    cout << "Reverse print" << endl;
}
```

```
bool isPositive(int x) {
  return x > 0;
}
bool isNotOdd(int x) {
  return x % 2 == 0;
}
bool isGreate(int x, int y){
  return x > y;
}
bool pairProperty(spPtr L, bool(*f)(int, int)){
  if (L && L->next){
    return f(L->inf, L->next->inf) && pairProperty(L->next,f);
  }
  else return true;
}

int countPred(spPtr L, bool(*f)(int)){
  if (L){
    if (f(L->inf))
      return 1 + countPred(L->next,f);
    else
      return countPred(L->next,f);
  }
  else
    return 0;
}
```

Вернёмся к описанию структуры для элемента списка

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

Обратим внимание, что определение является рекурсивным. Поэтому для обработки списочных структур удобно использовать рекурсивные алгоритмы.

При поэлементной обработке списка условие выхода из рекурсии определяется проверкой указателя на пустоту:

```
if (L){
  //рекурсивная часть
  …
}
else {
  //нерекурсивная часть
  …
}
```

В случае попарной обработки элементов списка для рекурсивного продолжения необходимо выполнение двух условий (при этом важен порядок проверки):

```
if (L && L->next){
  //рекурсивная часть
  …
  }
 else {
  //нерекурсивная часть
  …
}
```

Примеры вызовов:

```
reversePrintSp(F);
cout<<"count of Positive = "<<countPred(F,isPositive)<<endl;
cout<<"count of NotOdd = "<<countPred(F,isNotOdd)<<endl;
cout<<"desc ordered - "<<pairProperty(F,isGreate)<<endl;
```

5.7. Двусвязные списки

Линейные односвязные списки не поддерживают с должной эффективностью некоторые важные операции, например переход на предыдущий элемент списка.

Самым простым решением является добавление ещё одного указателя (на предыдущий элемент). В результате чего перемещение по списку выполняется одинаково эффективно в обоих направлениях. Такой список называется линейным двусвязным. Чтобы все было симметрично, рекомендуется хранить информацию (указатель) не только для первого, но и для последнего элемента.

Однако добавление дополнительного указателя требует внесения изменений в реализацию всех операций.

Пример 71. Реализовать набор функций для обработки элементов линейного двусвязного списка: добавление элемента в список между двумя указателями; печать списка в прямом порядке; печать списка в обратном порядке; удаление списка.

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

struct list {
  int inf;
  list *prev, *next;
};

typedef list* pList;

void insert(pList &F, pList &L, int a, pList L1= nullptr,
            pList L2= nullptr);
void print(pList F);
void reversePrint(pList L);
void delNode(pList &F, pList &L, pList p);
void erase(pList &F, pList &L);

int main() {
  pList F = nullptr, L = nullptr;
  insert(F, L, 5);
  insert(F, L, 3, nullptr, F);
  insert(F, L, 8, L);
  insert(F, L, 4, F, F->next);
  insert(F, L, 7, L->prev, L);
  print(F);
  reversePrint(L);

  delNode(F, L, F);
  print(F);
  reversePrint(L);

  delNode(F, L, L);
  print(F);
  reversePrint(L);
  delNode(F, L, F->next);
  print(F);
  reversePrint(L);

  erase(F, L);
  return 0;
}

// поместить элемент между L1 и L2
void insert(pList &F, pList &L, int a, pList L1, pList L2 ) {
  // L1 и L2 должны указывать на расположенные подряд
  // элементы одного списка, причём L1 < L2
  // или L1 должно быть nullptr, если L2 – голова
  // или L2 должно быть nullptr, если L1 – хвост
  assert(L1 == nullptr || L2 == nullptr || L1->next == L2);
```

```
  pList p = new list;
  p->inf = a;
  p->prev = L1;
  p->next = L2;
  if (L1)
    L1->next = p;
  else
    F = p;
  if (L2)
    L2->prev = p;
  else
    L = p;
  if (!L1 && !L2)
    F = L = p;
}

void print(pList F) {
  pList p;
  p = F;
  while (p != nullptr) {
    cout << p->inf << " ";
    p = p->next;
  }
  cout << "\n";
}

void reversePrint(pList L){
  pList p;
  p = L;
  while (p != nullptr) {
    cout << p->inf << " ";
    p = p->prev;
  }
  cout << "\n";
}

void delNode(pList &F, pList &L, pList p) {
  assert(F!= nullptr && p != nullptr);
  if (p == F) {
    F = F->next;
    if (F == nullptr)
      L = nullptr;
    else
      F->prev = nullptr;
  }
  else if (p == L){
    L = L->prev;
    L->next = nullptr;
  }
```

```
  else{
    p->next->prev = p->prev;
    p->prev->next = p->next;
  }
  delete p;
}

void erase(pList &F, pList &L) {
  pList t;
  t = F;
  while (t != nullptr){
    F = t->next;
    delete t;
    t = F;
  }
  F = L = nullptr;
}
```

Описание узла списка теперь выглядит следующим образом:

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

При создании элемента двусвязного списка добавляется инициализация ещё одной ссылки.

```
pList p = new list;
p->inf = a;
p->prev = L1;
p->next = L2;
```

Во всех функциях, где добавляются или удаляются элементы, список передаётся двумя указателями (на первый и последний элементы), причём оба передаются по ссылке.

```
void insert(pList &F, pList &L, int a, pList L1= nullptr,
              pList L2= nullptr);
void delNode(pList &F, pList &L, pList p);
void erase(pList &F, pList &L);
```

Особенностью реализации функции вставки является её универсальность. Она позволяет вставлять элемент с заданным значением `a` в любое место списка. Позиция вставки определяется двумя параметрами `L1`, `L2`. Для корректной работы функции на значения параметров накладывается ряд условий. Для добавления в начало списка параметр `L2`

должен указывать на первый элемент, а L1 должно быть равен nullptr. Для добавления в конец списка параметр L1 должен быть указателем на последний элемент, а L2 должен быть равен nullptr. В остальных случаях параметры L1, L2 должны указывать на расположенные подряд элементы одного списка, причём L1<L2.

Ниже приведены примеры вызовов этой функции:

```
insert(F, L, 5);         //Добавление элемента в пустой список
insert(F, L, 3,nullptr,F);//Добавление элемента в начало
списка
insert(F, L, 8, L);      //Добавление элемента в конец списка
insert(F, L, 4, F, F->next);  //Вставка после первого элемента
insert(F, L, 7, L->prev, L);  //Вставка перед последним
```

Несмотря на то, что функция обрабатывает все случаи вставки элемента, её код очень лаконичен. Помимо команд создания нового элемента и обработки случая пустого списка, код представляет собой два полных условных оператора:

```
if (L1)
  L1->next = p;
else
  F = p;
if (L2)
  L2->prev = p;
else
  L = p;
```

В функции удаления в качестве параметра передаётся указатель на удаляемый элемент.

5.8. Бинарные деревья

Деревья можно определить двумя способами: рекурсивным и нерекурсивным. Рекурсивное определение позволяет компактно описывать алгоритмы для работы с деревьями. При рекурсивном описании дерево или пусто или состоит из корня и нуля или более непустых поддеревьев. Каждое поддерево является деревом.

Если у каждого узла количество поддеревьев не превышает двух, то дерево называется *бинарным*. Оно может быть реализовано в виде нелинейного двусвязного списка. В этом случае указатели ссылаются на поддеревья. Если порядок расположения поддеревьев важен, то их принято называть левым и правым, а само дерево упорядоченным.

Самый верхний узел дерева называется *корневым узлом*. У него отсутствуют предки. С корневого узла начинается выполнение большинства операций над деревом (хотя некоторые алгоритмы начинают выполнение с листов и выполняются, пока не достигнут корня). Все прочие узлы могут быть достигнуты путём перехода от корневого узла по ссылкам. *Лист* (*терминальный узел)* — узел, не имеющий дочерних узлов. Каждый узел, кроме корневого, имеет ровно одного предка.

Уровень узла — длина пути от корня дерева до этого узла. Корень дерева имеет нулевой уровень. *Высота* узла — это максимальная длина нисходящего пути от этого узла к самому нижнему листу. Высота корневого узла равна высоте всего дерева. *Глубина* дерева равна высоте корневого узла.

Дерево из `n` узлов может иметь различную структуру и, соответственно, различную глубину. Дерево из `n` узлов, имеющее минимальную глубину, называется *сбалансированным*. В этом случае для каждого узла глубины его правого и левого поддеревьев могут отличаться не более чем на единицу.

Идеально сбалансированным называется дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более чем на 1.

Однако идеальную сбалансированность довольно трудно поддерживать. В некоторых случаях при добавлении или удалении

элементов может потребоваться значительная перестройка дерева, не гарантирующая логарифмическую сложность.

В 1962 году советские математики Г.М. Адельсон-Вельский и Е.М. Ландис ввели менее строгое определение сбалансированности и доказали, что при таком определении можно написать программы добавления и/или удаления, имеющие логарифмическую сложность и сохраняющие дерево сбалансированным. Дерево считается сбалансированным по АВЛ (сокращения от их фамилий), если для каждой вершины выполняется требование: высота левого и правого поддеревьев различаются не более, чем на 1. Не всякое сбалансированное по АВЛ дерево идеально сбалансировано, но всякое идеально сбалансированное дерево сбалансировано по АВЛ.

Пример 72. Создать и распечатать сбалансированное дерево, содержащее n целых чисел.

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

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

Напомним, что определение узла дерева является рекурсивным:

```
struct elem{
  int inf;
  elem* lt,*rt;
};
typedef elem* t_ptr;
```

Поэтому для обработки древовидных структур удобно использовать рекурсивные алгоритмы и, соответственно, рекурсивные функции:

```
t_ptr create(int n);
void erase(t_ptr t);
void printLKR(t_ptr t);
void printTREE(t_ptr t,int n);
```

Функция `create` создания сбалансированного дерева из `n` узлов создаёт один узел и выполняет рекурсивные вызовы для создания левого и правого поддерева, содержащих соответственно `n/2` и `n-1-n/2` узлов. Рекурсия завершается в случае `n=0`. При создании дерева используются следующий порядок обработки: корень, левое поддерево, правое поддерево (КЛП).

Функция `erase` в качестве параметра принимает указатель на корень дерева. Важно отметить, что рекурсивные вызовы для удаления поддеревьев должны выполняться раньше, чем удаление самого корня. Поэтому используется порядок: левое поддерево, правое поддерево, корень (ЛПК).

Большинство рекурсивных алгоритмов обработки деревьев основываются на трех порядках обхода: префиксный (КЛП), инфиксный (ЛКП) и постфиксный (ЛПК). В функциях `create` и `erase` проиллюстрированы первый и третий порядки обхода.

Для иллюстрации инфиксного обхода (левое поддерево, корень, правое поддерево) использована функция `printLKR`. Входным параметром для неё является указатель на корень дерева. Такой обход чаще всего применяется для бинарных деревьев специального вида — деревьев поиска. В этом случае выводится отсортированная последовательность.

Однако функция `printLKR` не позволяет увидеть структуру дерева. Для этих целей реализована функция `printTREE` с двумя параметрами: первый — указатель на корень, второй — уровень корня. С помощью второго параметра определяется величина отступа при выводе текущего

узла. В этой функции используется порядок обхода — правое поддерево, корень, левое поддерево. Он также является инфиксным.

На рисунке 5.1 приведены результаты двух вариантов вывода дерева: при инфиксном обходе слева направо и в виде дерева, повёрнутого на 90 градусов против часовой стрелки.

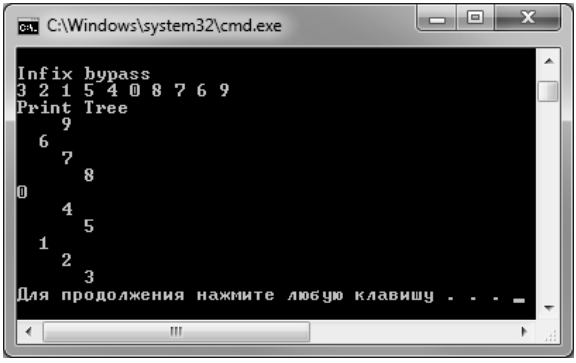

Рис. 5.1. Вывод сбалансированного дерева

Пример 73. Написать функции для вычисления суммы чётных элементов, максимального элемента и количества листьев в дереве целых чисел.

```
#include <iostream>
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 printTREE(t_ptr t, int n);
int sum(t_ptr t);
int max(t_ptr t);
int leafsCount(t_ptr t);

int main(){
  int n;
  t_ptr tree;
  cout << "Number of elements? (Cardinality) ";
```

```
  cin >> n;
  tree = create(n);
  cout << endl << "Print Tree" << endl;
  printTREE(tree, 0);
  cout << "sum="<< sum(tree)<< endl;
  cout << "max";
  if (tree)
    cout << " = " << max(tree) << endl;
  else
    cout << " for empty tree not defined" << endl;
  cout << "leafsCount=" << leafsCount(tree) << endl;
  erase(tree);
  return 0;
}

void printTREE(t_ptr t, int n) {
  if (t != nullptr) {
    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 nullptr;
}

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

```
int sum(t_ptr t){
  if (!t)
    return 0;
  if (t->inf % 2)
    return sum(t->lt) + sum(t->rt);
  return t->inf + sum(t->lt) + sum(t->rt);
}

int max(t_ptr t){
  int max1= t->inf;
  int max2;
  if (t->lt){
    max2 =max(t->lt);
    if (max2 > max1)
      max1 = max2;
  }
  if (t->rt){
    max2 = max(t->rt);
    if (max2 > max1)
      max1 = max2;
  }
  return max1;
}

int leafsCount(t_ptr t){
  if (!t)
    return 0;
  if (!t->lt && !t->rt)
    return 1;
  return leafsCount(t->lt) + leafsCount(t->rt);
}
```

При решении данной задачи используются рекурсивные функции:

```
int sum(t_ptr t);
int max(t_ptr t);
int leafsCount(t_ptr t);
```

В этих функциях выбирается порядок обхода, обеспечивающий наиболее прозрачную реализацию алгоритма.

Функция `max` не может быть вызвана для пустого дерева. Эта особенность учтена в основной программе следующим образом:

```
cout << "max";
if (tree)
  cout << " = " << max(tree) << endl;
else
  cout << " for empty tree not defined" << endl;
```

Пример 74. Написать функции для работы с деревом поиска: добавление элемента, поиск элемента, удаление элемента.

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

struct elem{
  int inf;
  elem* lt, *rt;
};

typedef elem* t_ptr;

void erase(t_ptr t);
void printLKR(t_ptr t);
void printTREE(t_ptr t, int n);
void insert(t_ptr & t, int a);
t_ptr find(t_ptr t, int a);
void deleteEl(t_ptr &t, int a);
void deleteNode(t_ptr &t);
void delLeftLeaf(t_ptr &t, int &repl);

int main(){
  int n, a;
  t_ptr tree;
  cout << "Number of elements? (Cardinality) ";
  cin >> n;
  tree = nullptr;
  for (int i = 0; i < n; ++i){
    cout << "Input element ";
    cin >> a;
    insert(tree, a);
  }
  cout << endl << "Infix bypass" << endl;
  printLKR(tree);
  cout << endl << "Print Tree" << endl;
  printTREE(tree, 0);
  if (find(tree, 5)){
    cout << "Item is found" << endl;
    deleteEl(tree, 5);
  }
  else
    cout << "Item is not found" << endl;
  printTREE(tree, 0);
  erase(tree);
  return 0;
}
```

```
void insert(t_ptr & t, int a){
  if (!t) {
    t = new elem;
    t->inf = a;
    t->lt = 0;
    t->rt = 0;
  }
  else {
    if (a< t->inf)
      insert(t->lt, a);
    else
      insert(t->rt, a);
  }
}

t_ptr find(t_ptr t, int a){
  if (!t)
    return nullptr;
  if (a == t->inf)
    return t;
  if (a< t->inf)
    return find(t->lt, a);
  return find(t->rt, a);
}

void deleteEl(t_ptr &t, int a) {
  if (t != nullptr)
  if (a == t->inf)
    deleteNode(t); // узел найден - удаляем
  else if (a<t->inf)
    deleteEl(t->lt, a); // рекурсия
  else
    deleteEl(t->rt, a); // рекурсия
}

void deleteNode(t_ptr &t){
  // 1. корень является листом
  // 2. у корня нет левого поддерева
  // 3. у корня нет правого поддерева
  // 4. корень имеет два поддерева

  t_ptr delNode;
  int repl;
  if ((t->lt == nullptr) && (t->rt == nullptr)) {// 1
    delete t;
    t = nullptr;
  }
  else if (t->lt == nullptr) {// 2
    delNode = t;
    t = t->rt;
```

```
    delete delNode;
  }
  else if (t->rt == nullptr) {// 3
    delNode = t;
    t = t->lt;
    delete delNode;
  }
  else {// 4
    delLeftLeaf(t->rt, repl);
    t->inf = repl;
  }
}

void delLeftLeaf(t_ptr &t, int &repl) {
  if (t->lt == nullptr) {// у самого левого нет левых потомков
    repl = t->inf;
    t_ptr delNode = t;
    t = t->rt;
    delete delNode;
  }
  else
    delLeftLeaf(t->lt, repl);
}
```

Бинарное дерево является *деревом поиска*, если для каждого узла выполняются следующие условия: все элементы его левого поддерева меньше значения узла, все элементы правого поддерева не меньше значения узла. Поэтому операция вставки должна сначала найти место новому узлу. Рекурсивная реализация алгоритма вставки представлена в функции `insert`.

```
void insert(t_ptr & t, int a){
  if (!t) {
    t = new elem;
    t->inf = a;
    t->lt = nullptr;
    t->rt = nullptr;
  }
  else {
    if (a< t->inf)
      insert(t->lt, a);
    else
      insert(t->rt, a);
  }
}
```

Алгоритм поиска является частью алгоритма вставки элемента в дерево поиска.

```
t_ptr find(t_ptr t, int a){
  if (!t)
    return nullptr;
  if (a == t->inf)
    return t;
  if (a< t->inf)
    return find(t->lt, a);
  return find(t->rt, a);
}
```

Если проанализировать порядок рекурсивных вызовов, то можно увидеть, что алгоритмы поиска и вставки имеют простые итеративные решения.

 Реализовать итеративные решения для алгоритмов поиска и вставки элемента в дерево поиска.

Функция удаления дерева и функции печати элементов дерева поиска не имеют особенностей. Поэтому использована их реализация из предыдущих примеров.

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

При этом функция `printLKR` всегда выводит элементы дерева поиска в отсортированном виде.

Удаление элемента из дерева поиска является более сложной задачей. При удалении должно быть сохранено основное свойство дерева поиска. Существует несколько вариантов решения этой задачи. В примере рассмотрен один из вариантов. Для удобства решение разделено на несколько подзадач, каждая из которых реализована отдельной функцией.

```
void deleteEl(t_ptr &t, int a);
void deleteNode(t_ptr &t);
void delLeftLeaf(t_ptr &t, int &repl);
```

Функция `deleteEl` находит узел с удаляемым значением и вызывает для него функцию `deleteNode`. Функция определяет к какому из типов

относится удаляемый узел (лист, узел только с правым поддеревом, узел только с левым поддеревом, узел с двумя поддеревьями). В зависимости от этого применяется нужный способ удаления. Удаление листа не имеет особенностей и аналогично удалению последнего элемента односвязного списка.

```
if ((t->lt == nullptr) && (t->rt == nullptr)) { // 1
  delete t;
  t = nullptr;
}
```

В случае когда узел имеет одно поддерево (левое или правое) ссылка на него `t` заменяется ссылкой на непустое поддерево (`t = t->rt` или `t = t->lt`), а узел удаляется.

```
if (t->lt == nullptr) {// 2
  delNode = t;
  t = t->rt;
  delete delNode;
}
else if (t->rt == nullptr) {// 3
  delNode = t;
  t = t->lt;
  delete delNode;
}
```

Если узел имеет оба поддерева, то для его удаления вызывается функция `delLeftLeaf`. Эта функция ищет самый левый узел в правом поддереве удаляемого узла и запоминает его значение в параметре `repl`. Затем удаляет найденный самый левый узел.

```
void delLeftLeaf(t_ptr &t, int &repl) {
  if (t->lt == nullptr) {
  // у самого левого нет левых потомков
    repl = t->inf;
    t_ptr delNode = t;
    t = t->rt;
    delete delNode;
  }
  else
    delLeftLeaf(t->lt, repl);
}
```

После выполнения `delLeftLeaf` функция `deleteNode` заменяет значение удаляемого элемента на найденное значение `repl`.

ГЛАВА 6. КЛАССЫ И ОБЪЕКТЫ

6.1. Основы создания классов

Попытаемся ответить на вопрос: в чем же состоит принципиальное отличие языка C++ от языка C? Сошлёмся при этом на одного из специалистов по языку C++ Брюса Эккеля [18]: «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 *класса*. В результате чего его поля (характеристики) и функции (поведение) рассматриваются как единое целое. Такое объединение получило название *инкапсуляция* (encapsulation). При этом существует возможность регламентировать доступ к членам класса (полям и функциям).

Существует три спецификатора доступа:

`private` (закрытый) — доступен только для членов класса;

`protected` (защищённый) — 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. Если спецификатор при объявлении опущен, то используется спецификатор по умолчанию.

Для объявления класса можно использовать либо ключевое слово `struct`, либо ключевое слово `class`. Рекомендуется использовать слово `class`. Различие проявляется в спецификаторах доступа по умолчанию (табл. 8).

Таблица 8

Различие в доступе по умолчанию

Описание	`class <имя класса> { <поля и функции> };`	`struct <имя класса> { <поля и функции> };`
Спецификатор доступа по умолчанию	`private`	`public`

Следующие два объявления эквивалентны:

```
struct A{
  int f();
  void g();
private:
  int i, j, k;
};

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();
}
```

☺ Рекомендуется всегда явно задавать спецификатор доступа к членам класса.

Приняты два варианта стиля объявлений класса:

- ✓ сначала располагаются поля, потом — функции;
- ✓ сначала располагаются функции, потом — поля.

☺ Рекомендуется в одной программе придерживаться одного из вариантов стилей объявлений.

Класс определяет новый тип данных. Переменная данного типа называется *объектом*. Объект также называют *экземпляром класса*.

Пример 75. Реализовать класс для описания геометрической фигуры «круг». Определить конструкторы, функции-члены класса для вычисления площади, функции доступа к радиусу. Реализовать сравнение объектов на равенство.

```
//circle.h

#pragma once

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(circle a);
  bool operator ==(circle a);
};

//circle.cpp

#include <cmath>
#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(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(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(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 функций.

☺ В C++ рекомендуется реализацию функций 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 функции этого класса — открытые.

Среди функций присутствуют:

- ✓ два конструктора

```
circle();
circle(double x1, double y1, double r1);
```

- ✓ функция вычисления площади

```
double s();
```

- ✓ пара функций для установки доступа к закрытому полю класса (их называют сеттеры и геттеры соответственно)

```
void set_r(double r1);
double get_r();
```

- ✓ две функции для сравнения двух объектов на равенство – один реализован как функция, другой как перегруженная операция

```
bool equal(circle a);
bool operator ==(circle a);
```

В C++ для пользовательских типов данных возможна собственная реализация стандартных операций, которая называется *перегрузкой операций*. Перегруженная операция — это функция, имя которой состоит из слова `operator` и следующего за ним символа операции. Использовать перегруженную операцию можно двумя способами:

- ✓ традиционное использование операции

```
cout<<(a==b)<<endl;
```

- ✓ вызов функции

```
cout<<a.operator==(b)<<endl;
```

☺ В основном поля рекомендуется делать закрытыми, интерфейсные функции — открытыми, а служебные функции — закрытыми.

Реализация функций расположена в отдельном cpp-файле. При реализации вне описания классов имена функций должны быть уточнены

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 функции класса `circle` являются функциями объектов (экземпляров класса). Они вызываются через точку `a.s()`:

```
cout<<a.s()<<endl<<b.s()<<endl;
```

6.2. 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 инициализации полей объекта начальными значениями, он вызывается в момент создания объекта.

В примере 75 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.

Если явно не описан ни один конструктор, компилятор автоматически (неявно) сгенерирует конструктор по умолчанию. Поведение неявно созданного конструктора будет таким же, как если бы он был объявлен явно без списка параметров и с пустым телом.

Когда программа достигает точки выполнения, в которой определяется объект

```
circle a,b(0,0,1);
```

происходит *автоматический вызов конструктора.*

Синтаксис *деструктора* в целом схож с синтаксисом конструктора: имя функции тоже определяется именем класса. Но чтобы деструктор отличался от конструктора, его имя начинается с префикса ~ (тильда). Кроме того, деструктор всегда один и у него нет аргументов. В случае отсутствия явного задания деструктора компилятор неявно создаёт деструктор с пустым телом.

Деструктор автоматически вызывается компилятором при выходе объекта из области видимости или при уничтожении объекта, размещённого в динамической памяти. При этом очищается память, занимаемая объектом. Если перед уничтожением объекта необходимо освободить используемые ресурсы (например, динамическую память для размещения полей объекта, открытые файлы и т. д.), то необходимо явно определить деструктор, выполняющий эти действия.

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`).

Чтобы лучше понять механизмы вызовов конструкторов и деструкторов, можно включать в их код операторы вывода.

 Добавьте в конструкторы и деструктор класса `circle` операторы вывода информации о том, кто из них сработал. Попробуйте объяснить, почему количество сообщений от деструктора превышает количество сообщений от конструкторов.

Пример 76. Реализовать два класса для описания геометрических фигур «круг» и «прямоугольник». Описать функцию, позволяющую совместить центры круга и прямоугольника (переместить круг), если круг можно поместить в прямоугольник.

```
//shape.h

#pragma once

class rectangle;

class circle {
private:
  double x, y, r;
public:
  circle();
  circle(double x1, double y1, double r1);
  void print();
  friend bool tofit(circle &a, rectangle b);
};

class rectangle {
private:
  double x1, y1, x2, y2;
public:
  rectangle();
  rectangle(double x1, double y1, double x2, double y2);
  double width();
  double height();
  void print();
  friend bool tofit(circle &a, rectangle b);
};

//shape.cpp

#include <cmath>
#include <iostream>
#include "shape.h"

using namespace std;
```

```
circle::circle(){
  x = 10;
  y = 10;
  r = 10;
}

circle::circle(double x1, double y1, double r1){
  x = x1;
  y = y1;
  r = r1;
}

void circle::print(){
  cout << x << ' ' << y << ' ' << r << endl;
}

rectangle::rectangle(){
  x1 = 0;
  y1 = 0;
  x2 = 10;
  y2 = 10;
}

rectangle::rectangle(double x1,double y1,double x2,double y2 ){
  this->x1 = x1;
  this->y1 = y1;
  this->x2 = x2;
  this->y2 = y2;
}

double rectangle::width(){
  return abs(x2 - x1);
}

double rectangle::height(){
  return abs(y2 - y1);
}

void rectangle::print(){
  cout << x1 << ' ' << y1 << ' ' << x2 << ' ' << y2 << endl;
}

bool tofit(circle & a, rectangle b){
  if (b.width() >= 2*a.r && b.height() >= 2*a.r){
    a.x = (b.x1 + b.x2) / 2;
    a.y = (b.y1 + b.y2) / 2;
    return true;
  }
  return false;
}
```

```
//главный файл – main.cpp

#include "shape.h"
#include <iostream>

using namespace std;

int main(){
  setlocale(0, "Russian");
  circle a(0, 0, 5);
  rectangle b(10,10,20,20);
  a.print();
  b.print();
  if (tofit(a, b)) {
    cout << "перемещение выполнено" << endl;
    a.print();
    b.print();
  }
  else
    cout << "круг не помещается в прямоугольник" << endl;
  return 0;
}
```

В заголовочном файле shape.h размещены описания двух классов `circle` и `rectangle`, а их определения в файле shape.cpp.

В конструкторе с параметрами для класса `rectangle` возникает ситуация, когда имена параметров совпадают с именами переменных-членов класса. Для разрешения коллизии имён используется ключевое слово `this`.

```
rectangle::rectangle(double x1,double y1,double x2,double y2 ){
  this->x1 = x1;
  this->y1 = y1;
  this->x2 = x2;
  this->y2 = y2;
}
```

Ключевым словом `this` обозначается указатель на объект в функциях-членах класса. Если коллизии имён не возникают, то при обращении к членам класса его обычно опускают. Указатель на объект передаётся как неявный параметр во все функции, которые могут вызываться только для экземпляров классов (в том числе, конструкторы и деструкторы).

6.3. Дружественные функции

В файле shape.cpp расположена функция `tofit`.

```
bool tofit(circle & a, rectangle b){
  if (b.width() >= 2*a.r && b.height() >= 2*a.r){
    a.x = (b.x1 + b.x2) / 2;
    a.y = (b.y1 + b.y2) / 2;
    return true;
  }
  return false;
}
```

В функции `tofit` используются не только функции этих классов, но и их поля. При этом поля описаны как `private`, т. е. защищены от внешнего доступа. Чтобы разрешить внешней функции доступ к защищённым полям объектов, необходимо её сделать дружественной для этих классов. Для этого описание функции со спецификатором `friend` помещается в интерфейс каждого из классов:

```
friend bool tofit(circle &a, rectangle b);
```

В соответствии с принципом инкапсуляции работа с защищёнными членами класса должна быть организована через открытые функции. В рассматриваемом примере можно было бы организовать геттеры и сеттеры для доступа к закрытым членам классов. Если больше ни для каких других целей он не нужны, то вместо них можно использовать механизм дружественности. Важно, что при этом сокращается количество вызовов функций.

Объявление функции дружественной классу разрешает доступ к защищённым полям класса только для этой функции.

Дружественными по отношению к рассматриваемому классу могут быть не только внешние функции, но и функции другого класса, или даже другой класс (все его функции).

На функции, объявленные дружественными, не распространяются действия спецификаторов (`public`, `private`, `protected`).

☺ Объявление функций дружественными рекомендуется располагать либо в самом начале, либо в самом конце описания класса.

Параметрами функции `tofit` являются объекты обоих классов. Поскольку объявление дружественности для `tofit` расположено в интерфейсах обоих классов, возникает проблема очерёдности их описания. Решением является использование предварительного описания одного из классов.

```
class rectangle;
```

Пример 77. Используя классы, описанные в примере 76, создать объект «прямоугольник» и массив объектов класса «круг». Посчитать количество кругов, которые можно поместить внутри прямоугольника. Для таких кругов совместить их центры с центром прямоугольника.

```
#include "shape.h"
#include <iostream>
#include <cstdlib>  //содержит srand() и rand()
#include <ctime>    //содержит time()

using namespace std;

int main(){
  setlocale(0, "Russian");
  rectangle b(10, 10, 40, 40);
  circle* c[5];
  srand((unsigned)time(0));
  for (int i = 0; i < 5; ++i) {
    c[i] = new circle(rand()%100, rand()%100, rand()%10 + 10);
    c[i]->print();
  }
  int count = 0;
  for (int i = 0; i < 5; ++i)
  if (tofit(*c[i], b))
    count++;
  cout << "количество перемещённых = " << count << endl;
  for (int i = 0; i < 5; ++i)
    c[i]->print();
  for (int i = 0; i < 5; ++i)
    delete c[i];
}
```

В условии задачи требуется создать массив объектов класса `circle`. Если бы описание массива выглядело таким образом

```
circle c[5];
```

то все объекты были бы созданы с помощью конструктора по умолчанию, т.е. с одинаковыми значениями переменных-членов класса. Чтобы иметь возможность для каждого элемента массива вызывать конструктор с параметрами, необходимо разместить их в динамической памяти, т.е. при объявлении использовать массив указателей.

```
circle* c[5];
```

Затем для каждого элемента массива создать объект класса `circle`.

```
for (int i = 0; i < 5; ++i) {
  c[i] = new circle(rand()%100, rand()%100, rand()%10 + 10);
  c[i]->print();
}
```

В конце программы динамическая память освобождается.

```
for (int i = 0; i < 5; ++i)
  delete c[i];
```

Пример 78. 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; << вывод массива в поток.

```
//classArray.h

#pragma once

#include <iostream>
using namespace std;

class Array {
private:
  int n;
  int *A;
public:
  Array();     //конструктор по умолчанию
  //конструктор с параметром по умолчанию
  Array(int _n, int x = 0);
```

```
  Array(const Array &B);      //конструктор копии
  int length() const; //функция для нахождения размера массива
  void resize(int nsize);     //изменение размера массива
  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);
};

//classArray.cpp

#include "classArray.h"

Array::Array() {
  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) {
  n = B.n;
  A = new int[n];
  for (int i = 0; i<n; i++)
    A[i] = B.A[i];
}

int Array::length() const{
  return n;
}

void Array::resize(int nsize) {
  int * ndata = new int[nsize];
  int sz = (n < nsize) ? n : nsize;
  for (int i=0;i<sz;++i)
    ndata[i] = A[i];
  delete[] A;
  A = ndata;
  n = nsize;
}
```

```
Array Array::operator + (const Array &B) {
  if (n != B.n)
    throw (1);
  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) {
  if (n != B.n)
    throw (1);
  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(){
  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(){
  setlocale(0, "Russian");
  Array Q(10, 5);
  Array P(10);
  Array W, E;
  Array R(Q);
  cout << "Q " << Q;
  cout << "P " << P;
  cout << "W " << W;
  cout << "E " << E;
  cout << "R " << R;
  cout << "срединные элементы массива Q "
       << Q[Q.length() / 2 - 1] << ' '
       << Q[Q.length() / 2] << endl;

  for (int i = 0; i < Q.length(); ++i)
    Q[i] = Q.length() - 1 - i;
  cout << "изменённый Q " << Q;
  cout << "срединные элементы массива Q "
       << Q[Q.length() / 2 - 1] << ' '
       << Q[Q.length() / 2] << endl;

  for (int i = 0; i < E.length(); ++i)
    E[i] = i;
  cout << "изменённый E " << E;
  W = Q + 15;
  cout << "изменённый W " << W;
  P = W + R;
  cout << "изменённый P " << P;
  E += P;
  cout << "изменённый E " << E;
  cout << "срединные элементы массива P "
       << P[P.length() / 2 - 1] << ' '
       << P[P.length() / 2] << endl;
  Array Z(15);
  try {
    Z += Q;
    cout << "изменённый Z " << Z;
  }
  catch (int) {
    cout << "Массивы имеют разную длину" << endl;
  }
  return 0;
}
```

Когда классы имеют сложную структуру, удобно использовать UML диаграммы классов, которые наглядно показывают их внутреннюю организацию. В частности, на диаграммах представляются члены-данные (поля) и члены-функции (методы) с указанием в виде пиктограмм областей видимости. Пример UML диаграммы для класса `Array` представлен на рисунке 6.1.

На диаграмме не отображается перегруженная операция вывода в поток, реализованная с помощью дружественной функции, поскольку она не является членом класса.

Обычно разработку классов начинают с построения UML диаграмм. Помимо этого, UML диаграммы можно использовать при описании требований к задаче.

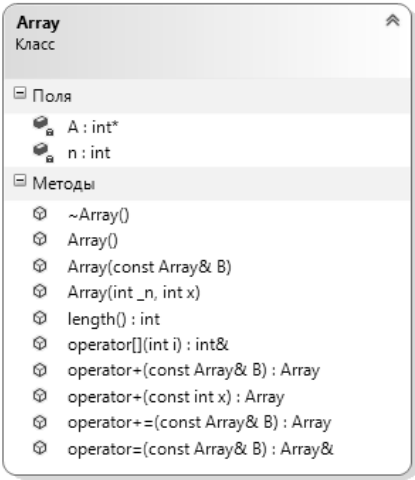

Рис.6.1. UML диаграмма класса `Array`

Для класса `Array` размер динамического массива хранится в приватной переменной-члене класса `n`. Открытая функция-член класса `length()` предназначена для получения длины динамического массива.

Функция `length()` объявлена константной, так как она не изменяет вызывавший её объект.

```
int Array::length() const{
  return n;
}
```

Динамический массив подразумевает, что у него может меняться размер в ходе выполнения программы. Для это в классе предусмотрена функция `resize(int)`.

```
void Array::resize(int nsize) {
  int * ndata = new int[nsize];
  int sz = (n < nsize) ? n : nsize;

  for (int i=0;i<sz;++i)
    ndata[i] = A[i];

  delete[] A;
  A = ndata;
  n = nsize;
}
```

Для создания объектов класса `Array` используются разные определения. Каждому виду определения соответствует свой конструктор.

```
Array Q(10,5); //конструктор с двумя параметрами
Array P(10); //конструктор с параметром по умолчанию.
//Второй параметр задан неявно
Array W,E; //конструктор по умолчанию
Array R(Q); //конструктор копии
```

Конструктор копии, или *копирующий конструктор*, создаёт копию объекта, передаваемого в конструктор в качестве параметра. Кроме явного вызова конструктора копии при создании объекта, он неявно вызывается каждый раз, когда объект передаётся в функцию в качестве параметра по значению или функция возвращает объект.

Поясним эти три случая, когда вызывается конструктор копии.

- Явное создание нового объекта как копии существующего:

```
Array myv1 (Q);
Array myv2 = myv1;
```

- Вызов функции с передачей параметра по значению:

```
void f(Array arr) {/* … */}
```

- Возврат объекта из функции по значению:

```
Array g() {/* … */}
```

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 переменных-членов класса. Такое копирование называется поверхностным.

В классе `Array` для размещения массива используется динамическая память, адрес которой хранится в переменной `A`. В случае использования конструктора копии, создаваемого автоматически, вместо копии массива будет создана копия ссылки на него, т. е. ссылки двух разных объектов будут указывать на одно и то же место в памяти. Для классов, размещающих данные в динамической памяти, необходим конструктор копии.

☠ Для классов, использующих размещение данных в динамической памяти, отсутствие конструктора копии является грубой ошибкой.

Для класса `Array` реализация конструктора копии выглядит следующим образом:

```
Array::Array (const Array &B){
  n=B.n;
  A=new int[n];
  for (int i=0; i<n; i++)
    A[i]=B.A[i];
}
```

Завершая обсуждение конструктора копии, рассмотрим следующую ситуацию.

```
class Array {
  ...
public:
  ...

};
```

```
Array g() {
  return Array();
}

int main(){
  Array myv1 = g();
}
```

Возникает вопрос — сколько раз будет вызван конструктор копии? Здесь присутствуют случаи 1 и 3 вызова конструктора копии, а значит должны создаваться две копии. Чтобы проверить, так ли это, можно добавить в конструктор копии вывод сообщения о выполнении конструктора.

На самом деле в большинстве случаев, запуск данного примера покажет, что во время выполнения программы конструктор копии не будет вызван ни разу. Это объясняется работой оптимизирующего компилятора. Заметим, что такая оптимизация может существенно повлиять на поведение программы в случае, когда конструктор копии содержит побочные эффекты (вывод сообщения). Эта оптимизация носит название *Return Value Optimization (RVO)* и выполняется по умолчанию подавляющим большинством современных компиляторов. Требование выполнения RVO по умолчанию явно оговорено в стандарте языка.

Если специальными ключами компиляции запретить RVO, то вызов конструктора копии будет выполнен ровно два, как и ожидалось. Однако в реальных программах просто стараются не помещать в конструктор копии дополнительный код для реализации побочного эффекта.

Если в классе используется выделение памяти, то её необходимо освободить средствами этого же класса. Это должен делать деструктор.

```
Array::~Array( ){
  delete[] A;
}
```

Напомним, что деструктор вызывается компилятором автоматически для каждого созданного объекта.

☺ В случае использования динамической памяти в реализации класса — настоятельно рекомендуется реализовать конструкторы и деструктор. Среди конструкторов обязательно должен присутствовать конструктор копии.

Остальные функции-члены класса `Array` реализуют перегрузку операций.

6.4. Перегрузка операций

Перегружать операции можно только для определённых пользователем типов, т. е. для классов. Операции для стандартных типов перегружать нельзя. Перегруженная операция должна иметь хотя бы один операнд, тип которого определён пользователем.

Для определения перегруженных операций используется специальная функция с именем `operator@`, где `@` — идентификатор перегружаемой операции. В примере 75 для класса `circle` была перегружена операция сравнения на равенство (= =):

```
bool circle::operator==(circle a){
  return equal(a);
}
```

Ключевое слово `operator` служит для использования операций в функциональном стиле.

Вызов перегруженной операции можно выполнять двумя способами:

- ✓ используя функциональный стиль `a.operator==(b)`, например:

```
cout<<a.operator==(b)<<endl;
```

- ✓ используя синтаксис операции `a==b`, например:

```
cout<<(a==b)<<endl;
```

Попробовать убрать скобки в примере 75 в операторе вывода в поток `cout<<(a==b)<<endl;`
Объяснить, почему они необходимы.

Хотя C++ позволяет перегружать почти все операции, доступные в C++, возможности перегрузки ограничены. В частности, отсутствует возможность изменения приоритета или количества аргументов у операций. Кроме того, рекомендуется не изменять семантику операций.

Количество аргументов в списке перегруженной операции зависит от двух факторов:

- ✓ категории операции — унарная или бинарная;
- ✓ 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).

При определении функции-члена класса для бинарной операции объект, для которого она будет вызываться, всегда будет её левым операндом.

Рассмотрим реализацию перегрузки бинарной операций + для класса `Array` в примере 78. Эта операция может быть реализована функцией-членом класса с одним аргументом. Напомним, что для всех функций-членов класса первым параметром всегда неявно передаётся указатель на объект, для которого вызывается функция. Именно поэтому у бинарной операции всего один аргумент. Операция сложения перегружается для двух списков параметров.

Если параметром является целое число, то операция предназначена для увеличения всех элементов массива на заданное значение.

```
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;
}
```

Данная перегруженная операция сложения является некоммутативной, поскольку её левый операнд — это объект класса `Array`, а правый — целое число.

Операция сложения двух объектов класса `Array` выглядит следующим образом:

```
Array Array::operator + (const Array &B){
  if (n != B.n)
    throw (1);
  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;
}
```

Хотя предполагается, что эта операция будет использована только для массивов одинаковой размерности, функция начинается с проверки корректности вызова и, в случае несоответствия, выбрасывается исключение. Чтобы отслеживать эту ошибку, операцию следует использовать внутри блока `try catch`.

```
Array Z(15);
try {
  Z = Z + Q;
  cout << "изменённый Z " << Z;
}
catch (int) {
  cout << "Массивы имеют разную длину" << endl;
}
```

6.5. Перегрузка операции присваивания

Рассмотрим перегрузку операции присваивания. Она занимает важное место при реализации классов, использующих динамическую память. Присваивание, так же, как и конструктор копии, по умолчанию реализуется побайтным копированием, которое подразумевает только копирование значений полей объекта. Поэтому при использовании динамической памяти в классе произойдёт копирование ссылки на неё, а сама область динамической памяти станет разделяемой между двумя объектами. Такое

разделение доступа к памяти в C++ при работе с указателями не является корректным. В частности, это приведёт к ошибке двойного освобождения памяти. Рассмотрим в качестве примера фрагмент кода

```
{
  Array myv1;
  Array myv2;
  myv2 = myv1;
...
}
//здесь неявно вызываются деструкторы для объектов myv1 и myv2
...
```

В этом фрагменте после присваивания `myv2 = myv1` переменные `myv2` и `myv1` будут разделять общую область. Кроме того за пределами блока, где были объявлены эти переменные, для них будут неявно вызваны деструкторы и, соответственно, произойдёт ещё и утечка памяти, так как старое значение указателя `A` в объекте `myv2` потеряется.

☺ В случае использования динамической памяти в реализации класса настоятельно рекомендуется перегружать операцию присваивания.

Функция `operator=` должна быть обязательно функцией класса. В определении функции `operator=` необходимо скопировать всю необходимую информацию из правостороннего объекта в левосторонний.

```
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;
}
```

Сначала следует проверить, не происходит ли самоприсваивание объектов (`A=A`). Это позволяет избежать выполнения бесполезных операций.

```
if (this != &B) {…}
```

Поскольку размерности полей, размещённых в динамической памяти, у левостороннего и правостороннего операторов могут не совпадать, рекомендуется очистить динамическую память левостороннего операнда и заново выделить память нужного размера.

```
delete[] A;
n=B.n;
A=new int[n];
```

☠ Игнорирование проверки самоприсваивания в случае использования динамической памяти в реализации класса может привести к потере данных.

Напомним, что каждая операция в C++ возвращает значение. Операция присваивания возвращает значение, совпадающее со значением левого операнда после выполнения операции. Поскольку рекомендуется не изменять семантику при перегрузке операции присваивания необходимо вернуть значение левого операнда. Левый операнд — это объект, на который указывает неявный параметр `this`.

```
return *this;
```

В данной реализации операции могут появиться проблемы при возникновении исключения в операции `new` (такое исключение — это не такая уж редкая ситуация). Поскольку к этому моменту уже выполнена операция

`delete[] A`, после возникновения исключения в операции `new` объект останется в «полуразрушенном» состоянии. В C++ выделяют три уровня гарантий безопасности кода при возникновении исключений:

- ✓ базовый — при возникновении исключения не возникает утечек ресурсов, однако объекты могут находиться в непредсказуемом состоянии;
- ✓ строгий — если во время операции произошло исключение, то объект будет находиться в том же состоянии, что до начала операции;

✓ без исключений — в данном коде не может возникнуть исключений.

Приведённая реализация `operator=` даёт лишь базовую гарантию. Достаточно несложно изменить её на строгую. Заведём вспомогательную переменную `newdata` для результата выделения памяти, а операцию удаления `A` перенесём в конец функции.

```
Array Array::operator = (const Array &B) {
  if (this != &B) {
    n=B.n;
    int * newdata = new int[n];
    for (int i=0; i<n; i++)
      newdata[i]=B.A[i];
    delete[] A;
    A = newdata;
  }
return *this;
}
```

Заметим, что обе версии реализации `operator=` выполняют действия, которые используются в других функциях, а именно в деструкторе и в конструкторе копии.

Идиома *copy-and-swap* опирается на это наблюдение и предполагает реализацию операции копирующего присваивания с использованием конструктора копий. При этом требуется вначале создать вспомогательную функцию-член `swap(Array & other)`, для обмена текущего объекта с объектом `other`.

```
void Array::swap(Array & other) {
  swap(n, other.n);
  swap(A, other.A);
}

Array& Array::operator=(Array other) {
//вызов конструктора копии
  this -> swap(other);
  return *this;
}
```

☺ Идиома copy-and-swap позволяет разрабатывать устойчивые к исключениям операторы присваивания и сокращает количество кода в них ценой определения полезной вспомогательной функции `swap`.

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 78

```
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.

6.6. Перегрузка операции индексирования

Перегрузка операции индексирования также обладает некоторыми особенностями. Её нужно реализовывать только как функцию-член класса. Важно, чтобы перегруженная операция доступа к элементу массива по индексу [ ] возвращала ссылку на элемент массива. Это обусловлено требованиями к соблюдению семантики.

```
int& Array::operator [] (int i){
  return A[i];
}
```

Перегруженную операцию доступа к элементу массива по индексу [ ] можно использовать как для получения значения элемента массива, так и для его изменения.

```
  for (int i = 0; i < Q.length(); ++i)
    Q[i] = Q.length() - 1 - i;
  cout << "изменённый Q " << Q;
  cout << "срединные элементы массива Q "
       << Q[Q.length() / 2 - 1] << ' '
       << Q[Q.length() / 2] << endl;
  for (int i = 0; i < E.length(); ++i)
    E[i] = i;
```

Возможно, операцию индексирования потребуется использовать в функциях, в которые объект класса `Array` передаётся как константный параметр по ссылке. Например, функция `printAndSum`, которая выведет на экран содержимое нашего вектора и вычислит сумму его элементов.

```
int printAndSum (const Array &v) {
  int sum=0;
  for (int i = 0; i < v.length(); i++){
    cout << v[i] << ' ';
    sum+=v[i];
  }
  return sum;
}
```

При компиляции этой функции возникнет ошибка «`IntelliSense: отсутствует оператор "[]", соответствующий этим операндам: const Array[int]`».

Для корректной компиляции в данном случае необходимо определить вторую функцию перегрузки операции индексирования.

```
int Array::operator [] (int i) const {
  return A[i];
}
```

Несмотря на то, что списки параметров совпадают у обеих операций индексации, перегрузка возможна, поскольку модификатор `const` включается в сигнатуру функции.

Обратите внимание, что аналогичная ошибка возникала бы, если бы функция `length` не являлась константной. Это связано с тем, что для константных объектов можно вызывать только константные функции.

6.7. Перегрузка операций ввода/вывода

Иногда бывает необходимо, чтобы левосторонний операнд был объектом другого класса или примитивного типа. 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 запрещён доступ напрямую к полям класса, объявленным как `private`. Способ решения проблемы — объявить такую операцию дружественной для класса.

В примере 78 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` даёт право функции напрямую обращаться к закрытым членам класса.

 Самостоятельно реализовать для класса `Array` операцию ввода из потока.

Всякий раз, когда нужно перегрузить бинарную операцию для операндов двух разных типов с сохранением коммутативности, часто используют дружественные функции.

Перегрузить операцию сложения, у которой левой операнд — целое число, а правый — объект класса `Array`.

Большинство операций можно перегружать и как внутренние функции, и как внешние, используя дружественность. Однако некоторые операции можно перегружать только одним из способов. Перегрузку только в виде внешней функции требуют все бинарные операции, у которых левый операнд является объектом другого класса или примитивного типа. Перегрузку только в виде внутренней функции требуют следующие операции: присваивания =, индексирования [ ], преобразования типов `type`, вызова функции () и доступ к элементу через указатель –>.

6.8. Перегрузка операций инкремента и декремента

При перегрузке операций инкремента и декремента нужно учитывать, что они имеют две формы префиксную и постфиксную.

Пример 79. Реализовать класс `Time` («Время»). Для него перегрузить операцию инкремента в двух формах (увеличение времени на одну секунду). Требования к классу представлены на UML диаграмме (рис.6.2).

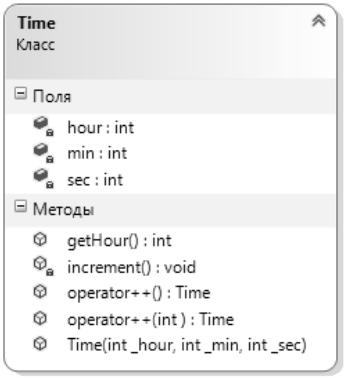

Рис. 6.2. UML диаграмма класса `Time`

```
//classTime.h
#pragma once

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
};
```

Реализация функций класса `Time` располагается в файле classTime.cpp.

```
//classTime.cpp
#include "classTime.h"

Time::Time(int _hour, int _min, int _sec) :hour(_hour),
min(_min), sec(_sec)
{}

int Time::getHour() const{
     return hour;
}

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++;
     }
}
```

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 приведёт 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 класса на чтение (`getXX`- функции) объявлять константными.

Это становится важным, если мы хотим объявить объект-константу для данного класса. Такой объект может использовать только константные член-функции.

Например, можно объявить объект-константу для некоторого эталона времени

```
const Time X(12,0,0);
```

Для объекта `X` можно использовать только функцию `getHour()`.

Уберите квалификатор `const` из функции `getHour()` и попробуйте использовать его для константного объекта.

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;
```

По соглашению, принятому в C++, когда компилятор встречает выражение с префиксным инкрементом `++t`, генерируется вызов функции-элемента `t.operator++()`, объявление которой должно иметь вид:

```
Time operator++();
```

Когда компилятор встречает выражение постфиксной формы инкремента `t++`, он генерирует вызов функции `t.operator++(0)`, объявлением которой является:

```
Time operator++(int);
```

Ноль (0) в генерируемом вызове функции является чисто формальным значением, введённым 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.

Обе формы перегрузки операции ++ используют `private` функцию `increment`. Это связано с нетривиальным алгоритмом увеличения времени на одну секунду.

```
// 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 изменённым временем. Это происходит потому, что текущий объект `*this` возвращается как объект класса `Time`, что активизирует конструктор копии.

```
Time Time::operator++() {
  increment();
  return *this;
}
```

Чтобы эмулировать действие постфиксного инкремента, необходимо вернуть неизменённую копию объекта `Time`. При входе в `operator++` текущий объект `*this` сохраняется во временном объекте `temp`. Затем вызывается `increment()`, чтобы инкрементировать объект. В итоге возвращается неизменённая копия объекта `temp`.

```
// фиктивный целый параметр не имеет имени
Time Time::operator++(int) {
  Time temp(*this);
  increment();
  return 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>
#include "classTime.h"
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;
}
```

6.9. Реализация преобразования типов

В языке C++ определены операции преобразования между встроенными типами данных. А преобразования между встроенными типами и типами, определёнными пользователями, или только между типами, определёнными 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 80. 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*`. Требования к классу представлены на UML диаграмме (рис. 6.3).

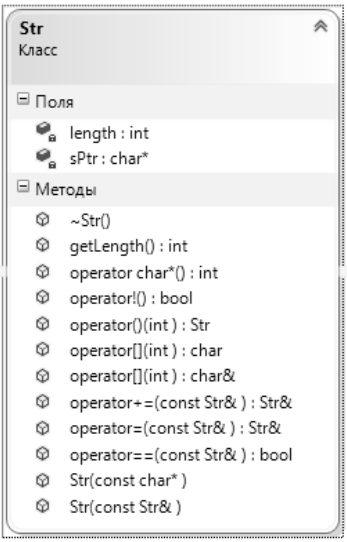

Рис.6.3. UML диаграмма класса `Str`

```
//STR.h
#pragma once

#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
  char operator[] (int) const; //получение символа по номеру
  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
};

//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 {
  if (length!=right.length)
    return 0;
  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 символа
char Str::operator[] (int ind) const{
  //проверка, не находится ли индекс вне диапазона
  assert(ind >= 0 && ind < length);
  return sPtr[ind];      //создание R-величины
}

//получение подстроки заданной длины, начинающейся с заданного
//индекса
Str Str::operator() (int ind, int sublength) {
  //проверка, что 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);
  //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];
  assert(sPtr != 0);
  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 определённого 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 создаёт объект `Str`

```
Str(const char * = "");
```

Конструктор с одним параметром `char*` 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` определённого 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`.

 Уберите из примера 80 перегрузку операции вывода в поток и проверьте работоспособность программы.

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`.

 Напомним, что некоторые преобразования типов могут быть источниками ошибок. Например, при преобразовании значения от вещественного типа к целому может быть получен некорректный результат, хотя преобразование из целого типа в вещественный всегда гарантирует правильный результат. Поэтому преобразование из целого типа в вещественный является безопасным, а обратное — небезопасным. Для выполнения небезопасных преобразований типов используется операция явного преобразования.

Поскольку любой конструктор с одним параметром будет использоваться всегда по умолчанию для неявного преобразования типа, то для запрета неявного преобразования необходимо дополнить такой конструктор модификатором `explicit`.

В классе `Str` конструктор с параметром `char * ` использовался специально для демонстрации возможности неявного преобразования. Поэтому он был описан без модификатора `explicit`.

Чтобы пояснить ситуацию, когда необходимо использовать модификатор `explicit`, рассмотрим следующий пример. Допустим, необходимо к строке, представляемой классом `Str`, дописать константную строку, содержащую число, например номер курса. Сделаем это, используя перегруженную операцию +=.

```
Str s1("Hello");
s1+="2";       // s1 содержит Hello2
```

Предположим, при наборе кода программы сделана ошибка — оказались пропущены кавычки.

```
s1+=2;       // сообщение об ошибке компиляции
```

В нашем примере произойдёт ошибка компиляции, т. к. компилятор не найдёт перегруженной операции += с правым операндом целого типа, и не определено приведение целого типа к типу `Str`.

Если бы в классе `Str` был определён конструктор преобразования из типа `int`, такая операция была бы разрешена, и ошибка компиляции не возникла бы. Предположим, что конструктор с одним параметром типа `int` существует и предназначен для формирования строки с заданным количеством пробелов.

```
Str(int n);    //конструктор неявного преобразования
Str::Str(int n) {
  length = n;
  sPtr = new char[length + 1];
  assert(sPtr != 0);     //завершение, если память не выделена
  for (int i = 0; i < length; ++i)
    sPtr[i] = ' ';
  sPtr[length] = 0;
}
```

По правилам языка C++ он является конструктором преобразования типа, но его назначением является формирование пробельной строки

заданной длины. В этом случае результат операции с пропущенными кавычками окажется неожиданным.

```
s1+=2;       // s1 содержит Hello и еще два пробела
```

Именно поэтому следовало указать компилятору, что мы не хотим рассматривать такой конструктор как конструктор неявного преобразования. В этом случае конструктор нужно объявить с модификатором `explicit`.

```
//конструктор преобразования, вызываемого явно
explicit Str(int n);

Str::Str(int n) {
  length = n;
  sPtr = new char[length + 1];
  assert(sPtr != 0);     //завершение, если память не выделена
  for (int i = 0; i < length; ++i)
    sPtr[i] = ' ';
  sPtr[length] = 0;
}
```

При использовании `explicit` конструктора будут получены следующие результаты:

```
s1+=2;       // сообщение об ошибке компиляции
s1+=Str(2);      //s1 содержит Hello и еще два пробела
```

В C++11 ключевое слово `explicit` применимо и к операторам преобразования. По аналогии с конструкторами оно защищает от непредвиденных неявных преобразований.

6.10. Обработка исключений

Из-за большого количества недиагностируемых ошибок времени выполнения при работе со строками `char*` в примере 80 используется макрос `assert`.

```
void assert(int выражение)
```

Он позволяет включить в программу диагностические сообщения о таких ошибках. 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 ноль, то `assert(выражение)` напечатает сообщение следующего вида:

```
Assertion failed: выражение, file имя файла, line nnn
```

После чего будет вызвана функция `abort`, которая завершит вычисления.

Основное применение макроса `assert` — диагностика ошибок на этапе отладки приложения. В частности, такой механизм широко применяется при создании тестов, в том числе и unit-тестов.

Но в некоторых случаях ошибки времени выполнения могут возникать и в отлаженной программе. Например, ошибки при выделении динамической памяти или несоответствия типов входных данных. Для обработки таких ситуаций используется механизм 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 создаётся 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 (рассмотренные в разделе 1.10), так и создаваемые пользователями классы исключений.

В простейшем случае класс исключения может быть пустым. Этого достаточно, если нам нужно только просигнализировать о возникновении ситуации и никакие данные передавать обработчику не нужно.

Например, создадим класс, описывающий ошибку, возникающую при невозможности выделить динамическую память.

```
class OutOfMemoryException{};
```

Чтобы воспользоваться классом исключения нужно использование `assert` заменить запуском исключения.

```
//assert(sPtr!=0);
if (sPtr==0)
  throw OutOfMemoryException();
```

В этом случае определение перегруженной операции конкатенации будет выглядеть следующим образом:

```
//сцепление (конкатенация)
Str & Str::operator +=(const Str & right) {
  char * tempPtr = sPtr;
  length += right.length;
  sPtr = new char [length+1];
  if (sPtr==0)
    throw OutOfMemoryException();
  strcpy(sPtr, tempPtr);
  strcat(sPtr, right.sPtr);
  delete [] tempPtr;
  return *this;
}
```

Оператор генерации исключения `throw` производит целый ряд действий. Сначала создаётся копия запускаемого объекта, которая возвращается из функции, содержащей `throw`, даже если тип объекта исключения не соответствует типу, который положено возвращать этой функции. При этом управление передаётся 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» (см. раздел 1.10).

Создайте класс исключения `BadIndexException` для контроля выхода индекса за границы. Замените все использования `assert` генерацией исключений соответствующих типов.

Поскольку ситуация, когда память выделить невозможно, встречается крайне редко, рассмотрим ошибку, связанную с индексацией. После проведённых изменений в программе неправильное значение индекса и в случае использования макроса `assert`, и в случае оператора `throw` приводит к аварийному завершению программы. Различаются только сообщения, описывающие причину завершения программы. Но в отличие от использования `assert` сгенерированные исключения можно перехватывать и обрабатывать в программе без обязательного аварийного завершения.

В том случае, когда команда `throw` не должна приводить к аварийному завершению, следует создать специальный блок, который должен реагировать на исключение. 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 определённый 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.

Если в программе запускается исключение, механизм обработки исключений начинает искать первый обработчик с аргументом, соответствующим типу исключения. Управление передаётся 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.

Рассмотрим использование в операции индексирования исключения BadIndexException для контроля выхода индекса за границы строки.

```
//получение ссылки на символ
char & Str::operator[] (int ind){
  //проверка, не находится ли индекс вне диапазона
  if (ind < 0 || ind >= length)
    throw BadIndexException();
  return sPtr[ind];      //создание L-величины
}
```

Тип исключения обычно предоставляет достаточно информации для определения характера ошибки, например:

```
Str s="New string for tests";
for (int i=0; i<10; ++i){
  cout<<"введите количество символов"<<endl;
  cin>> i;
  try {
    cout<<s[i];
  }
  catch (BadIndexException){
    //обработка исключения
  .. cout<<"illegal index"<<endl;
  }
}
```

Рассмотренный цикл при любых входных данных будет выполнен 10 раз, но в случаях выхода за границы вместо символа строки будет выведено сообщение «illegal index».

Иногда кроме типа исключения обработчику желательно передавать дополнительную информацию, например значение индекса, вызвавшего исключение. Тогда для исключений нужно использовать классы, имеющие конструктор с параметрами, поля и методы.

Расширим класс BadIndexException, чтобы можно было передавать значение индекса в обработчик.

```
class BadIndexException {
  private:
    int ind;
  public:
    BadIndexException(int i): ind(i) {}
    int getInd() const {return ind;}
};
```

Теперь при выбрасывании исключения нужно указать значение некорректного индекса.

```
throw BadIndexException(ind);
```

Тогда обработчик будет выглядеть следующим образом:

```
  catch (BadIndexException e){
    //обработка исключения
  .. cout<<"illegal index"<<e.getInd()<<endl;
  }
```

Механизм исключений в первую очередь предназначен для разделения места, где возникает ошибка, и места, где она обрабатывается. Поэтому чаще всего исключения, выбрасываемые в функциях, обрабатываются там, где эти функции вызываются. При этом такие функции могут входить в состав библиотек, поставляемых сторонними разработчиками. При этом могут быть известны только заголовки функций. Чтобы сообщить, какие исключения может выбрасывать функция, в её объявление можно добавить список возможных исключений.

```
char & operator[] (int ind) throw (BadIndexException);
```

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.

6.11. Статические члены класса

Классы могут содержать статические поля и статические функции. Если данные-члены объявлены с квалификатором `static`, то для всех

объектов класса поддерживается только одна копия таких данных. Статический член используется совместно всеми объектами данного класса. Для того чтобы существовал статический член, не обязательно, чтобы существовали объекты такого класса.

Пример 81. Реализовать класс, позволяющий подсчитывать количество существующих объектов данного класса.

```
class Counter {
private:
  static int count;
public:
  static int getCount() {
    return count;
  }
  Counter() {
    ++count;
  }
  ~Counter() {
    --count;
  }
};
int Counter::count = 0;
int main() {
  cout << "before " << Counter::getCount() << endl;
  Counter first;
  cout << "after first " << first.getCount() << endl;
  Counter * second = new Counter();
  cout << "after second " << second->getCount() << endl;
  delete second;
  cout << "after delete second " << Counter::getCount() <<
endl;
  return 0;
}
```

Внутри класса возможно только объявление статического члена, но не его определение.

```
private:
  static int count;
```

Статические данные можно использовать только после определения. Это делается путём нового объявления статической переменной, причём используется оператор разрешения области видимости для того, чтобы идентифицировать тот класс, к которому принадлежит переменная.

```
int Counter::count = 0;
```

Статические члены-данные подчиняются правилам доступа к членам класса. Однако определение статических членов-данных выполняется вне зависимости от спецификатора доступа.

☺ Определение статических членов-данных класса рекомендуется располагать вместе с определением самого класса в заголовочном файле.

К статическим членам класса можно обращать как через класс, так и через объект или указатель/ссылку на объект.

```
cout << "before " << Counter::getCount() << endl;
Counter first;
cout << "after first " << first.getCount() << endl;
Counter * second = new Counter();
cout << "after second " << second->getCount() << endl;
```

Статические функции-члены не могут обращаться к нестатическим данным или вызывать нестатические функции этого же класса. Это связано с тем, что в статические функции не передаётся указатель `this` на объект, для которого вызвана функция.

Реализация статической функции записывается так же, как и реализация любой другой функции-члена класса. При этом если реализация вне класса, то ключевое слово `static` не указывается.

Например, определение статической функции `getCount` вне класса могло бы выглядеть следующим образом

```
int Counter::getCount() {
    return count;
}
```

6.12. Автоматически создаваемые члены класса

Поскольку некоторые члены класса создаются компилятором автоматически, следует понимать принципы их создания и знать ситуации, в которых они создаются. Это позволит в одних случаях избежать ошибок, а в других — эффективно использовать действия по умолчанию.

Рассмотрим следующий «не очень полезный» класс.

```
class Empty {};
```

На самом деле, такое описание класса эквивалентно следующему:

```
class Empty{
public:
  Empty() {}
  Empty(Empty const &) {}
  Empty & operator = (Empty const &) {}
  ~Empty() {}
};
```

Здесь присутствуют четыре функции:

`Empty()` — конструктор без параметров,

`Empty(Empty const &)` — конструктор копий,

`Empty& operator=(Empty const&)` — операция копирующего присваивания,

`~Empty()` — деструктор.

Если класс не имеет членов-данных, то эти функции не выполняют никаких действий.

Напомним, что класс для описания исключения часто можно оставить пустым, используя принцип определения по умолчанию. И тогда его описание будет выглядеть именно таким образом.

Добавим член класса `value` в определение «не очень полезного» пустого класса.

```
class NotQuiteEmpty{
public:
  int value;
};
```

У классов, имеющих члены-данные, конструктор без параметров и деструктор, создаваемые по умолчанию, остаются пустыми. Конструктор копии, создаваемый по умолчанию, выполняет побитовое копирование членов-данных. Операция присваивания, создаваемая по умолчанию, выполняет побитовое присваивание.

Принцип создания функций-членов класса по умолчанию состоит в следующем: если любая из перечисленных функций, кроме конструктора без параметров, явно не задана, она создаётся неявно по умолчанию. Принцип создания функций-членов класса по умолчанию для конструктора без параметров имеет особенность. Такой конструктор создаётся неявно, только если не задан явно ни один конструктор. Если в классе будет явно описан хотя бы один конструктор, неявного создания конструктора без параметров не произойдёт. А это в дальнейшем при использовании класса может привести к ошибкам.

На практике иногда создаваемые неявно функции могут привнести нежелательную функциональность, от которой нужно избавиться. Например, нужно создать класс, для которого должен существовать только один экземпляр. Такие классы называются *синглетонами*. Для них рекомендуется запретить операцию присваивания и конструктор копирования. С другой стороны, рекомендуется наряду с другими конструкторами всегда иметь конструктор без параметров, даже если его тело является пустым.

В C++ предусмотрен механизм управления генерацией стандартных конструкторов и функций. Рассмотрим его на примере класса A:

```
class A {
private:
    int x;
public:
    A(int i) {…}
// Следующая запись указывает на необходимость сгенерировать
// конструктор по умолчанию
    A() = default;

// Запретить генерацию конструктора копии по умолчанию
    A(const A&) = delete;

// А так можно запретить генерацию operator=
    A& operator = (const A&) = delete;
}
```

6.13. Семантика перемещения

Семантика перемещения или move-семантика появилась в стандарте C++11 (ISO/IEC 14882:2011). Она нацелена на уменьшение количества создаваемых копий объектов при выполнении конструктора копии и операции присваивания, которые вызываются для rvalue выражений. Любое выражение в C++ является или левосторонним (lvalue), или правосторонним (rvalue). Выражение lvalue — это объект, который имеет имя. Все переменные являются lvalue. А выражение rvalue — это временный безымянный объект, не существующий за пределами того выражения, которое его создало. Более подробно rvalue и lvalue выражения в C++ рассматривались в разделе 1.6.

Для иллюстрации проблемы лишних копий рассмотрим упрощённую реализацию класса для динамического массива.

Пример 82. Реализовать move-семантику на примере упрощённого класса для динамического массива.

Для этого достаточно описать конструктор с параметрами по умолчанию и конструктор копии, деструктор и перегрузить операцию сложения двух массивов одинакового размера и операцию присваивания. Чтобы отслеживать процесс создания и удаления объектов, в классе добавлены два поля:

- ✓ поле `name` — имя, отражающее способ создания объекта;
- ✓ статическое поле `f` для подсчёта существующих в текущий момент объектов. Значение поля `f` увеличивается каждый раз при выполнении операции `new` и уменьшается при выполнении операции `delete`.

```
#include <iostream>
#include <string>

using namespace std;
```

```
class myvector {
private:
  static int f;
  string name;
  int size;
  int * vect;
public:
  myvector(int s = 1, string nm ="noname"): size(s), name(nm){
    f++;
    cout << "constructor > " << name<<" "<<f<< endl;
    vect = new int[s];
  }

  myvector(const myvector & v): size(v.size){
    f++;
    name = "Copy ( "+v.name+" )";
    cout << "copy > " << name << " " <<  f << endl;
    vect = new int[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
  }

  ~myvector(){
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }

  myvector& operator= (const myvector &v){
    cout<< name <<" operator= > " << v.name <<" "<< f << endl;
    if (&v != this) {
      delete[] vect;
      vect = new int[v.size];
      for (int i = 0; i<v.size; ++i)
        vect[i] = v.vect[i];
      size = v.size;
    }
    return *this;
  }

  myvector operator+(const myvector& v){
    if (size == v.size) {
      myvector v1(size, name + " + "+v.name);
      for (int i = 0; i < size; ++i)
        v1.vect[i] = vect[i] + v.vect[i];
      return v1;
    }
    return *this;//лучше использовать исключение
  }
};
```

В следующей программе выражениями rvalue являются `a + b` внутри вызова конструктора копии для объекта `cc` и `a + D` в операторе присваивания. Именно для них будут создаваться дополнительные временные объекты, которые после использования в конструкторе копии и операции присваивания тут же будут удалены.

```
#include "vect.h"
int myvector::f = 0;
int main() {
  myvector a(3,"A"), b(3,"B"),D(a);
  myvector cc (a + b);
  a = b;
  b = a + D;
  return 0;
}
```

В окне вывода можно проследить последовательность вызовов конструкторов и деструкторов для данной программы (рис. 6.4).

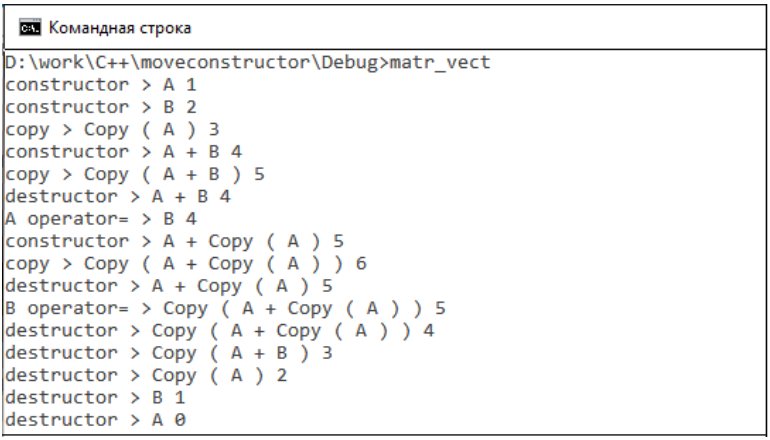

Рис.6.4. Последовательность вызовов конструкторов и деструкторов

Каждая строка содержит информацию о вызываемой функции (конструктор, деструктор, операция присваивания), имени объекта и количестве объектов, существующих в текущий момент. При этом можно заметить создание и уничтожение временных объектов, а также накладные расходы на копирование этих объектов.

Например, при создании объекта `myvector cc (a + b)` сначала создаётся временный объект при вычислении значения `a + b`. Затем вызывается конструктор копии `myvector(const myvector & v)`, в который в качестве значения передаётся, созданный ранее, временный объект с именем `A + B`. После копирования временный объект удаляется. В результате происходит лишнее выделение памяти и копирование данных одного и того же объекта.

Во многих современных компиляторах встроен механизм Return Value Optimization (RVO), решающий, в том числе, и эту проблему. Однако автоматическая оптимизация не всегда бывает эффективной, поэтому в стандарте C++11 Бьярн Страуструп предложил вынести решение на уровень языка [16]. Для этого были введены move-конструктор и move-operator=

Идея move-семантики состоит в том, чтобы не удалять временный объект и не выделять память для полей в новом объекте, а инициализировать поля в создаваемом объекте ссылками на поля временного объекта.

Деструкторы для временных переменных вызываются в тот момент, когда эти переменные уже не используются для вычислений. Однако при использовании move-семантики деструктор не должен освобождать память временного объекта, поскольку ссылкой на неё инициализируется поле в другом объекте. Для этого в move-конструкторе и move-операции присваивания поле указатель временного объекта меняет значение на `nullptr`, а в деструкторе выделенная память освобождается только, если поле указатель не равен `nullptr`.

➢ Конструктор копирования выделяет новую область памяти для хранения данных, вызывая оператор `new`, а перемещающий конструктор — забирает данные у переданного ему временного объекта.

Добавим в класс move-конструктор, move-операцию присваивания и внесём изменения в деструктор.

```
//move-конструктор
myvector(myvector&& v){
  name = "Move_Copy ( " + v.name+" )";
  cout << "move > " << name << " " << f << endl;
  size = v.size;
  vect= v.vect;
  // Не позволит сразу удалить временный объект
  v.vect = nullptr;
}

//измененный деструктор
~myvector(){
  if (vect != nullptr) {
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }
}

//move-операция присваивания
myvector& operator=(myvector&& v){
  cout << name <<" operator-move= > "<< v.name <<" "<< f
<<endl;
  if (vect != nullptr)
    delete[] vect;
  size = v.size;
  vect = v.vect;
  v.vect = nullptr;
  return *this;
}
```

Чтобы отличать функции с перемещающей семантикой в стандарте C++11 введены rvalue ссылки — `myvector && v`. При этом компилятор будет использовать функции с перемещающей семантикой только в случае, если параметром является rvalue выражение (временный объект).

Теперь, при выполнении той же самой программы, для строки `myvector cc(a+b);` компилятор выберет move-конструктор вместо конструктора копии. А для строки `b = a + D;` компилятор выберет move-операцию присваивания. Результат выполнения приведён на рисунке 6.5.

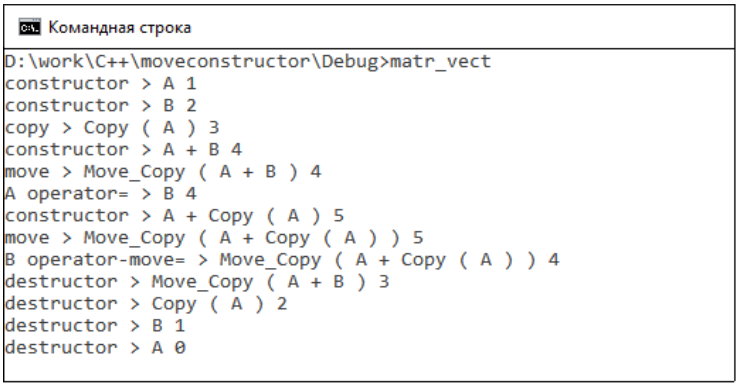

Рис.6.5. Последовательность вызовов для move-семантики

☺ Ввиду наличия большого количества стандартных классов использовать move-конструкторы и move-операции присваивания приходится редко, однако знание такого механизма необходимо.

ГЛАВА 7. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ

7.1. 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 определённого 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, определённые в нём самом. Кроме того, производный класс может изменять (переопределять) любую наследуемую функцию-член.

При использовании механизма наследования в описании класса появляется новый раздел — защищённый (protected). Члены, помещённые в защищённый раздел, доступны из функций классов-наследников, но скрыты вне определения класса. Общие правила доступа относительно разделов класса в языке C++ представлены в таблице 9.

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 9

Правила доступа

Раздел базового класса	Открытый (public)	Защищенный (protected)	Закрытый (private)
Доступность из функций базового класса, функций дружественных классов и из дружественных функций	Да	Да	Да
Доступность из функций классов-наследников базового	Да	Да	Нет
Доступность из других классов и функций	Да	Нет	Нет

Открытое наследование. Открытые и защищённые члены базового класса остаются, соответственно, открытыми и защищёнными членами производного класса.

Защищённое наследование. Открытые и защищённые члены базового класса становятся защищёнными членами производного класса.

Закрытое наследование. Открытые и защищённые члены базового класса становятся закрытыми членами производного класса.

Среди указанных видов наследования открытое наследование является наиболее важным и часто используемым.

7.2. 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 83. Реализовать иерархию классов `Sphere` («сфера») — `Ball` («мяч»), которая представлена на UML диаграмме (рис. 7.1).

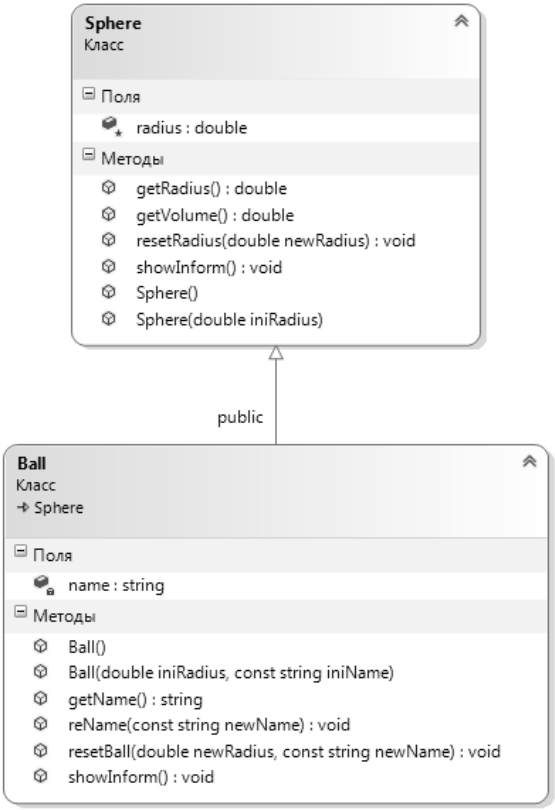

Рис. 7.1. UML диаграмма иерархии классов `Sphere` – `Ball`

Определение класса «сфера» – файл sphere.h.

```
#pragma once

class Sphere {
  public:
    //Конструкторы
    Sphere();
    Sphere(double iniRadius);
```

```
    // Операции
    void resetRadius(double newRadius);
    double getRadius();
    double getVolume();
    void showInform();
  protected:
    double radius; //радиус сферы
};
```

Реализация класса «сфера» – файл sphere.cpp

```
#include <iostream>
#include <cmath>
#include "sphere.h"

using namespace std;

Sphere::Sphere(): radius(1.0){ }

Sphere::Sphere(double iniRadius){
  if (iniRadius>0)
    radius=iniRadius;
  else
    radius=1.0;
}

void Sphere::resetRadius(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 = "<<radius
      <<"\n Volume = "<<getVolume()<<endl;
}
```

Определение класса «мяч» – файл ball.h

```
#pragma once
#include <string>
#include "sphere.h"
using namespace std;
```

```
class Ball: public Sphere{
  public:
    //конструкторы
    Ball();
    Ball(double iniRadius, const string iniName);

    //дополнительные или измененные операции
    string getName();
    void reName (const string newName);
    void resetBall(double newRadius, const string newName);
    void showInform();

  private:
    string name;
};
```

Реализация класса «мяч» – файл ball.cpp

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

Ball::Ball(): Sphere(){
  setName("NoName");
}

Ball::Ball(double iniRadius,const string iniName):
                  Sphere(iniRadius), name(iniName){}

void Ball:: reName(const string newName){
  name = newName;
}

string Ball::getName(){
  return name;
}

void Ball::resetBall(double newRadius, const string newName) {
  radius = newRadius;
  name = newName;
}

void Ball::showInform(){
  cout<<" Ball type - "<<name<<". Specifications:";
  Sphere::showInform();
}
```

📖 Конструктор производного класса выполняется после конструктора базового класса. Это правило действует и в цепочках наследования любой длины.

Например, конструктор класса `Ball` выполняется после конструктора класса `Sphere`.

```
Ball::Ball(): Sphere(){
  setName("NoName");
}
Ball::Ball(double iniRadius, const string iniName):
                  Sphere(iniRadius), name(iniName){}
```

Конструкторы класса `Ball` вызывают соответствующие конструкторы класса `Sphere`. 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. В этом случае конструктор без параметров для класса `Ball` может выглядеть следующим образом:

```
Ball::Ball(){
  setName("NoName");
}
```

Если в классе-потомке не определён ни один конструктор, то будет создан конструктор по умолчанию, который вызовет конструкторы по умолчанию всех предков.

📖 Важно, чтобы в иерархии классов всегда были определены конструкторы по умолчанию.

Конструктор производного класса отвечает за инициализацию всех элементов данных, добавленных к унаследованным данным из базового

класса. Конструктор базового класса выполняет инициализацию унаследованных элементов данных.

Деструктор производного класса выполняется перед деструктором базового класса. В цепочках наследования произвольной длины деструкторы вызываются в порядке обратном порядку вызова конструкторов.

Соответственно, в нашем примере вначале выполнится деструктор класса `Ball`, затем — деструктор класса `Sphere`. Поскольку они явно не определены, то используются автоматические деструкторы.

Наследование не открывает доступ к закрытым членам. Если бы переменная `radius` была объявлена как `private`, она была бы недоступна в классе `Ball`. Тогда внутри класса `Ball` для доступа к ней пришлось бы использовать открытые функции класса `Sphere` — `resetRadius` и `getRadius`. Но всегда следует помнить, что каждый вызов функции приводит к увеличению накладных расходов. Поэтому если наследникам нужен прямой доступ к полям предка, рекомендуют использовать спецификатор доступа `protected`.

```
class Ball{
...
protected:
double radius; //радиус сферы
};
...
void Ball::resetBall(double newRadius, const string newName) {
  radius = newRadius;
  name = newName;
}
```

В реализации класса `Ball` можно вызывать функции, наследуемые от класса `Sphere`. Если бы `radius` был объявлен как `private`, функция `resetBall` должна была бы вызывать наследуемую функцию `resetRadius`.

```
void Ball::resetBall(double newRadius, const string newName) {
  resetRadius(newRadius);
  name = newName;
}
```

Функция `showInform` переопределяется в классе `Ball`. При этом она вызывает унаследованную версию функции класса предка `Sphere::showInform`. Для разрешения коллизии имён используется операция разрешения области видимости (`::`).

```
void Ball::showInform(){
  cout<<" Ball type - "<<name<<". Specifications:";
  Sphere::showInform();
}
```

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 имён. Каждый класс имеет своё пространство имён. Следовательно, производный класс вводит новое пространство имён.

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");
pb.showInform();
cout << pb.getRadius() << endl;
pt->showInform(); cout << pt->getVolume() << endl;
p->showInform(); cout << p->getVolume() << endl;
```

Однако указатель или ссылка базового типа позволяет вызывать только функции базового класса, поэтому воспользоваться `pb`, `pt` или `p` для вызова функции `getName()` производного класса нельзя.

Следует обратить внимание и на то, что вызов функции `showInform()` через указатели `p`, `pt` или ссылку `pb` на базовый класс обратится к реализации этой функции в классе `Sphere`, игнорируя её переопределение в классе `Ball`.

Пример 84. Описать иерархию классов `Person` – `Student`. Класс `Student` не должен иметь доступ к личным данным, содержащимся в классе `Person`. Необходимые методы представлены на UML диаграмме классов (рис. 7.2).

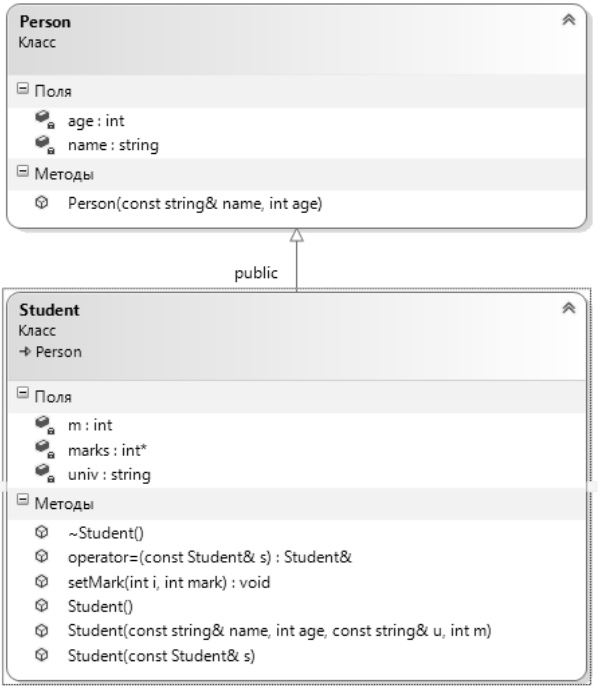

Рис. 7.2. UML диаграмма иерархии классов `Person` – `Student`

```
#pragma once
#include <string>
using namespace std;

class Person {
private:
  string name;
  int age;
public:
  Person(const string& name = "noname", int age = 18):
         name(name), age(age) {}
  friend ostream & operator<<(ostream & os, const Person & p){
    os << p.name << " " << p.age << endl;
    return os;
  }
};

class Student: public Person{
private:
  string univ;    // Университет
  int m;
  int* marks;     // Оценки

public:
  Student() : univ("МГУ"), m(3) {
    marks = new int[m];
    for (int i = 0; i<m; ++i)
      marks[i] = 0;
  }

  Student(const string& name,int age, const string& u, int m):
    Person(name, age), univ(u), m(m) {
    marks = new int[m];
    for (int i = 0; i<m; ++i)
      marks[i] = 0;
  }

  Student(const Student& s): Person(s), univ(s.univ), m(s.m) {
    marks = new int[s.m];
    for (int i =0; i<m; ++i)
      marks[i] = s.marks[i];
  }

  ~Student() { delete[] marks; }
  Student& operator=(const Student& s) {
    if (&s != this) {
      delete[] marks;
      Person::operator=(s);
      univ = s.univ;
      m = s.m;
```

```
      marks = new int[m];
      for (int i = 0; i<m; ++i)
        marks[i] = s.marks[i];
    }
    return *this;
  };

  void setMark(int i, int mark) {
    if (i < 0 || i >= m)
      throw - 1;
    marks[i] = mark;
  }

  friend ostream & operator<< (ostream& os, const Student& s){
    os<<(Person)s;
    os << s.univ << endl;
    for (int i = 0; i < s.m; ++i)
      os << s.marks[i] << " ";
    os << endl;
    return os;
  }
};

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

int main() {
  Student s("Pinokkio", 18, "ЮФУ", 3);
  s.setMark(0, 5);
  s.setMark(1, 4);
  cout << s;
  Student s1(s);
  cout << s1;
  Student s2;
  cout << s2;
  s2 = s1;
  cout << s2;
  system("PAUSE");
  return 0;
}
```

Наличие динамически выделяемой памяти `int* marks` в классе `Student` создаёт дополнительные проблемы. В этом случае нельзя использовать создаваемые по умолчанию функции: конструктор без

параметров, конструктор копии, деструктор и операцию присваивания. Необходимо предусмотреть их реализацию в классе.

Конструктор без параметров использует список инициализаторов для членов-данных `univ`, `m`. Выделение памяти для массива оценок `marks` и его инициализация происходит в теле конструктора.

```
Student(): univ("МГУ"), m(3){
  marks = new int[m];
  for (int i = 0; i<m; ++i)
    marks[i] = 0;
}
```

Инициализация членов-данных, унаследованных от класса `Person`, может выполниться только внутри конструктора класса `Person`. Вызов конструктора предка возможен только в списке инициализаторов конструктора наследника. В данном примере его нет, поэтому будет вызван конструктор по умолчанию класса предка. Если у класса предка нет конструктора по умолчанию, то произойдёт ошибка.

☺ Если предполагается иерархия наследования, рекомендуется для каждого класса в иерархии определять конструктор по умолчанию.

Конструктор с параметрами требует явного указания конструктора предка в списке инициализаторов.

```
Student(const string& name, int age, const string& u, int m):
  Person(name, age), univ(u), m(m){
  marks = new int[m];
  for (int i = 0; i<m; ++i)
    marks[i] = 0;
}
```

Здесь `Person(name, age)` — это вызов конструктора предка. Если не написать вызов `Person(name, age)`, то произойдёт вызов конструктора без параметров, который проинициализирует поля `name` и `age` значениями по умолчанию.

В деструкторе необходимо только освободить память, занимаемую массивом `marks`, а память, выделенная для `univ` и `Person`, будет

освобождена автоматически при вызове соответствующих деструкторов в эпилоге деструктора `~Student()`.

```
~Student() {
  delete[] marks;
  // эпилог
}
```

Рассмотрим конструктор копии класса `Student`.

```
Student(const Student& s): Person(s), univ(s.univ), m(s.m){
  marks = new int[s.m];
  for (int i =0; i<m; ++i)
    marks[i] = s.marks[i];
}
```

Заметим, что `Person(s)` будет работать корректно благодаря тому, что ссылке на объект класса предка можно присвоить ссылку на объект класса потомка. При этом происходит приведение типа-наследника к базовому типу. Такое приведение называется *upcast*.

Рассмотрим преобразование типов в конструкторе копии класса `Student`, которое возникает при вызове конструктора копии `Person(s)` в списке инициализации. Мы фактически параметру типа `const Person &` присваиваем переменную типа `const Student &`. Аналогичные преобразования также могут выполняться как для указателей, так и для самих объектов. Преобразования типов для параметров при вызове функций встречаются достаточно часто, в том числе и для параметра `this`. Например, если вызывается для потомка унаследованная функция предка.

Поскольку передача параметров сводится к выполнению операции присваивания, то рассмотрим преобразование в иерархии «предок-потомок» на примере операции присваивания.

 Для преобразования типов в иерархии «предок-потомок» работает правило: переменной типа предок можно присвоить переменную типа потомок, но не наоборот.

```
Person p("Иванов", 20);
Student s("Петров", 19, "ЮФУ", 3);
p = s;
```

При присваивании объекта производного класса переменной базового класса происходит копирование только полей базового класса, остальная часть информации объекта производного класса будет утеряна.

Попытка присвоить объекту класса наследника объект класса предка приведёт к ошибке компиляции.

```
s = p; // ошибка компиляции
```

При работе с указателями и ссылками на объекты предка и наследника действует аналогичное правило преобразования типов. Указателю или ссылке на объект базового класс можно присвоить адрес объекта или ссылку на объект, соответственно, производного класса, но не наоборот.

```
Person* pp = &p;
Student* ss = &s;
pp = ss;
ss = pp; // ошибка компиляции
Person& rp = s;
Student& rs = p; // ошибка компиляции
```

Таким образом, если мы напишем `Person& rp = s;`, тогда `rp` будет давать доступ только к двум полям объекта `s`, унаследованным от `Person`. Именно поэтому в конструкторе копии класса `Student` не возникало проблем с вызовом конструктора копии `Person(s)` в списке инициализации.

Операция присваивания будет реализована несколько сложнее:

```
Student& operator=(const Student& s){
  if (&s != this) {
    delete[] marks;
    Person::operator=(s);
    univ = s.univ; m = s.m;
    marks = new int[m];
    for (int i = 0; i<m; ++i)
      marks[i] = s.marks[i];
  }
  return *this;
};
```

Операция присваивания не наследуется. Поэтому, чтобы выполнить присваивание для `private` полей, унаследованных от базового класса `Person`, необходимо выполнить операцию присваивания, определённую в классе `Person`.

```
Person::operator=(s);
```

Операция вывода в поток демонстрирует ещё одну особенность обращения к функции, определённой для предка.

```
friend ostream & operator<< (ostream & os, const Student & s){
  os<<(Person)s;
  os << s.univ << endl;
  for (int i = 0; i < s.m; ++i)
    os << s.marks[i] << " ";
  os << endl;
  return os;
}
```

Поскольку операция вывода в поток является внешней, то к ней невозможно применить разрешение области видимости. Поэтому используется явное приведение типа.

```
os<<(Person)s;
```

Обратное действие, когда происходит приведение базового типа к типу-наследнику, называется *downcast*. В этом случае необходимо использовать явное приведение типов с помощью шаблонной функции `static_cast<тип_наследника>`.

Добавим в класс `Student` функцию `get_univ()`, которая возвращает название учебного заведения, где учится студент. Такой функции нет, и не может быть в `Person`.

```
class Student: public Person {


public:

  string get_univ() const
  {
    return univ;
  }
};
```

Теперь рассмотрим ситуацию, когда нам может понадобиться приведение базового типа к типу наследника:

```
Person *p = new Student("Петров", 19, "ЮФУ", 3);
p->get_univ();    // ошибка компиляции
```

При вызове `p->get_univ()` произойдёт ошибка компиляции, так как в `Person` не определена функция `get_univ()`. Для корректного вызова указатель `p` нужно привести к типу `*Student`.

```
// Совеременный стиль:
static_cast<Student*>(pp)->get_univ();

// Старый стиль:
((Student*)pp)->get_univ;
```

Аналогичная ситуация возникает и при использовании ссылок:

```
Person & rp = *new Student("Петров", 20, "ЮФУ", 3);
static_cast<Student &>(rp).get_univ();
delete &rp;
```

📖 Преобразование downcast в C++ возможно только для указателей или ссылок на объекты.

Корректное преобразование downcast возможно только как обратное преобразование к upcast.

7.3. Отношение включения

Каждый ресурс, под который выделяется память в конструкторе, обычно стремятся обернуть объектом класса, контролирующим этот ресурс, что упрощает код.

В этом случае между классами возникает не отношение наследование, а отношение включения («has-a»). Оно означает, что один класс содержит в качестве члена объект другого/других классов. При этом он использует возможности включённых объектов. Можно сказать, что он реализован посредством классов включённых объектов.

Пример 85. Модифицировать класс `Student` из иерархии `Person-Student`, используя для хранения оценок разработанный ранее класс `Array` в примере 78.

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

class Student: public Person{
private:
  string univ;     // Университет
  Array marks;      // Оценки
public:
  Student() : marks(3), univ("МГУ") {   }

  Student(const string& name, int age, const string& u, int m):
    Person(name, age), univ(u), marks(m) { }

  void setMark(int i, int mark) {
    if (i < 0 || i >= marks.length())
    throw - 1;
    marks[i] = mark;
  }

  friend ostream & operator<< (ostream &os, const Student& s){
    os<<(Person)s;
    os << s.univ << endl;
    for (int i = 0; i < s.marks.length(); ++i)
      os << s.marks[i] << " ";
    os << endl;
    return os;
  }
};
```

Поскольку теперь для хранения оценок используется объект `marks` класса `Array`, он берёт на себя всю работу с динамической памятью. Благодаря этому код класса `Student` становится проще. Это отражено на UML диаграмме (рис.7.3).

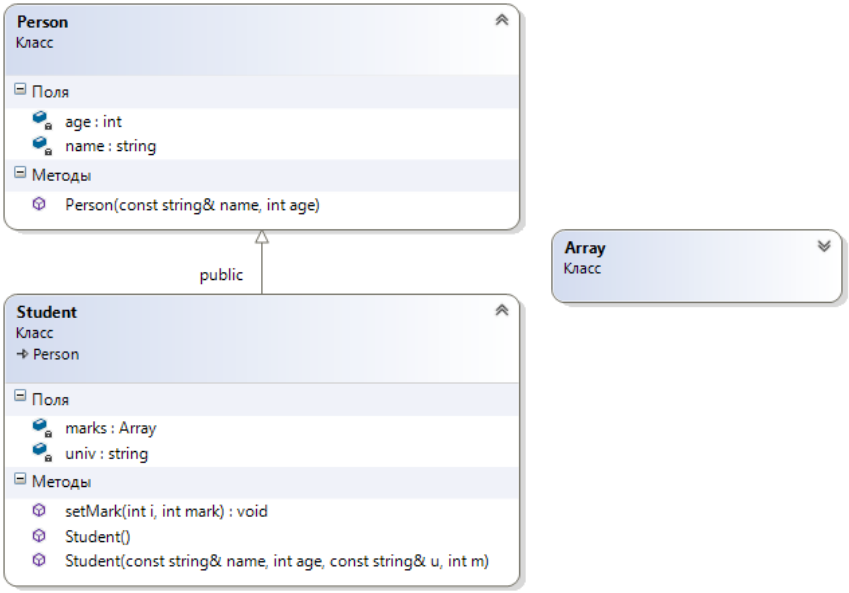

Рис.7.3. UML диаграмма к примеру 85

Теперь класс `Student` не нуждается в переопределении конструктора копии, деструктора и операции присваивания. Работают их версии по умолчанию, в которых происходит вызов соответствующих функций классов `Array` и `Person`.

Например, конструктор копии, создаваемый по умолчанию, выглядит следующим образом:

```
Student(const Student& s):
                    Person(s),univ(s.univ),marks(s.marks){ }
```

Поэтому создавать собственную реализацию нет никакого смысла. Она ничем не будет отличаться от версии по умолчанию.

☺ Старайтесь все динамически выделяемые ресурсы оборачивать в отдельные классы.

Если использовать класс `Array` из примера 78, то возникают ошибки компиляции в операции вывода в поток. Это связано с тем, что для константного объекта `s` класса `Student` используется неконстантная функция для операции индексации класса `Array`.

```
friend ostream & operator<< (ostream & os, const Student & s) {
  os<<(Person)s;
  os << s.univ << endl;
  for (int i = 0; i < s.marks.length(); ++i)
    os << s.marks[i] << " ";
  os << endl;
  return  os;
}
```

Данная проблема описывалась при обсуждении примера 78, для её решения предлагалось добавить реализацию константной операции индексирования. Следующий фрагмент кода демонстрирует описанные изменения.

```
class Array {
…
public:
…
  int& operator [] (int i);
  int operator [] (int i) const;
…
};

int& Array::operator [] (int i) {
  return A[i];
}

int Array::operator [] (int i) const {
  return A[i];
}
```

☺ Если в классе предполагается использование операции индексирования, то рекомендуется всегда реализовывать двумя способами.

В случае разработки сложных классов, использующих механизмы наследования и включения объектов других классов, необходимо хорошо понимать порядок вызова конструкторов и деструкторов.

Для конструкторов он выглядит следующим образом:

1. вызов конструктора базового класса;
2. вызов конструкторов для полей включённых объектов;
3. вызов конструктора основного объекта.

Деструкторы вызываются в обратном порядке:

1. вызов деструктора основного объекта;
2. вызов деструкторов полей;
3. вызов деструктора предка.

Порядок вызовов конструкторов для полей включённых объектов зависит от порядка их следования в объявлении класса и не зависит от порядка в списке инициализаторов.

7.4. Позднее связывание и виртуальные функции

Если базовый класс имеет несколько классов-наследников, то единственным способом объединить в одной коллекции объекты различных наследников базового класса является создание массива (или другого контейнера), содержащего указатели (или ссылки) на объекты базового класса.

Предположим, что у нас кроме класса `Student` есть ещё один наследник класса `Person` – класс `Professor`. Создадим массив указателей на объекты класса `Person`.

```
Person *p [5];
```

Инициализируем массив `p` указателями на объекты классов наследников.

```
p[0] = new Student("Петров", 19, "МГУ", 3);
p[1] = new Student("Кораблина", 18, "ЮФУ", 3);
p[2] = new Professor ("Тьюринг", 32, "ИВЭ");
p[3] = new Professor ("Страуструп", 32, "ПМП");
p[4] = new Student("Гончаров", 20, " ЮФУ ", 3);
```

Предположим, в классе `Person` есть функция `showInform`, которая переопределена в классах-наследниках `Student` и `Professor`. При этом в случае следующего кода:

```
for ( int i=0; i<5; ++i)
  p[i]->showInform();
```

мы увидим результат выполнения функции `showInform`, определённой в классе `Person`. Это обусловлено тем, что определение, какую из переопределённых функций вызвать, происходит в момент компиляции по типу переменной-объекта или переменной-ссылки (указателя).

Определение на этапе компиляции вызываемого варианта переопределённой функции называется *статическим,* или *ранним, связыванием*.

Однако в классах `Student` и `Professor` функция `showInform` переопределена и хотелось бы при вызове `p[0]->showInform()` увидеть информацию о студенте Петрове, а при вызове `p[2]->showInform()` — о профессоре Тьюринге.

Для того чтобы обеспечить возможность определения, какая из переопределённых функций должна быть вызвана, 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 переопределённой функции называется динамическим, или поздним, связыванием.

Позднее связывание реализует принцип *полиморфизма*. Полиморфизм позволяет выбирать вариант вызываемой функции в ходе выполнения программы.

📖 Позднее связывание реализуется с помощью *виртуальных функций*.

Для того чтобы сделать функцию виртуальной, нужно перед её объявлением в базовом классе указать спецификатор `virtual`.

Например, объявление класса `Person` должно быть изменено следующим образом:

```
class Person{
  public:
    //все, как было описано выше, кроме функции showInform
    ...
    virtual void showInform();
  private:
    //все, как было описано выше
};
```

Реализация функции `showInform` как для класса `Person`, так и для классов-наследников остаётся без изменений.

📖 Если объявление функции 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».

Поэтому добавлять спецификатор `virtual` в объявление функции `showInform` для классов-наследников не обязательно, но его использование улучшает читаемость программы.

☺ Рекомендуется всегда использовать спецификатор `virtual` в объявлении виртуальных функций, независимо от их расположения в иерархии наследования.

Пример 86. Создать базовый класс `shape`, для хранения параметров произвольной геометрической фигуры на плоскости и вычисления её площади по этим параметрам. Создать два класса-потомка `rectangle` (параметры: две стороны) и `triangle` (параметры: сторона и высота к

ней), представляющие, соответственно, прямоугольник и треугольник. Для демонстрации работы создать два полиморфных контейнера. Один – на основе статического массива указателей на базовый класс, а второй – на основе динамического массива.

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

class shape {
  protected:
    double x,y;
  public:
    //конструкторы не создаём, так как
    //используется автоматический конструктор по умолчанию
    void set_param(double x0, double y0) {
      x=x0;
      y=y0;
    }
    virtual void show_area(){
      cout<<"Area calculation for this class is undefined";
      cout <<endl;
    }
};

class triangle: public shape {
  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 shape {
  public:
    //используется автоматический конструктор по умолчанию
    //вызывающий конструктор по умолчанию базового класса
    void show_area() {
      cout<<endl<<"rectangle with sides "<<x<<" * "<<y;
      cout<<endl<<"has an area "<<x*y<<endl;
    }
};

void tellMeAboutYourself(shape *s) {
  s-> show_area();
}
```

```
int main() {
  shape *p[2]; // полиморфный массив в статической памяти
  triangle t;
  rectangle r;

  p[0]=&t;
  p[0]->set_param(10.0,5.0);
  p[0]->show_area();

  p[1]=&r;
  p[1]->set_ param (10.0,5.0);
  p[1]->show_area();

  for (auto x: p)
    tellMeAboutYourself(x);

  // полиморфный массив в динамической памяти
  cout << "input count of shapes->";
  int n;
  cin >> n;
  shape **pp = new shape *[n];
  int type_shape;
  double a, b;
  for (int i = 0; i < n; ++i) {
    cout << "input shape type[1-rectangle, 2-triangle]"<<endl;
    cin >> type_shape;

    if (type_shape == 1) {
      cout << "input rectangle sides->" << endl;
      cin >> a >> b;
      pp[i] = new rectangle();
      pp[i]->set_param(a, b);
    }
    else {
      cout << "input triangle base ->";
      cin >> a;
      cout << "input triangle height ->";
      cin >> b;
      pp[i] = new triangle();
      pp[i]->set_param(a, b);
    }
    pp[i]->show_area();
  }
  for (int i = 0; i < n; ++i)
    tellMeAboutYourself(pp[i]);
  for (int i = 0; i < n; ++i)
    delete pp[i];
  delete[] pp;
  return 0;
}
```

В этом примере наследование не имеет целью расширение базового класса (рис. 7.4). Каждый из классов-наследников реализует своё собственное поведение на основе виртуальной функции, объявленной в базовом классе.

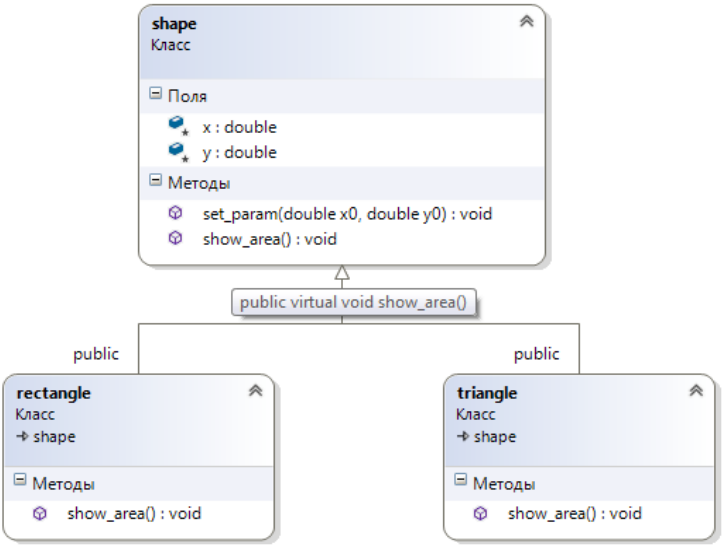

Рис.7.4. UML диаграмма классов примера 86

Функция `tellMeAboutYourself` ожидает от любого наследника класса `shape` собственную реализацию функции `show_area`. Принято говорить, что между функцией `tellMeAboutYourself` и наследниками класса `shape` устанавливается соглашение по возможностям взаимодействия. Такое соглашение называется *интерфейсом*.

В C++11 добавили новый тип цикла — *foreach*, который предоставляет более простой и безопасный способ итерации по статическому массиву или любой другой структуре типа списка. Синтаксис цикла foreach для случая статического массива:

```
for (переменная: массив)
  statement;
```

Выполняется итерация по каждому элементу массива, присваивая значение текущего элемента массива переменной. В целях лучшей производительности объявляемый элемент должен быть того же типа, что и элементы массива, иначе произойдёт неявное преобразование типа.

Данный цикл имеет второе название — цикл `for` *по диапазону*. Используется как более читаемый эквивалент традиционного цикла `for`, работающего с диапазоном значений, например со всеми элементами в массиве или контейнере. Поскольку цикл работает по диапазону, то для набора перебираемых элементов должны быть определены начало и конец диапазона. В связи с этим для массивов в динамической памяти цикл по диапазону использовать нельзя. Именно поэтому в примере 86 для массива `shape **pp` не используется цикл `for` по диапазону

```
  shape **pp = new shape *[n];
…
  for (int i = 0; i < n; ++i)
    tellMeAboutYourself(pp[i]);
```

Чтобы не задумываться о типах, в заголовке цикла `for(auto x: p)` используется автоматическое определение типа.

☺ Для пользовательских типов данных, используемых в качестве параметра цикла, рекомендуется применять `auto`.

До C++ 11 ключевое слово `auto` использовалось для явного указания, что переменная имеет автоматический класс памяти (время существования ограничено блоком, в котором она определена). Однако, поскольку все переменные в современном C++ по умолчанию имеют автоматический класс памяти, ключевое слово `auto` стало излишним и, следовательно, устаревшим.

Начиная с C++ 11, ключевое слово `auto` при инициализации переменной может использоваться вместо типа переменной, чтобы

сообщить компилятору, что он должен определить тип переменной исходя из инициализируемого значения. Это называется *выводом типа* (или *автоматическим определением типов*).

7.5. Абстрактные классы

Во многих случаях вообще нет смысла давать определение виртуальной функции в базовом классе. Например, в классе `shape` определение функции `show_area()` — лишь способ корректно сообщить об ошибке использования. Объект класса `shape` без конкретизации типа фигуры не может иметь площади. Для того чтобы не прибегать к такому искусственному приёму, 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 полиморфизма.

Если в базовом классе имеется чисто виртуальная функция, производный класс должен иметь определение её собственной реализации. Если реализация хотя бы одной из чисто виртуальных функций не будет выполнена, производный класс, в свою очередь, останется абстрактным.

Абстрактные классы посредством чисто виртуальных функций описывают интерфейс, который можно использовать в других классах или функциях, ожидая, что наследники реализуют эти функции.

Рассмотрим ещё одну иерархию наследования для геометрических фигур на плоскости, на базе абстрактного класса `shape`.

Пример 87. Создать абстрактный класс `shapeC` и его наследников: классы `circle` и `filled_circle`.

```
#include <iostream>
#include <cmath>
using namespace std;
class shapeC {
  public:
    //используется автоматический конструктор по умолчанию
    virtual ~shapeC() {}
    virtual double square() const =0;
    virtual shapeC* clone() const =0;
    virtual void debug(ostream &out) const=0;
};
class circle: public shapeC {
  public:
    circle(double r=0): radius(r) { }
    ~circle() {}
    double square() const{
      return 3.14*radius*radius;
    }
    circle* clone() const {
      return new circle(radius);
    }
    void debug (ostream & out) const {
      out<<" radius = "<<radius <<endl;
    }
  protected:
    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(radius, color);
    }
    void debug (ostream & out) const {
      circle::debug(out);
```

```
      out<<" color = "<<color <<endl;
    }
  private:
    int color;
};

int main() {
shapeC *p[4];
  circle t(5);
  filled_circle r(10, 255);
  p[0] = &t; p[1] = &r;
  p[2] = p[0]->clone(); p[3] = p[1]->clone();
  for (auto x : p) {x->debug(cout);}
  return 0;
}
```

Соответствующая UML диаграмма представлена на рисунке 7.5.

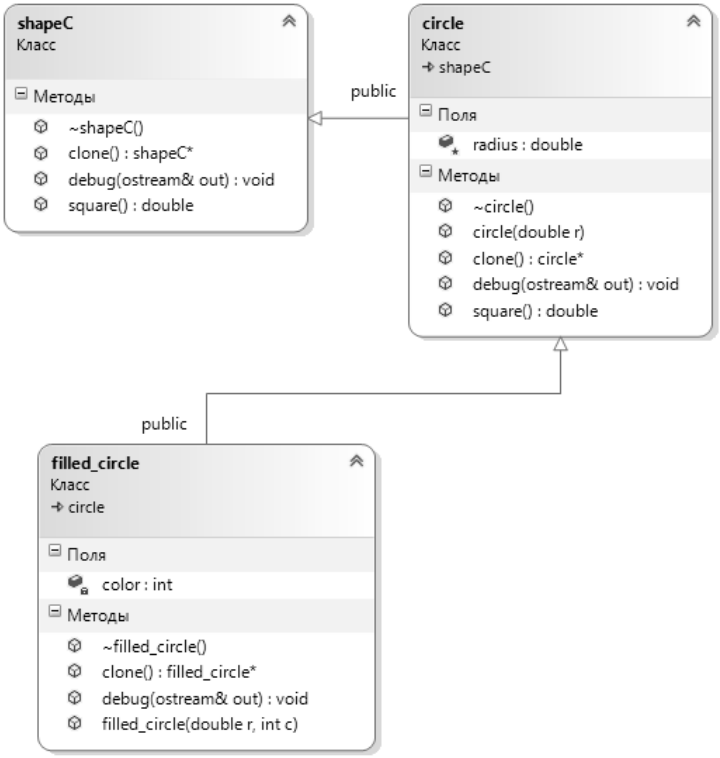

Рис. 7.5. UML диаграмма классов для примера 87

Массив p указателей на shapeC, представляет собой полиморфный контейнер, поскольку он может содержать указатели на circle и filled_circle.

Имея объекты наследников класса shapeC, можно их адреса присвоить элементам массива p.

```
circle t(5);
filled_circle r(10, 255)
p[0] = &t;
p[1] = &r;
```

Но если потребуется создать копии этих двух элементов, операция присваивания не поможет, поскольку будет выполнено присваивание адресов.

```
p[2] = p[0];
p[3] = p[1];
```

В этом случае p[2] и p[0] будут ссылаться на один и тот же объект t, а p[3] и p[1] — на объект r.

Решением данной проблемы могло бы быть создание виртуального конструктора, однако конструкторы в C++ не могут быть виртуальными. Поэтому для решения этой проблемы следует использовать функцию clone().

```
virtual shapeC* clone() const =0;
class circle: public shapeC {
  public:
...
  circle* clone() const {
    return new circle(radius);
...
  }
}
class filled_circle: public shapeC {
  public:
...
  filled_circle* clone() const {
    return new filled_circle(radius, color);
  }
...
}
```

Эта функция является виртуальной и для каждого наследника создаёт и возвращает объект соответствующего класса. Клонирование является полиморфным, так как объект должен клонировать себя, а не объект базового типа. То есть `circle` клонирует `circle`, `filled_circle` клонирует `filled_circle`.

```
p[2] = p[0]->clone();
p[3] = p[1]->clone();
```

Следует обратить внимание на то, что деструктор базового класса тоже объявлен виртуальным.

```
virtual ~shapeC() {}
```

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.

Статические функции также не могут быть виртуальными или объявляться с модификаторами `const` иди `volatile`.

7.6. Цена виртуальности и система RTTI

Полиморфизм в C++ реализуется с помощью таблиц виртуальных функций — *Virtual Methods Table (VMT)*.

Для каждого класса, содержащего хотя бы одну виртуальную функцию, создаётся VMT, которая содержит адреса всех виртуальных функций, как этого класса, так и всех его предков. В каждом объекте такого класса появляется дополнительный указатель `vptr` на таблицу виртуальных функций. Если в классе и его предках нет виртуальных функций, то в его объекте поле `vptr` отсутствует (реализуется принцип: не платим за то, что не используем).

> ➢ В C++ нет общего главного класса-предка, как например, `Object` в PascalABC.Net, C#, Java**.** Это делается в целях повышения эффективности. Так как общий предок предоставляет набор виртуальных функций (методов), что приводит к созданию VMT для каждого класса.

Пусть в классе `filled_circle` есть функция `set_color()`.

```
void set_color(int c) {color = c;}
```

Даже, если мы точно знаем, что в элементе массива `p[1]` находится указатель на объект класса `filled_circle`, вызвать функцию `set_color()` через этот указатель нельзя.

```
p[1]->set_color(10); // ошибка компиляции !!!
```

Нужно выполнить явное приведение типа с помощью операции `dynamic_cast <тип>` для преобразования указателя на `shapeC` к указателю на `filled_circle`. Оператор `dynamic_cast` может быть применён к указателям или ссылкам.

```
dynamic_cast<filled_circle *>(p[1])->set_color(10);
```

➢ В отличие от обычного приведения типа в стиле Си, проверка корректности приведения типов в стиле C++ для классов с виртуальными функциями производится во время выполнения программы.

Если в классе имеются виртуальные методы, то на этапе выполнения операция `dynamic_cast` пытается выполнить преобразование к указанному типу данных. В случае преобразования указателя к типу данных, который не является фактическим типом объекта, в результате будет получен нулевой указатель. При работе со ссылками, если преобразование невозможно, будет сгенерировано исключение `std::bad_cast`. Для его обработки операцию `dynamic_cast` следует поместить в блок `try/catch`.

Если виртуальных методов в классе и его предках нет, то `dynamic_cast` будет работать как `static_cast`.

Операция `dynamic_cast` использует *механизм динамической идентификации типа данных RTTI* (*Runtime Type Identification*). RTTI основана на способности системы сообщать о динамическом типе объекта и предоставлять информацию об этом типе во время выполнения (в отличие от времени компиляции). RTTI доступен только для классов, которые являются полиморфными, т.е. у них есть хотя бы одна виртуальная функция.

Таким образом, операция `dynamic_cast` позволяет идентифицировать динамический тип переменной во время выполнения. Операция `typeid()` используется для определения типа переменной во время выполнения. Она возвращает ссылку на объект класса `std::type_info`, который содержит поля, позволяющие получить информацию о типе. Этот класс содержит перегруженные операции `==` и `!=`, а также функцию `name()`.

Следующие проверки идентичны по результатам.

Первый вариант

```
filled_circle *q = dynamic_cast<filled_circle *>(p[1]);
if (q!=nullptr)
  q ->set_color(10);
```

Второй вариант

```
#include <typeinfo> //требуется для использования typeid()
...
if (typeid(*p[1]).name() == typeid(filled_circle).name())
  dynamic_cast<filled_circle *>(p[1])->set_color(10);
```

Операция `typeid()` для полиморфных типов работает полиморфным образом — возвращает динамический тип объекта.

Обратите внимание, что результатом вызова функции `typeid(*p[1]).name()` является строка «class filled_circle», а результатом вызова `typeid(p[1]).name()` — «class shapeC *».

7.7. Отношение подобия

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.

Напомним, что C++ рассматривает открытое наследование как отношение типа «является». В частности, компиляторы, столкнувшись с иерархией открытого наследования, неявно преобразуют указатель или ссылку на объект класса потомка в указатель или ссылку на объект предка, если это необходимо для вызова функций. Например, в рассмотренных примерах класс `Student` открыто наследует классу `Person`. При этом в случае необходимости компилятор неявно преобразует указатель или ссылку на объект класса `Student` в указатель или ссылку на объект класса `Person`.

В противоположность открытому наследованию компиляторы в случае закрытого наследования не преобразуют указатели или ссылки на объекты производного класса в указатели или ссылки на объекты базового класса. Кроме того, члены, наследуемые от закрытого базового класса, становятся закрытыми, даже если в базовом классе они были объявлены как защищённые или открытые. Поэтому для объектов наследника мы не можем вызывать функции предка.

Закрытое наследование означает «реализовано посредством…». Делая класс `Derived` закрытым наследником класса `Base`, мы заинтересованы в использовании уже написанного для `Base` кода.


Можно сказать, что закрытое наследование означает наследование одной только реализации, без интерфейса. Закрытое наследование ничего не означает в ходе проектирования программного обеспечения и обретает смысл только на этапе реализации.

Таким образом, закрытое наследование — это исключительно приём реализации, который является альтернативой включению (композиции). Например, класс `Derived` может не наследовать, а содержать объект класса `Base`.

Если класс `Base` имеет защищённые (`protected`) члены, то в случае включения в класс `Derived` они недоступны для использования во включающем классе. Если же эти члены необходимы классу `Derived`, то следует применять закрытое наследование.

Пример 88. Предположим, что имеется реализация класса `Point` «точка на прямой». Использовать этот класс при реализации класса `GreenHopper` «исполнитель Кузнечик из задач по информатике». С его помощью решить следующую задачу. Кузнечик может перемещаться по числовой оси с помощью команды `jump(N)`, где `N` — любое целое число.

Кроме того, он может сообщать о своём положении на числовой оси с помощью команды `WhereAreYou()`. Кузнечик выполнил программу из 50 пар команд: `jump(5)`, `jump(-3)`. На какую одну команду можно заменить эту программу, чтобы Кузнечик оказался в той же точке, что и после выполнения программы.

```
class Point {
private:
  int x;
public:
  Point(int x = 0) : x(x)
  {}
protected:
  void setx(int newx) {
    x = newx;
  }
  int getx() {
    return x;
  }
};
```

По условию задачи в основу решения должен быть положен предложенный класс `Point`. Использование открытого наследования невозможно вследствие требования, что кузнечик должен понимать только две команды `jump(N)` и `WhereAreYou()`. Использование включения не позволяет обращаться к функциям `setx(N)` и `getx()`, поскольку они объявлены как `protected`. Единственным вариантом решения является закрытое наследование (рис.7.6).

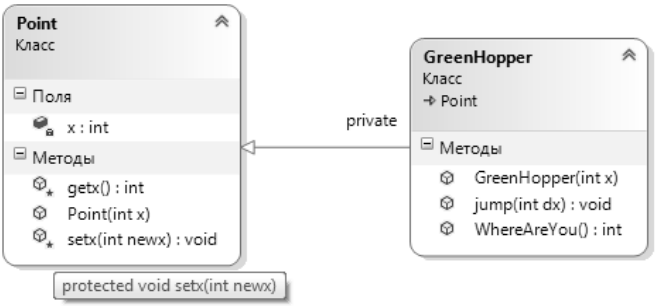

Рис. 7.6. UML диаграмма классов для примера 88

```
#include<iostream>

using namespace std;

class GreenHopper : private Point {
public:
  GreenHopper(int x=0) :Point(x) {}
  void jump(int dx) {
    setx(getx()+dx);
  }
  int WhereAreYou() {
    return getx();
  }
};

int main() {
  GreenHopper G;
  //cout<<G.getx();// ошибка доступа
  int t = G.WhereAreYou();
  cout <<  t << endl;
  for (int i = 0; i < 50; ++i) {
    G.jump(5);
    G.jump(-3);
  }
  cout << "jump(" << G.WhereAreYou() - t << ")"<<endl;
  return 0;
}
```

Пример 89. Используя готовую реализацию класса «список целых чисел», создать класс «стек целых чисел».

```
class error{
};

class list{
private:
  // внутренняя организация класса может быть разной
  public:
  list();
  ~list();
  bool isEmpty() const;
  void inHead(int val);
  void inTail(int val);
  int getFirst()const throw (error);
  int getLast()const throw (error);
  void delFirst()throw (error);
  void delLast()throw (error);
  //в полной реализации могут быть ещё функции
};
```

Теперь реализуем класс «стек» (рис. 7.7).

```
class Stack:private list{
  public:
    Stack():list(){}
    bool isEmpty(){
      return list::isEmpty();
  }
  void push(int i){
    inHead(i);
  }
  int top() const throw (error);{
    return getFirst();
  }
  int pop throw (error); (){
    int res=getFirst();
    delFirst();
    return res;
  }
};
```

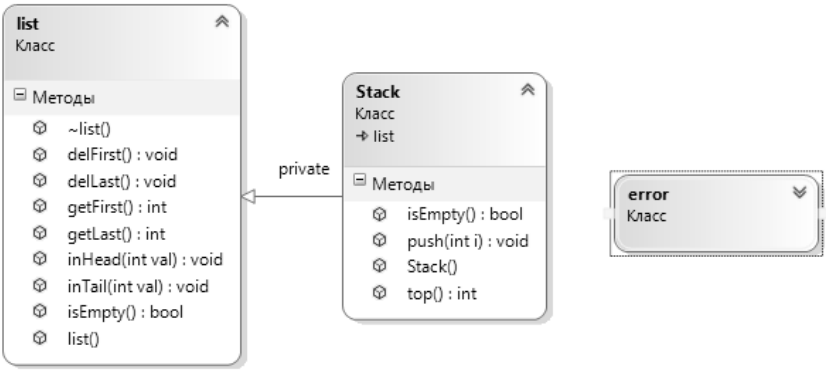

Рис. 7.7. UML диаграмма классов для примера 89

Использование закрытого наследования позволило в этом примере скрыть возможности базового класса и тем самым защитить стек от некорректного использования. Например, нельзя попытаться вынуть элемент «снизу» из стека.

Приведём 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;
}
```

 Для проверки работоспособности основной программы необходимо иметь реализацию класса «список целых чисел». Создать реализацию класса на базе массива.

 Реализовать класс стек, используя принцип включения, а не закрытого наследования.

7.8. Коллекции и итераторы

Коллекция или *контейнер* — объект программы, содержащий в себе набор значений одного или различных типов, и позволяющий обращаться к этим значениям.

 Термин «объект» в данной книге используется в двух смыслах: программный объект и экземпляр класса. С точки зрения коллекции — это программный объект.

Назначение коллекции — служить хранилищем программных 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 функции, в зависимости от её логической организации. Реализация может допускать выполнение отдельных операций над коллекциями в целом. Наличие операций над коллекциями во многих случаях может существенно упростить программирование.

Примерами коллекций являются классы для реализации массива, строки, списка.

Итератор — объект, позволяющий программисту перебирать все элементы коллекции без учёта особенностей её реализации.

В простейшем случае итератором в низкоуровневых языках является указатель. Операцию индексирования можно считать примитивной формой итератора. Необходимо отметить, что счётчик цикла иногда называют итератором цикла. Тем не менее, счётчик цикла обеспечивает только перебор элементов, но не доступ к элементу.

Использование итераторов в обобщённом 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 определён 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 функции 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 придётся использовать. Кроме того, такой встроенный итератор обычно только один, а могут возникнуть задачи, в которых потребуется наличие двух, трёх 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 даёт возможность перемещаться по объектам, содержащимся в коллекции, аналогично тому, как текущий указатель позволяет перемещаться внутри коллекции.

Как правило, итератор имеет операцию доступа к элементу коллекции по ссылке. Такая операция может быть реализована путём перегрузки операции разыменования `*`. Для итераторов предусматриваются также операции перемещения вперёд и назад по коллекции объектов. Чаще всего для этого перегружаются операции `++` и `--`. Кроме того, операции `==` и `!=` обычно перегружаются для проверки равенства итераторов. 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.

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 90. Реализовать линейный односвязный список и итератор для него с возможностью перемещения вперёд по списку.

В этом примере рассматривается простейшая организация линейного списка с минимальными требованиями к итератору, что позволяет существенно упростить реализацию итератора.

Класс список `List` содержит элементы, описываемые структурой `node`:

```
struct node{
  int data;
  node* next;
  node(int data, node* next) {
    this->data = data;
    this->next = next;
  }
};
```

Для упрощения в дальнейшем вывода элементов списка перегружена операция вывода в поток. Для структуры не требуется объявлять её дружественной.

```
ostream & operator<<(ostream & out, const node& X) {
  out << X.data;
  return out;
}
```

Чтобы в определении класса `List` операции `begin()` и `end()` могли возвращать объекты типа итератор списка, нужно сделать предварительное объявление класса итератора.

```
class listIterator;
```

Это связано с рекурсией в определении классов коллекции `List` и итератора `listIterator`.

Определение класса `List`:

```
class List{
public:

  //вложенный класс
  class Error {
  public:
    void what(){
      cout << "List is empty" << endl;
    }
  };

  List() {
    head = 0; tail = 0;
  }
  List(const List& l);
  ~List();
  bool isEmpty() const;
  void inHead(int val);
  void inTail(int val);
  int getFirst()const;
  int getLast()const;
  void delFirst();
  void delLast();
  listIterator begin() const;
  listIterator end() const;
  friend ostream& operator<<(ostream & os, const List& l);
  //в полной реализации могут быть еще методы
private:
  node* head;
  node* tail;
  Error err;
};
```

Определение класса итератора для списка:

```
class listIterator {
public:
  class Error {
  public:
    void what(){
      cout << "Iterator error" << endl;
    }
  };
private:
  node *cur;
```

```
public:
  listIterator(node *e) : cur(e){}
  const int operator *(){
    if (!cur)
      throw Error();
    return cur->data;
  }
  listIterator operator++(); //префиксный ++
  int operator == (const listIterator &ri) const;
  int operator != (const listIterator &ri) const;
};
```

Реализация класса `list` для односвязного списка:

```
List::List(const List& l){
  head = nullptr; tail = nullptr;
  node* q = l.head;
  while (q) {
    inTail(q->data);
    q = q->next;
  }
}
List::~List(){
  while (head){
    node* cur = head;
    head = head->next;
    delete cur;
  }
  tail = head = nullptr;
}
bool List::isEmpty() const {
  return (head == nullptr);
}

void List::inHead(int val){
  node* t = new node(val,head);
  if (!head)
    tail = t;
  head = t;
}
void List::inTail(int val){
  node* t = new node(val, nullptr);
  if (head){
    tail->next = t;
    tail = t;
  }
  else{
    head = tail = t;
  }
}
```

```
int List::getFirst() const{
  if (head)
    return head->data;
  else
    throw err;
}

int List::getLast()const{
  if (head)
    return tail->data;
  else
    throw err;
}

void List::delFirst(){
  if (head){
    node *t = head;
    head = head->next;
    delete t;
  }
  else
    throw err;
}

void List::delLast(){
  if (head){
    if (head == tail) { delete tail; head = nullptr; }
    else{
      node *t = head;
      while (t->next != tail) t = t->next;
      delete tail;
      tail = t;
      t->next = nullptr;
    }
  }
  else
    throw err;
}

ostream& operator<<(ostream & os, const List& l){
  node *p = l.head;
  while (p) {
    os << *p << " ";
    p = p->next;
  }
  os << endl;
  return os;
}
```

Реализация класса `listIterator` для итератора односвязного списка:

```
listIterator listIterator:: operator++() {
  if (cur) {
    cur = cur->next;
    return *this;
  }
  else throw Error();
}
int listIterator:: operator==(const listIterator &ri) const {
  return (cur == ri.cur);
}

int listIterator:: operator!=(const listIterator &ri) const {
  return !(*this == ri);
}
```

В закрытых полях класса хранится указатель на текущее положение итератора в коллекции. Конструктор с параметрами позволяет инициализировать поля.

Структуры являются открытыми классами с открытым доступом, поэтому возможно обращение к полям структуры, представляющей элемент списка напрямую.

Чтобы использовать итератор для работы с классом-коллекцией, необходимо, чтобы класс содержал функции инициализации итератора:

```
listIterator begin() const;
listIterator end() const;
```

Инициализация итератора ссылкой на первый элемент списка осуществляется следующей функцией:

```
listIterator list::begin() const {
  return listIterator (head);
}
```

В качестве параметров конструктору итератора передаются указатель на объект-список, для которого будет создан итератор, и указатель на голову списка.

Функция `listIterator list::end() const` возвращает значение, которое является признаком того, что итератор достиг конца списка:

```
listIterator list::end() const{
  listIterator iter(nullptr);
  return iter;
}
```

Для поддержания общего стиля написания итераторов функцию `end()` нельзя заменить простым сравнением указателя с `nullptr`. Для других реализаций списков признак конца списка может отличаться от `nullptr`.

Рассмотрим использование итератора для линейного односвязного списка на примере нескольких функций.

Нахождение суммы элементов списка, расположенных между двумя итераторами.

```
int sum(listIterator b, listIterator e) {
  int sum = 0;
  while (b != e){
    sum+= *b; ++b;
  }
  return sum;
}
```

Нахождение итератора, ссылающегося на максимальный элемент списка.

```
listIterator max(listIterator b, listIterator e) {
  listIterator maxx = b;
  while (b != e){
    if ( *b > *maxx ) maxx = b;
    ++b;
  }
  return maxx;
}
```

Ниже приведён код для тестирования созданных функций.

```
int main(){
  List l1;
  for (int i = 0; i < 5;++i)
    l1.inHead(i);
  for (int i = 5; i < 10; ++i)
    l1.inTail(i);
```

```
  List l2(l1);
  cout << " list \n" << l2 << endl;
  cout << sum(l2.begin(), l2.end())<<endl;
  listIterator mt = max(l2.begin(), l2.end());
  cout << *mt << endl;
  cout << sum(l2.begin(), mt) << endl;
  cout << sum(mt, l2.end()) << endl;
  return 0;
}
```

Пример 91. Дополнить реализацию класса односвязного списка методом вставки нового элемента перед элементом, на который указывает итератор.

Обычно методы вставки должны возвращать указатель или итератор на вставленный элемент.

```
listIterator insert(listIterator it, int val);
```

Поскольку внутри функции возникает необходимость обращения к полю `cur` итератора, то класс `List` делаем дружественным классу итератора списка.

```
class listIterator {
...
friend class List;
};
```

Чтобы выполнить вставку перед элементом нужно найти предыдущий элемент. В силу семантики односвязного списка поиск предыдущего элемента реализуется через цикл.

```
listIterator List::insert(listIterator it, int val) {
  //пустой список или итератор указывает на первый
  if (it.cur == head) {
    inHead(val);
    return begin(); //возвращаем итератор на начало списка
  }
  node* tmp = new node(val, it.cur);
  node* p = head;
  while (p->next != it.cur) {
    p = p->next;
  }
  p->next = tmp;
  //возвращаем итератор на вставленный элемент
  return listIterator(tmp);
}
```

Продемонстрируем работоспособность метода `insert()`. Результаты представлены на рисунке 7.8.

```
int main() {
  List l1;
  auto ps=l1.insert(l1.begin(), -77);
  for (int i = 0; i < 5; ++i)
    l1.inHead(i);
  for (int i = 5; i < 10; ++i)
    l1.inTail(i);
  auto ps1 = l1.insert(l1.begin(), -222);
  cout << " list \n" << l1 << endl;
  List l2(l1);
  listIterator mt = max(l2.begin(), l2.end());
  auto it = l2.insert(mt,777);
  cout << " list \n" << l2 << endl;
  return 0;
}
```

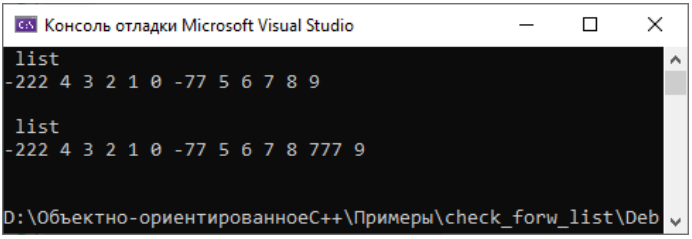

Рис. 7.8. Окно вывода

К сожалению, данная реализация метода `insert()` не проверяет принадлежность итератора, передаваемого в качестве параметра, коллекции, для которой вызывается метод. Поэтому могут возникать трудноуловимые ошибки времени выполнения.

Например, следующий код приводит к такой ошибке.

```
l1.insert(l2.begin(), 0);
```

Чтобы гарантированно избегать таких ситуаций необходимо проверять соответствие итератора и коллекции. Один из вариантов решения — хранить в итераторе указатель на итерируемую коллекцию.

В примере 91 изменения коснутся класса `listIterator`, методов `begin()`, `end()` и `insert()` в классе `List`.

В классе итератора добавляем поле указатель на коллекцию и меняем конструктор.

```
class listIterator {
private:
  const List* collection;
  node* cur;
public:
  listIterator(const List* s,node* e):collection(s),cur(e) {}
…
};
```

В методах `begin()`, `end()` используем новый вариант конструктора итератора.

```
listIterator List::begin() const {
    return listIterator(this,this-> head);
}
listIterator List::end() const {
    listIterator iter(this,Bnullptr);
    return iter;
}
```

Добавляем проверку на идентичность коллекций в методе `insert()`.

```
listIterator List::insert(listIterator it, int val) {
  if (it.collection != this) {
  // чтобы запретить операцию для итератора не по этому списку
  // it.collection доступно, т.к. List друг listIterator
    throw listIterator::Error();
  }
  if (it.cur == head) {
    val++;
    inHead(val);
    return begin();
  }
  node* tmp = new node(val, it.cur);
  node* p = head;
  while (p->next != it.cur) {
    p = p->next;
  }
  p->next = tmp;
  return listIterator(it.collection, tmp);
}
```

7.9. Реализация итератора для двусвязного списка

Реализация простейшего итератора для линейного односвязного списка является простой и лёгкой для понимания. С неё удобно начинать.

Но она не позволяет делать более сложные вещи, которые могут быть востребованными.

Пример 92. Реализовать линейный двусвязный список и итератор для него с возможностью перемещения в двух направлениях.

Структура узла двусвязного списка будет содержать дополнительное поле — указатель на предыдущий элемент.

```
struct node {
    int data;
    node* next;
    node* prev;
    node(int data, node* next, node* prev) {
        this->data = data;
        this->next = next;
        this->prev = prev;
    }
};

ostream& operator<<(ostream& out, const node& X) {
    out << X.data;
    return out;
}
```

Если попытаться расширить реализацию из примера 90 на случай двусвязного списка, то мы столкнёмся с неразрешимой проблемой. В этом случае операция `operator--()`, реализуемая по аналогии с операцией `operator++()`, неприменима к итератору в позиции `end()`. Для корректной реализации потребуется пересмотреть организацию списка и откорректировать реализацию итератора.

Для этого требуется предусмотреть фиктивный головной элемент (dummy head node). Он всегда существует, даже если список пуст. В этом случае первый элемент в действительности является вторым.

Фиктивный (заглавный) элемент в случае пустого списка представлен на рисунке 7.9.

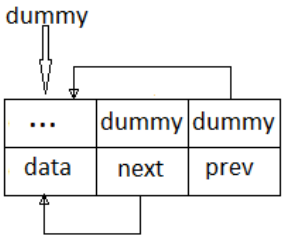

Рис. 7.9. Пустой список с фиктивным элементом

В таком списке поля указателей не могут принимать значения `nullptr`. Ссылка `prev` первого элемента списка указывает на заглавный узел `dummy`, а ссылка `next` последнего элемента тоже ссылается на заглавный узел `dummy`. Организация непустого списка представлена на рисунке 7.10.

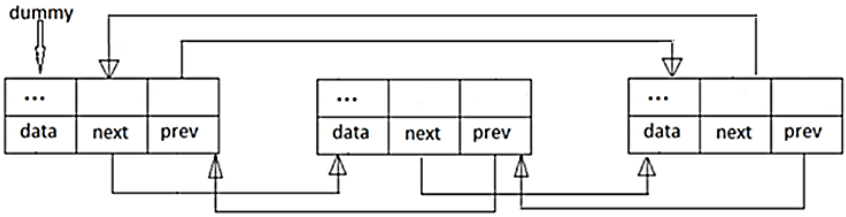

Рис. 7.10. Непустой двусвязный список с фиктивным элементом в начале

В таком случае признаком последнего элемента является тот факт, что `next` элемента равен `dummy`, а `dummy->prev` указывает на последний элемент списка. Следовательно мы можем корректно определить операцию `operator--()`.

Внесём необходимые изменения в организацию списка `List`, связанные с использованием фиктивного головного узла.

```
class listIterator;
class List {
public:
  class Error {
    public:
      void what() {cout << "List is empty" << endl;}
    };
    List();
```

```
    List(const List& l);
    ~List();
    bool isEmpty() const;
    void inHead(int val);
    void inTail(int val);
    int getFirst()const;
    int getLast()const;
    void delFirst();
    void delLast();
    listIterator begin() const;
    listIterator end() const;
    friend ostream& operator<<(ostream& os, const List& l);
    //в полной реализации могут быть еще методы
private:
    node *dummy;
    Error err;
    friend class listIterator;
};
```

Первое существенное изменение относится к конструктору без параметров. В конструкторе создаётся фиктивный головной узел `dummy`, значения указателей `next` и `prev` в котором заполнены так, как отражено на рисунке 7.9.

```
List::List() {
    dummy = new node(0, nullptr, nullptr);
    dummy->next = dummy;
    dummy->prev = dummy;
}
```

Проверка на пустоту предполагает не сравнение с `nullptr`, а проверку того, что головной узел ссылается сам на себя.

```
bool List::isEmpty() const {
    return (dummy->next == dummy);
}
```

Учтём появление заглавного узла в конструкторе копии и в деструкторе.

```
List::List(const List& l): List() {
    if (l.dummy->next!=l.dummy) {
        node* q = l.dummy->next;
        while (q != l.dummy){
            inTail(q->data);
            q = q->next;
        }
  }
}
```

```
List::~List() {
    if (dummy->next!=dummy) {
        node* t = dummy->next;
        while (t!=dummy) {
            node* cur = t;
            t = t->next;
            delete cur;
        }
    }
    delete dummy;
}
```

Алгоритмы добавления/удаления в позициях начала и конца списка становятся сложнее не только из-за наличия второго указателя, но и из-за необходимости учёта заглавного узла.

```
void List::inHead(int val) {
    node* t = new node(val, dummy->next, dummy);
    if (dummy->next==dummy) {
        dummy->prev = t;
    }
    else {
        t->next->prev = t;
    }
    dummy->next = t;
}

void List::inTail(int val) {
    if (dummy->next == dummy)
        inHead(val);
    else {
        node* t = new node(val, dummy, dummy->prev);
        t->prev->next = t;
        dummy -> prev = t;
    }
}
void List::delFirst() {
    if (dummy->next!=dummy) {
        node* t = dummy->next;
        if (t->next != dummy) {
            dummy->next = t->next;
            dummy->next->prev = dummy;
        }
        else {
            dummy->next = dummy;
            dummy->prev = dummy;
        }
        delete t;
    }
```

```
    else
        throw err;
}
void List::delLast() {
    if (dummy->next!=dummy) {
        node* t = dummy->prev;
        if (t->prev!=dummy) {
            dummy->prev = t->prev;
            t->prev->next = dummy;
        }
        else{
         dummy->next = dummy;
         dummy->prev = dummy;
        }
    delete t;
    }
    else
        throw err;
}
```

Алгоритмы доступа к значениям первого и последнего элементов, а также перегрузка вывода списка в поток требуют незначительных изменений в проверке на пустоту списка и в получении значения.

```
int List::getFirst() const {
    if (dummy->next!=dummy)
        return dummy->next->data;
    else
        throw err;
}

int List::getLast()const {
    if (dummy->next!=dummy)
        return dummy->prev->data;
    else
        throw err;
}

ostream& operator<<(ostream& os, const List& l) {
    if (l.dummy->next!=l.dummy) {
        node* p = l.dummy->next;
        while (p != l.dummy) { os << *p << " "; p = p->next; }
        os << endl;
    }
    return os;
}
```

Реализация списка с использованием заглавного элемента не должна нарушать соглашение: для пустого списка `begin() == end()`. Именно

поэтому next и prev заглавного элемента равны dummy. Методы begin() и end() возвращают итераторы, хранящие в случае пустого списка позиции dummy->next, равное dummy, и dummy соответственно.

```
listIterator List::begin() const {
    return listIterator(this, dummy->next);
}
listIterator List::end() const {
    listIterator iter(this, dummy);
    return iter;
}
```

В классе итератора не меняются поля класса и его конструктор. В остальные методы внесены изменения.

```
// iterator
class listIterator {
public:
    class Error {
    public:
        void what() {
            cout << "Iterator error" << endl;
        }
    };
private:
    const List* collection;
    node* cur;
public:
    listIterator(const List*s, node* e):collection(s),cur(e){}
    const int operator *() {
        if (cur == collection->dummy) throw Error();
        return cur->data;
    }
    listIterator operator++(); //префиксный ++
    listIterator operator--(); //префиксный --
    int operator == (const listIterator& ri) const;
    int operator != (const listIterator& ri) const;
};
```

В операции разыменования необходимо проверять, что текущий указатель не ссылается на dummy, поскольку это означает, что итератор находится в позиции end().

```
if (cur == collection->dummy) throw Error();
```

Такая проверка требует наличия доступа к полям класса `List`. Чтобы не добавлять методы доступа к полям, объявляем класс `listIterator` дружественным классу `List`.

Операция инкремента выполняет такую же проверку, чтобы не допустить выполнение `operator++()` для пустого списка или для итератора в позиции `end()`.

```
listIterator listIterator:: operator++() {
    if (cur!=collection->dummy) {
        cur = cur->next;
        return *this;
    }
    else throw Error();
    // или collection.isEmpty() или cur достиг end()
}
```

Для операции декремента требуется другое условие, чтобы не допустить выполнение `operator--()` для пустого списка или для первого элемента списка.

```
listIterator listIterator:: operator--() {
    if (cur->prev != collection->dummy) {
        cur = cur->prev;
        return *this;
    }
    else throw Error();
    // или collection.isEmpty() или cur достиг begin()
}
```

Реализация операций сравнения итераторов на равенство/неравенство не меняется.

```
int listIterator:: operator==(const listIterator& ri) const {
    return (cur == ri.cur);
}

int listIterator:: operator!=(const listIterator& ri) const {
    return !(*this == ri);
}
```

Продемонстрируем возможность использования операции декремента в функции печати в обратном порядке двусвязного списка, задаваемого диапазоном.

```
void reverseprint(listIterator b, listIterator be) {
    while (be != b) {
        cout << *(--be) << ' ';
    }
    cout << endl;
}
```

Если дополнить реализацию функцией определения позиции первого максимального элемента в списке (по аналогии с примером 90), то можно проверить работу со списком, используя следующий код.

```
int main() {
    List l1;
    l1.inHead(100);
    for (int i = 0; i < 5; ++i)
        l1.inHead(i);
    for (int i = 5; i < 10; ++i)
        l1.inTail(i);
    cout << " list \n" << l1 << endl;
    List l2(l1);
    cout << " list \n" << l2 << endl;
    listIterator mt = max(l2.begin(), l2.end());
    cout << *mt << endl;
    reverseprint(l2.begin(), mt);
    reverseprint(mt, l2.end());
    return 0;
}
```

7.10. Классы для рекурсивных типов данных

Рекурсивное определение данных возникает, когда структура данных ссылается на объект такой же структуры. При этом в случае одной ссылки чаще всего в алгоритмах обработки эффективней использовать итерацию. Если же ссылок больше одной (как в деревьях), то в большинстве случаев применяются рекурсивные реализации. Рекурсия помогает разрабатывать изящные и эффективные структуры данных и алгоритмы в тех случаях, когда решение без использования рекурсии оказывается сложным и неочевидным.

Рекурсивное определение бинарного дерева можно реализовать как класс, содержащий внутри себя указатель на узел дерева (корень) и рекурсивное описание узла дерева. При этом открытые функции-члены

класса не будут рекурсивными. Второй способ определения класса бинарного дерева состоит в том, чтобы определить в нём указатели на два поддерева и открытые рекурсивные функции для работы с деревом.

Для демонстрации этих двух подходов определим классы с минимально необходимым количеством функций-членов.

Пример 93. Реализовать класс бинарного дерева поиска, определив в нём указатели на два поддерева и открытые рекурсивные функции для работы с деревом: конструктор, конструктор копии, деструктор, добавление элемента в дерево и вывод всех элементов дерева в поток.

```
class TreeR{
private:
  int data;
  TreeR *lt;
  TreeR *rt;

public:
  TreeR (int val=0, TreeR *l = nullptr, TreeR *r = nullptr):
         data(val), lt(l), rt(r) {}
  TreeR(const TreeR * t){
    if (t->lt) lt = new TreeR(t->lt);
      data = t->data;
    if (t->rt) rt = new TreeR(t->rt);
  }

  ~TreeR(){
    if (lt != nullptr) delete lt;
    if (rt != nullptr) delete rt;
  }

  void add(int a){
    if (data > a){
      if (lt)
        lt->add(a);
      else
        lt = new TreeR(a);
    }
    else{
      if (rt)
        rt->add(a);
      else
        rt = new TreeR(a);
    }
  }
```

```
  friend ostream& operator<<(ostream& os, const TreeR *t){
    if (t->lt) os << t->lt;
    os <<  t->data << "  ";
    if (t->rt) os<< t->rt;
    return os;
  }

  void printRKL(){
    if (rt) rt->printRKL();
    cout << "  " << data << " ";
    if (lt) lt->printRKL();
  }
};
```

В данном примере вывод в поток реализован двумя способами: перегрузкой операции выдачи в поток << и функцией `printRKL`.

В перегруженную операцию << указатель на дерево передаётся параметром, посредством которого реализуются рекурсивные вызовы.

```
friend ostream& operator<<(ostream& os, const TreeR *t)
```

Аналогично реализуется рекурсия и в функции `printRKL`, в которую указатель на дерево передаётся через неявный параметр `this`.

```
if (rt) rt->printRKL();
```

эквивалентно

```
if (this->rt) this->rt->printRKL();
```

При использовании класса `TreeR` приходится объявлять переменную указатель на объект и создавать объект динамически. При этом конструктор без параметров создаёт не пустое дерево, а дерево с одной вершиной, имеющей значение по умолчанию.

```
TreeR *t= new TreeR();// это непустое дерево
t->add(5);
t->add(-7);
t->add(3);
t->add(-4);
cout <<"T "<< t << endl;
t->printRKL();
TreeR *t1 = new TreeR(t);
cout << "T1 "<<t1 << endl;
```

Для класса `TreeR` пустота дерева определяется не на уровне класса, а на уровне указателя на объект класса.

```
TreeR* t0= nullptr; // это пустое дерево
```

Поэтому нельзя реализовать функцию-член класса для проверки на пустоту. Для безопасной работы перед вызовом функций-членов класса следует выполнять проверку на пустоту следующим образом:

```
if (t0!=nullptr) t0->add(3);
if (t0!=nullptr) t0->RKL();
if (t0!=nullptr) cout <<"T "<< t0 << endl;
```

Для динамических объектов при выполнении операции `delete` вызывается деструктор, но указатель на объект не обнуляется. Если предполагается дальнейшее использование указателя, ему необходимо явно присвоить значение `nullptr`.

```
if (t!=nullptr){
  delete t;
  t=nullptr;
}
```

При использовании операции удаления отдельных узлов из дерева возникает проблема при удалении последней вершины, которая не может быть решена в самой операции.

Пример 94. Реализовать класс бинарного дерева поиска, содержащий внутри себя указатель на узел дерева (корень) и рекурсивное описание узла дерева. Определить открытые функции-члены класса: конструктор, конструктор копии, деструктор, добавление элемента в дерево и вывод всех элементов в поток. При их реализации использовать рекурсивные функции в закрытой части описания класса.

```
class Tree{
private:
  struct TNode;
  typedef TNode* node_ptr;

  struct TNode{
    int data;
    node_ptr lt, rt;
```

```
    TNode (int val, node_ptr l=nullptr, node_ptr r=nullptr):
           data(val), lt(l), rt(r){}
  };

  node_ptr root;
  void delTree(node_ptr t){
    if (t!=nullptr){
      delTree(t->lt);
      delTree(t->rt);
      delete t;
    }
  }

  void add(node_ptr& t, int a){
    if (t == nullptr)
      t = new TNode(a);
    else if (t->data > a)
      add(t->lt, a);
    else
      add(t->rt, a);
  }

  void printLKR(node_ptr t, ostream& os) const{
    if (t){
      printLKR(t->lt, os);
      os << t->data << " ";
      printLKR(t->rt, os);
    }
  }

  void copy(node_ptr t, node_ptr &newT) const {
    if (t != nullptr){
      newT = new TNode(t->data, 0, 0);
      copy(t->lt, newT->lt);
      copy(t->rt, newT->rt);
    }
    else newT = nullptr;
  }

public:
  Tree(): root(nullptr) {}

  Tree(const Tree& t){
    copy( t.root, root);
  }

  ~Tree(){
    delTree (root);
  }
```

```
  void addNode(int a){
    add(root, a);
  }

  friend ostream& operator<<(ostream& os, const Tree &t){
    t.printLKR(t.root,os);
    return os;
  }
};
```

В примере типы `TNode` для узла дерева и `node_ptr` для указателя на узел определены внутри класса, что подчёркивает инкапсуляцию внутренней организации класса `Tree`. Именно в определении типа `TNode` присутствует рекурсия. Поэтому рекурсивные функции оперируют с указателями на узел дерева и должны быть объявлены как `private`. Для доступа к ним используются открытые функции класса, которые вызывают рекурсивные функции, передавая в качестве параметра указатель на корень дерева.

Такое описание класса допускает существование пустого дерева. Конструктор без параметров создаёт пустое дерево. Если предусмотреть операцию удаления узла дерева, то можно получить в процессе её выполнения пустое дерево. Для безопасной работы с таким деревом рекомендуется иметь операцию проверки дерева на пустоту.

```
Tree t;
t.addNode(5);
t.addNode(7);
t.addNode(3);
t.addNode(4);
cout <<"T "<< t << endl;
Tree t1(t);
cout << "T1 "<<t1 << endl;
```

Очевидно, что реализация класса бинарного дерева в примере 94 лишена недостатков, перечисленных в примере 93. Именно такой способ реализации рекомендуется использовать при создании классов для рекурсивных структур.

ГЛАВА 8. ОБОБЩЁННЫЙ ПОДХОД

Одной из важнейших целей объектно-ориентированного программирования является поддержка повторного использования кода. Один из механизмов для достижения этой цели — шаблоны C++. Шаблоны функций, которые были рассмотрены в разделе 4.3, позволяют избавиться от множественной перегрузки функций. При этом обобщается один и тот же алгоритм для разных типов данных. Этот подход относится к обобщённому программированию.

Обобщённое программирование (*generic programming*) заключается в описании структур данных и алгоритмов в терминах типов, подставляемых в качестве параметров в эти алгоритмы и структуры [18]. Обобщённое программирование является одним из проявлений полиморфизма. Обобщённый подход в языке C++ основывается на шаблонах функций и шаблонах
классов.

8.1. Шаблоны классов

Шаблоны классов, так же, как и шаблоны функций, являются трафаретами, по которым компилятор создаёт *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.

Например, шаблон класса `Stack` может служить основой для создания многочисленных классов `Stack`: «`Stack` для данных типа `int`», «`Stack` для данных типа `double`», «`Stack` для данных типа `Time`» и т. д.

Пример 95. Создать шаблон класса `Stack` с реализацией на основе массива.

```
//Шаблон класса Stack         файл stack.h
#pragma once
#include <iostream>

class stackEmpty {};

class stackFull {};

template <typename T>
class Stack {
private:
  T * start;   //указатель на стек
  int head;    //положение вершины стека
  int size;    //размер стека
public:
  //конструктор с параметром по умолчанию для размера стека
  Stack(int = 10);
  ~Stack() { delete[] start; }          //деструктор
  void push(const T&);   //добавление элемента в стек
  T pop();     //извлечение элемента из стека
  T top();     //просмотр элемента на вершине стека
  bool isEmpty() const {  //true, если стек пустой
    return head == -1;
  }
  bool isFull() const {  //true, если стек полон
    return head == size - 1;
  }
};

//конструктор
template <typename T>
Stack<T>::Stack(int n) {
  //определение «разумного» размера стека
  size = n>0 && n<1000 ? n : 10;
  start = new T[size];
  head = -1;
}

// добавление объекта в стек
//в случае переполнения стека выбрасывается исключение
template <typename T>
void Stack<T>::push(const T& x) {
  if (isFull())
    throw stackFull();
  start[++head] = x;
}
```

```
//извлечение элемента из стека
template <typename T>
T Stack<T>::pop() {
  if (isEmpty()) throw stackEmpty();
  T x = start[head--];
  return x;
}
//просмотр элемента на вершине стека
template <typename T>
T Stack<T>::top() {
  if (isEmpty()) throw stackEmpty();
  T x = start[head];
  return x;
}
```

Определение шаблона класса `Stack` похоже на определение класса, только впереди добавляется конструкция:

```
template <typename T>
```

Добавленная к заголовку конструкция `template <typename T>` указывает на то, что описывается шаблон класса `Stack` с параметром типа `T`. Идентификатор `T` определяет тип данных-элементов, хранящихся в стеке, и может использоваться при описании типов полей класса и в функциях-членах класса.

Компилятор рассматривает все функции-члены класса как шаблоны функций, параметры которых совпадают с параметрами шаблона класса. Поэтому каждое определение функции-члена класса вне шаблона класса начинается с конструкции:

```
template <typename T>
```

Внешнее определение каждой функции-члена шаблона класса использует операцию разрешения области видимости с именем шаблона класса `Stack<T>`.

☺ Хорошей идеей является написать и протестировать конкретный класс, а затем преобразовать его в шаблон. Таким образом, можно решить многие проблемы проектирования и обнаружить большую часть ошибок кода в тексте конкретного класса.

Аналогично шаблонам функций, все шаблоны классов и структур должны быть помещены в заголовочные файлы. Для шаблонов классов в отличие от просто классов выносить реализации их членов-функций в отдельный *.cpp файл нельзя. Это связано с тем, что шаблоны в языке C++ не компилируются, потому что шаблон представляет собой не фрагмент программного кода, а лишь инструкции для построения кода.

Для шаблонов классов и структур код генерируется в момент определения объекта шаблонного класса или структуры. Причём генерируются только те члены-функции класса, которые используются.

Подстановка конкретного типа в шаблон приводит к созданию шаблонного класса, т.е. к *инстанцированию* шаблона (*template instantiation*). Аналогично функция инстанцируется (генерируется, конкретизируется) из шаблона функции и аргумента шаблона. Использование различных терминов для одного понятия связано с терминологическими проблемами, возникающими при переводе. О них мы уже говорили в разделе «Используемые термины».

Количество созданных шаблонных классов (инстанций) зависит от количества используемых типов, подставляемых в шаблон при объявлении объекта. Версия шаблона для конкретного аргумента шаблона называется специализацией. Только специализации шаблонов содержат «настоящий» код, а не его шаблон. Генерация версий шаблона для набора аргументов шаблона является задачей компилятора, а не программиста.

По умолчанию, шаблон предоставляет единственное определение, которое должно использоваться для всех аргументов шаблона. Явная специализация позволяет обеспечить альтернативные реализации шаблона. Выделяют специализации, определяемые пользователями, или просто пользовательские специализации. Например, 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.

Так шаблон класса Stack может использоваться практически для любого типа. Однако может возникнуть необходимость создать специфический класс Stack для некоторого типа, например, Specific. Для этого нужно просто создать новый класс с именем Stack<Specific>.

```
Stack<Specific>{
...//определение специфического стека
}
```

Рассмотрим текст программы, в которой используется шаблон класса 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);
  cout << intStack.top() << endl;
  while (!intStack.isEmpy())
    cout << intStack.pop() << ' ';
  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 создаётся объект intStack типа Stack<int>, конструктор класса Stack создаёт массив элементов типа int, представляющий элементы данных стека.

В шаблонном классе Stack<int> компилятор замещает оператор

```
start=new T[size];
```

из описания шаблона класса Stack оператором

```
start=new int[size];
```

Шаблон класса `Stack` использовал только один параметр типа в заголовке шаблона. Но в шаблонах имеется возможность использования любого количества параметров. Кроме параметров типа, могут использоваться и нетиповые параметры. Например, заголовок можно модифицировать, указав в нем нетиповой параметр `int elements` для задания размера стека:

```
//заголовок для шаблона класса StackParam
//аналога шаблона класса Stack
template <typename T, int elements>
class StackParam {
private:
  T start [elements];    // массив для размещения данных стека
  int head;    //положение вершины стека
  int size;    //размер стека
public:
  StackParam ():size(elements),head(-1){}    //конструктор
  ~ StackParam () {}                    //деструктор
  void push(const T&);        //добавление элемента в стек
  T pop();                    //извлечение элемента из стека
  T top();          //просмотр элемента на вершине стека
  bool isEmpty() const {  //true, если стек пустой
    return head == -1;
  }
  bool isFull() const {  //true, если стек полон
    return head == size - 1;
  }
};
```

Тогда объявление типа

```
StackParam <int, 100> intStack1;
```

приведет к созданию во время компиляции шаблонного класса `StackParam` с именем `intStack1`, состоящего из 100 элементов данных типа `int`. Этот шаблонный класс будет иметь тип `StackParam<int,100>`.

В описании класса в разделе закрытых данных-членов помещено следующее объявление массива:

```
T start [elements];
//массив для размещения данных стека
```

Поэтому необходимость выделение динамической памяти под массив в конструкторе и освобождение её в деструкторе отпадает.

Если размер класса контейнера, например массива или стека, может быть определён во время компиляции (например, при помощи нетипового параметра шаблона, указывающего размер), это устранит расходы при динамическом выделении 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.

Поскольку у шаблона класса изменился список параметров, необходимо внести изменения в определение шаблонов членов-функций класса.

```
// добавление объекта в стек
//в случае успеха возвращается true, в противном случае false
template <typename T, int elements>
void StackParam<T,elements>::push(const T& a) {
  if (isFull())
    throw stackFull();
  start[++head] = a;    //элемент помещается в стек
}

//выталкивание элемента из стека
template <typename T, int elements>
T StackParam<T, elements>::pop() {
  if (isEmpty())
    throw stackEmpty();
  T a = start[head--];  //элемент выталкивается из стека
  return a;
}
//просмотр элемента из вершины стека
template <typename T, int elements>
T StackParam<T, elements>::top() {
  if (isEmpty()) throw stackEmpty();
  T y = start[head];  //элемент выталкивается из стека
  return y;
}
```

Пример 96. Дана символьная строка, содержащая латинские буквы, цифры и четыре вида скобок: ( ),[ ],{ }, < >. Проверить правильность расстановки скобок — каждой открывающей соответствует закрывающая скобка того же вида. Для решения задачи необходимо использовать стек.

Воспользуемся шаблоном класса `StackParam`.

```
#include <iostream>
#include <string>
#include "paramstack.h"

using namespace std;

class unpair {};
class missRight{};

int main() {
  locale::global(locale(""));
  string str;
  cin >> str;
  string left = "([{<";
  string right = ")]}>";
  StackParam<char,1024> St;
  char q;
  int pr;
  try {
    for (char c : str) {
      if (left.find(c) != string::npos)
        St.push(c);
      else {
        pr = right.find(c);
        if (pr != string::npos) {
          q = St.pop();
          if (pr != left.find(q))
            throw unpair();
        }
      }
    }

    if (!St.isEmpty()) throw missRight();
    cout <<"скобки расставлены верно" << endl;
  }
  catch (stackEmpty) {
    cout << "не хватает левой скобки" << endl;
  }
  catch (unpair) {
    cout << "непарные скобки" << endl;
  }
```

```
  catch (missRight) {
    cout << "не хватает правой скобки" << endl;
  }
  return 0;
}
```

Принцип проверки основан на том, что каждая открывающаяся скобка заносится в стек, а каждая закрывающаяся выталкивает из стека открывающуюся скобку. Полученная пара проверяется на соответствие.

При этом возможны три ошибочные ситуации, которые обрабатываются с помощью исключений.

8.2. Шаблоны коллекций

В коллекциях важно не столько тип элементов коллекции, сколько принцип организации хранения элементов и доступа к ним. Поэтому шаблоны являются удобным инструментом работы с коллекциями.

Пример 97. Реализовать шаблон класса «Линейный односвязный список». Список должен быть реализован на основе ссылочной организации.

Воспользуемся для создания шаблона линейного списка шаблоном структуры `node`:

```
template<typename T>
struct node{
  T data;
  node<T>* next;
  node(T data, node<T>* next) {
    this->data = data;
    this->next = next;
}};

template<typename Q>
ostream & operator << (ostream & out, const node<Q> & x) {
  out << x.data;
  return out;
}
```

Шаблон конструктора `node` содержит два параметра для заполнения информационной и ссылочной частей.

```
node(T data, node<T>* next) {
  this->data = data;
  this->next = next;
}
```

Такая форма конструктора позволяет легко добавлять новый узел в любое место списка. Например:

```
//создание единственного элемента pn1
node<int>* pn1 = new node<int>(5, nullptr);
//добавление элемента перед pn1
node<int>* pn2 = new node<int>(5, pn1);
//добавление элемента за pn1
pn1->next = new node<int>(7, nullptr);
```

Шаблон перегруженной операции вывода в поток для шаблона структуры `node` в поток не требуется объявлять дружественным, поскольку по умолчанию в структуре все поля открыты. При этом имя параметра типа в шаблоне не обязано совпадать с именем параметра в шаблоне структуры. В нашем примере используются `Q` и `T`.

```
template<typename Q>
ostream & operator << (ostream & out, const node<Q> & x) {
  out << x.data;
  return out;
}
```

Теперь, используя шаблон структуры `node`, описываем шаблон класса `list` для реализации линейного односвязного списка.

```
template<typename T>
class list{
private:
  node<T>* first;
  node<T>* last;
  error err;
public:
  list(): first(nullptr), last(nullptr) {    }
  ~list();
  bool isEmpty() const;
  void addFirst(T x);
  void addLast(T x);
  T getFirst()const;
  T getLast()const;
  T delFirst();
  T delLast();
  //Обработка с предикатом
  int kol(bool(*f) (T));
```

```
  void for_each(void(*action)(T&));
  template<typename Q>
  friend ostream & operator<< (ostream &out, const list<Q>
&y);
  //в полной реализации могут быть еще функции
};
```

Поскольку в процессе работы возможно возникновение исключительных ситуаций, создаём пустой класс `error`.

```
class error{};
```

Реализация деструктора и функции проверки на пустоту не содержат никаких особенностей.

```
template<typename T>
list<T>::~list(){
  while (first){
    node<T>* cur = first;
    first = first->next;
    delete cur;
  }
  last = first = nullptr;
}
template<typename T>
bool list<T>::isEmpty() const {
  return (first == nullptr);
}
```

Обратим внимание на объявление шаблона дружественной функции. Имя параметра шаблона дружественной функции может быть любым. Чтобы это подчеркнуть мы использовали имя `Q`, хотя могли оставить `T`.

```
template<typename Q>
friend ostream & operator<< (ostream &out, const list<Q> &y);
```

При этом в описании реализации шаблона дружественной функции не обязательно использовать то же самое имя параметра шаблона.

```
template<typename T>
ostream & operator << (ostream & out, const list<T> & y) {
  node<T>* p = y.first;
  while (p) {
    out << *p<<" ";
    p = p->next;
  }
  return out;
}
```

В шаблоне функции добавления элемента в начало списка используем шаблон конструктора структуры `node`. В качестве значения второго параметра используем указатель на начало списка `first`. При этом если список пуст, выполняется инициализация не только указателя `first`, но и указателя `last`.

```
template<typename T>
void list<T>::addFirst(T x){
  first = new node<T>(x, first);
  if (!last)
    last = first;
}
```

Обратите внимание, что добавление в конец имеет более сложный алгоритм.

```
template<typename T>
void list<T>::addLast(T x){
  if (first){
      last->next = new node<T>(x, nullptr);
      last = last->next;
  }
  else{
    first = last = new node<T>(x, nullptr);
  }
}
```

В шаблонах функций чтения первого и последнего элементов списка возможны ошибки доступа в случае, если список пуст. При этом выбрасывается исключение. Выбрасываемый объект `err` класса `error` объявлен в закрытой части шаблона класса `list`.

```
template<typename T>
T list<T>::getFirst() const{
  if (first)
    return first->item;
  else  throw err;
}
template<typename T>
T list<T>::getLast()const{
  if (first)
    return last->item;
  else
    throw err;
}
```

Функции удаления первого и последнего элементов списка в качестве результата возвращают значение удалённого элемента. В случае пустого списка они выбрасывают исключение.

```
template<typename T>
T list<T>::delFirst(){
  T x;
  if (first){
    x = first->data;
    node<T>*t = first;
    first = first->next;
    delete t;
    return x;
  }
  else
    throw err;
}

template<typename T>
T list<T>::delLast(){
  T x;
  if (first){
    x = last->data;
    node<T>*t = first;
    while (t->next != last)
      t = t->next;
    delete last;
    last = t;
    last->next = nullptr;
    return x;
  }
  else
    throw err;
}
```

Шаблон функции `int kol(bool(*f)(T))` реализует подсчёт количества элементов, для которых выполняется условие, задаваемое одноместным предикатом. Предикат задаётся указателем на функцию, соответствующую типу `bool(*f)(T)`.

```
//Обработка с предикатом
template<typename T>
// f — переменная типа "указатель на функцию"
int list<T>::kol(bool(*f) (T)) {
  int k = 0;
  for (node<T>* p = first; p; p = p->next)
```

```
    if (f(p->data))
      k++;
  return k;
}
```

В качестве примера такой функции для специализации шаблона списка типом `int` использована функция проверки на нечётность.

```
bool odd(int x){
  return x % 2 != 0;
}
```

Шаблон функции `void for_each(void(*action)(T&))` реализует применение функции `action` к информационному полю каждого элемента списка. Параметр `action` является указателем на функцию, соответствующую типу `void(*action)(T&)`.

```
template <typename T>
// action — переменная типа "указатель на функцию"

void list<T>::for_each(void(*action)(T&)) {
  node<T>* p = first;
  while (p) {
    action(p->data);
    p = p->next;
  }
}
```

В качестве примера такой функции для специализации шаблона списка типом `int` использована функция удвоения значения.

```
void mult2(int & x) {
  x *= 2;
}
```

В функции `main` использованы специализации шаблона списка типами `int` и `char`.

```
int main(void){
  list<int> l1;
  l1.addFirst(3);
  l1.addLast(4);
  l1.addLast(99);
  cout << l1<<endl;
  cout << l1.kol(odd)<<endl;
  l1.for_each(mult2);
  cout << l1 << endl;
  cout << l1.kol(odd) << endl;
```

```
  list<char> *l2 = new list<char>;
  l2->addFirst('1');
  l2->addLast('q');
  l2->addLast('a');
  cout << *l2 << endl;
  cout << *l2 << endl;
  cout << l2->delFirst() << endl;
  cout << *l2 << endl;
  cout << l2->delLast() << endl;
  cout << *l2 << endl;

  return 0;
}
```

Пример 98. Реализовать шаблон итератора для шаблона класса «Линейный односвязный список».

Предположим, что уже имеется шаблон класса линейный односвязный список из примера 97. Необходимо только добавить в него методы, возвращающие итераторы на начало и конец списка.

```
template<typename T>
class listIterator;

template<typename T>
class list{
private:
..node<T>* first;
..node<T>* last;
..error err;
public:
...
..listIterator<T> list<T>::begin() const;
..listIterator<T> list<T>::end() const;
};
```

В шаблоне итератора для шаблона линейного односвязного списка определены: операция доступа к элементу, префиксная операция инкремента и две операции сравнения итераторов. Реализация шаблона итератора аналогична реализации итератора из примеров 90 и 91.

```
template <typename T>
class listIterator {
public:

..class Error {
..public:
```

```
....void what(){
......cout << "Iterator error" << endl;
....}
..};

private:
..const list<T> *collection;
..node<T> *cur;

public:
..listIterator(const list<T>*s,node<T>*e):collection(s),cur(e){}
..T operator *(){
....if (cur)
......return cur->data;
....else
......throw Error();
..}
..listIterator operator++(); //префиксный ++
..bool operator == (const listIterator<T> &ri) const;
..bool operator != (const listIterator<T> &ri) const;
};

template <typename T>
listIterator<T> listIterator<T>:: operator++() {
..if (cur) {
....cur = cur->next;
....return *this;
..}
..else throw Error();
}

template <typename T>
bool listIterator<T>:: operator==(const listIterator<T> &ri) const {
..return (cur == ri.cur);
}
template <typename T>
bool listIterator<T>:: operator!=(const listIterator<T> &ri) const {
..return !(*this == ri);
}

template <typename T>
listIterator<T> list<T>::begin() const {
..return listIterator<T>(this, first);
}

template <typename T>
listIterator<T> list<T>::end() const {
..listIterator<T> iter(this, nullptr);
..return iter;
}
```

Пример 99. Создать шаблон класса `matrix` для представления матрицы как вектора, элементами которого являются векторы.

На основе класса `myvector` из примера 82 создадим шаблон класса `myvector<T>`.

```
template <typename T>
class myvector {
private:
  int size;
  T * vect;
public:
  myvector(int s = 1): size(s){
    vect = new T[s];
  }

  myvector(const myvector<T> & v): size(v.size){
    vect = new T[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
  }

  myvector(myvector<T>&& v){
    size = v.size;
    vect= v.vect;
    v.vect = nullptr;
  }
  ~myvector(){
    if (vect != nullptr)
    delete[] vect;
  }
  T& operator [] (int i){
    return vect[i];
  }
  T operator [] (int i) const {
    return vect[i];
  }

  myvector<T>& operator= (const myvector<T> &v){
    if (&v != this) {
      delete[] vect;
      vect = new T[v.size];
      for (int i = 0; i<v.size; ++i)
        vect[i] = v.vect[i];
      size = v.size;
    }
    return *this;
  }
```

```
  myvector<T>& operator=(myvector<T>&& v){
    if (vect != nullptr)
      delete[] vect;
    size = v.size;
    vect = v.vect;
    v.vect = nullptr;
    return *this;
  }

  myvector<T> operator+(const myvector<T>& v){
    if (size == v.size) {
      myvector<T> v1(size);
      for (int i = 0; i < size; ++i)
        v1[i] = vect[i] + v[i];
      return v1;
    }
    return *this;//лучше исключение использовать
  }
};
```

Попробуем объявить матрицу как вектор векторов.

```
myvector<myvector<int>> m(3);
```

Обратим внимание, что можно задать только одну размерность — количество строк, поскольку негде указать количество столбцов. Это связано с тем, что `myvector<int>` — это параметр типа для шаблонного класса `myvector<myvector<int>>`. При этом для каждого элемента `m[i]` будет вызван конструктор без параметров, в нашей реализации для него задано значение размера массива по умолчанию — 1.

Чтобы изменить количество столбцов в матрице, необходимо для каждой строки выполнить функцию изменения размера `resize`. Добавим её в шаблон класса `myvector<T>`.

```
template <typename T>
class myvector {
private:
  int size;
  T * vect;
public:
...
  void resize(int n){
    T* newVect = new T[n];
    int sz = (n < size) ? n: size;
```

```
    for (int i = 0; i<sz; ++i)
      newVect[i] = vect[i];
    delete[] vect;
    vect = newVect;
    size = n;
  }
};
```

Используем эту функцию в конструкторе шаблона класса матрицы на базе шаблона класса вектора.

```
template<typename T>
class matrix{
  int rows;
  myvector<myvector<T>> mdata;

public:
  matrix(int m, int n): rows(n), mdata(m){
    for (int i = 0; i < m; i++)
      mdata[i].resize(n);
  }
};
```

Обратим внимание на то, что вызывать конструктор `mdata(m)` в теле конструктора `matrix` уже поздно (уже отработает конструктор по умолчанию), а при объявлении ещё рано, поэтому конструктор необходимо вызывать в списке инициализации.

Деструктор для данного класса писать не надо, т.к. достаточно сгенерированного деструктора по умолчанию `~matrix(){}`. Поскольку деструктор по умолчанию вызывает деструкторы для всех полей класса, то будет вызван деструктор шаблона класса `myvector<myvector<T>>`.

Конструктор копии и `operator=` для шаблона класса `matrix` также сгенерируются автоматически и будут работать правильно.

Если в классе есть подобъект, который берёт на себя функции по выделению и освобождению динамической памяти, то определять конструктор копии и operator= не нужно. Они необходимы, только если непосредственно в конструкторе данного класса выделяется динамическая память, а в деструкторе возвращается.

Автоматически сгенерированный конструктор копии вызывает конструкторы копий для всех своих полей. Автоматически сгенерированная операция присваивания вызывает операции присваивания для всех своих полей. Например, сгенерированный конструктор копии будет выглядеть так:

```
matrix(const matrix<T> & mm): mdata(mm.mdata) {}
```

Доступ к элементу матрицы по индексам можно реализовать двумя способами. Первый способ предполагает перегрузку операции индексации для матрицы с возвращением вектора (по номеру строки). А элемент из этой строки возвращается перегруженной для вектора операцией индексации.

```
myvector<T>& operator [] (int i) {
  return mdata[i];
}

myvector<T> operator [] (int i)const {
  return mdata[i];
}
```

Второй способ предполагает перегрузку операции вызова функции `()` с двумя параметрами.

```
T& operator()(int i, int j){
  return mdata[i][j];
}
 T operator() (int i, int j) const {
  return mdata[i][j];
}
```

ГЛАВА 9. СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

9.1. Общая характеристика библиотеки

Механизм шаблонов встроен в компилятор C++, чтобы дать возможность программистам делать свой код короче за счёт обобщённого программирования. Внедрение этого механизма в реализацию стандартной библиотеки C++ привело к появлению *STL* (*Standard Template Library*) — *стандартной библиотеки шаблонов*.

Архитектура STL была разработана Александром Степановым, Менг Ли и другими сотрудниками AT&T Bell Laboratories и Hewlett-Packard Research Laboratories в начале 90-х годов. С 1998 года библиотека STL вошла в стандарты C++. Стандарт языка не называет её STL, так как эта библиотека стала неотъемлемой частью языка [10]. Однако до сих пор часто используется это название, чтобы отличать её от остальной части стандартной библиотеки.

Первоначальной целью STL было создание более гибкой модели контейнеров по сравнению с массивами и обобщение на них некоторых широко используемых алгоритмов (таких как поиск, сортировка и некоторых других). В дальнейшем возможности библиотеки были существенно расширены.

Главное, что следует сказать об обобщённых алгоритмах STL — это то, что поскольку они могут использоваться со многими или даже со всеми контейнерами, отпадает необходимость в определении соответствующих функций-членов у отдельных контейнеров, что снижает размер кода и упрощает интерфейсы контейнеров. Тем не менее, если эффективность алгоритма сильно зависит от внутренней реализации, он обычно реализуется как функция-член шаблона класса.

В STL выделяют шесть основных видов компонентов:

- ✓ *контейнеры* — объекты, которые хранят коллекции других объектов;
- ✓ *итераторы* — объекты доступа к содержимому контейнеров; они исполняют роль посредников между контейнерами и алгоритмами;
- ✓ *алгоритмы* — позволяют организовать обработку объектов вне зависимости от вида их организации в контейнерах;
- ✓ *адаптеры* — используются для изменения интерфейсов других компонентов STL;
- ✓ *функторы* (функциональный объекты) — объекты, каждый из которых скрывает функцию с целью передачи её в качестве параметра в алгоритмы;
- ✓ *аллокаторы* — используются контейнерами для выделения памяти объектам контейнеров и её освобождения.

Разработчики отказались от использования наследования при реализации библиотеки STL [10]. В частности, деструкторы в шаблонах классов не являются виртуальными. В связи с этим не рекомендуется разрабатывать собственные шаблоны контейнеров через наследование шаблонов STL.

☠ Не используйте наследование от шаблонов контейнеров STL.

Вся библиотека состоит только из заголовочных файлов. Организация STL на основе шаблонов имеет как достоинства, так и недостатки. В контейнерах можно использовать любые типы данных. Поскольку алгоритмы работают через итераторы, они могут быть применены к любым контейнерам, допускающим использование соответствующих итераторов. Этим обеспечивается универсальность. Однако шаблоны приводят к большему времени компиляции. Хотя исполняемый код может получиться более эффективным.

Использование библиотеки STL увеличивает скорость написания программ, поскольку предоставляет готовые решения по организации данных и реализации алгоритмов. Это позволяет создавать компактный код программы. С другой стороны, это влечёт сложность отладки из-за тяжёлых сообщений об ошибках.

Объём библиотеки STL не позволяет привести её полное описание в данной книге. В следующих разделах будут рассмотрены принципы и примеры использования STL. Для подробного изучения рекомендуем обратиться к специальной литературе [3, 8, 9, 10].

9.2. Контейнеры

Контейнер — это набор однотипных элементов, реализующий абстрактный тип данных. Простейшим прототипом контейнера в классическом языке C++ является массив.

STL вводит целый ряд разнообразных типов контейнеров: последовательные, ассоциативные, контейнеры-адаптеры и псевдоконтейнеры. К последовательным контейнерам относятся `deque`, `list`, `vector`, `array`, `forward_list`. К ассоциативным упорядоченным контейнерам — `map`, `multimap`, `set`, `multset`, `unordered_map`, `unordered_set`. Контейнеры-адаптеры ограничивают интерфейс базовых контейнеров — `priority_queue`, `queue`, `stack`. Певдоконтейнеры — это шаблоны классов, похожие на стандартные контейнеры, но удовлетворяющие не всем требованиям для стандартных контейнеров. К ним относятся `bitset`, `basic_string`, `valarray`, `vector<bool>`.

Как правило, элементы, хранимые в контейнерах STL, могут быть практически любого типа, если их можно копировать, т. е. имеют семантику значения. Если необходимо использовать собственный тип, то для него

должны быть обязательно определены: конструктор без параметров, конструктор копии, деструктор, операция присваивания. Дополнительно в некоторых случаях требуется определить `operator==` и `operator<`. Например, для каких-то типов контейнеров или для выполнения некоторых операций с контейнерами или их элементами.

Несмотря на то, что STL содержит большое разнообразие контейнеров, она позволяет разрабатывать собственные контейнеры, которые можно использовать с обобщёнными алгоритмами библиотеки. Для этого они должны удовлетворять определённому набору требований.

Все контейнеры должны иметь конструктор, создающий пустой контейнер, конструктор копирования и деструктор, который освобождает всю память, использованную контейнером, и вызывает деструктор каждого элемента в контейнере (напомним, что деструктор контейнера не виртуален). Возможно также наличие конструктора с параметрами — итераторами, задающими диапазон элементов в другом контейнере. Допустимы частные виды конструкторов в зависимости от типа контейнера.

Общий вид конструктора для любого контейнера `container` с элементами типа `T`:

```
container<T> c; // создание пустого контейнера
container<T> c2(c); // создание копии контейнера
container<T> c3(b,e); // создание контейнера с инициализацией
          // элементами из диапазона [b,e) другого контейнера
          // диапазон задается через итераторы
```

Одномерные массивы в C++ удовлетворяют всем требованиям к контейнерам и могут быть использованы в алгоритмах STL. При этом указатели играют роль примитивных итераторов. Удобно использовать массивы для инициализации других контейнеров.

```
//пустые контейнеры
list <int> il;
set<char> cs;
//контейнеры с инициализацией списком значений
vector<int> x{2, 5, 1, 3};
```

```
list<int> y{2, 5, 1, 3};
set<int> z{2, 5, 1, 3};
//контейнеры, созданные конструктором копии
vector<int> x_copy(x);
list<int> y_copy(y);
set<int> z_copy(z);

// Инициализация диапазоном другого контейнера
int a[] = {3,5,4,2,7};
vector <int> v(a,a+5);
vector<int> v1(a, a + 3);
list<double> l(a+2,a+5);
list<double> l2(l.begin(),l.begin()+1);

// частный случай конструктора для вектора
// задаётся количество элементов и значение для всех элементов
vector<int> v2(10,0);
```

Определение шаблона контейнера содержится в заголовочном файле, имя которого совпадает с именем контейнера. Поэтому для каждого контейнера нужно подключать соответствующий заголовочный файл. Например, для использования `vector` следует использовать

```
#include <vector>
```

Все контейнеры должны поддерживать операцию присваивания. Они должны также поддерживать все операции сравнения (=, ==, <, <=, !=, >, >=).

```
v = v1; // v, v1 - vector
l = l1; // l, l1 - list
```

Присваивание возможно только для объектов одного типа. Для тех случаев, когда `operator=` не подходит, используется функция `assign`, которая позволяет заполнить контейнер новыми данными. Например,

```
//разные типы контейнеров: v – vector, l - list
v.assign(l.begin(), l.end());
//перезаполнение существующего списка частью другого
контейнера
l.assign(a,a+3);
l.assign(l1.end()-3,l1.end());
```

Операции сравнения применимы только к контейнерам одного типа и выполняются в лексикографическом порядке.

```
if (v1 < v) ...; // Лексикографическое сравнение
```

Для того, чтобы контейнеры можно было обрабатывать алгоритмами STL, необходимо наличие у контейнеров функций-членов, возвращающих итераторы:

- ✓ `iterator begin()` — возвращает итератор, указывающий на первый элемент контейнера;
- ✓ `const_iterator begin() const` — возвращает константный итератор, указывающий на первый элемент контейнера;
- ✓ `iterator end()` — возвращает итератор, указывающий на позицию, следующую за последним элементом контейнера;
- ✓ `const_iterator end() const` — возвращает константный итератор, указывающий на позицию, следующую за последним элементом контейнера.

Кроме того, все контейнеры должны предоставлять ещё ряд функций:

- ✓ `bool empty() const` — возвращает `true`, если контейнер пуст;
- ✓ `void clear()` — очищает контейнер, делая его размер = 0;
- ✓ `size_type max_size() const` — возвращает максимальное количество элементов, которые может содержать контейнер;
- ✓ `size_type size() const` — возвращает количество элементов, в текущий момент хранящихся в контейнере;
- ✓ `void swap(ContainerType c)` — обменивает между собой содержимое двух контейнеров.

Все остальные функции зависят от конкретного вида контейнера.

Эффективность операций для различных контейнеров разная и зависит от внутреннего представления контейнера. Поэтому для каждой конкретной задачи нужно выбирать наиболее подходящий контейнер, исходя из того, какие именно операции при решении задачи будет использоваться чаще всего.

9.3. Последовательные контейнеры

В последовательных контейнерах сохраняется тот порядок, в котором элементы добавляются в контейнер, т. е. в них организуется хранение элементов в линейном порядке. К последовательным контейнерам относятся `vector`, `array`, `list`, `forward_list`, `dequee`.

Вектор является реализацией динамического массива. С точки зрения стоимости отдельных операций вектор следует использовать в задачах, требующих доступ к элементу по индексу. При этом следует учитывать, что операции вставки и удаления в конце вектора выполняются за постоянное время, вставка и удаление внутри вектора имеют линейную сложность.

Контейнер `array` добавлен в стандарт, начиная с версии C++11, с целью повышения эффективности в случае работы со статическим массивом. Предполагается использование нетипового параметра шаблона, позволяющего выделить память на этапе компиляции.

```
array<int, 4> Myarray;
array<int, 4> Myarray1 = {5,6,7,8};
```

Контейнер `array` имеет ту же семантику, что и массивы фиксированного размера. Размер и эффективность `array<T,N>` такие же, как у статического массива `T[N]`. В отличие от остальных контейнеров, `array` не позволяет управлять размещением элементов через аллокаторы, и для него не определены операции, изменяющие размер массива. Контейнер `array` предоставляет некоторые возможности стандартных контейнеров, такие как знание собственного размера, поддержка присваивания, итераторы произвольного доступа и т. д.

Контейнер `deque` — двусторонняя очередь, которая позволяет быстро выполнять операции вставки/удаления элементов в начале и в конце контейнера. В отличие от `vector` и `array` контейнер `deque` обычно реализован с помощью двустороннего списка массивов фиксированного

размера. Расширение deque дешевле, чем расширение vector, потому что оно не требует копирования существующих элементов в новый участок памяти.

Сложность операций вставки/удаления в начале и конце двусторонней очереди, а также операции произвольного доступа — постоянная. Для операций вставки/удаления внутри контейнера сложность — линейная.

Для контейнеров vector, array и deque доступ к элементу по индексу реализован двумя способами: перегруженной операцией operator[](i) и функцией at(i). Обе реализации подобны. Отличие заключается лишь в том, что функция at(i) контролирует выход за границу массива, выбрасывая в случае ошибки исключение out_of_range.

Контейнеры vector и array поддерживают итераторы произвольного доступа как прямой, так и обратный.

Пример 100. Проверить вектор целых чисел на симметричность относительно середины.

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> v;
  int n,a;
  cin >> n;
  for (int i = 0; i < n; ++i) {
    cin >> a;
    v.push_back(a);
  }
  bool b = true;
  if (v.size() > 0) {
    auto itb = v.begin();
    auto ite = v.end() - 1;
    while (b && itb < ite) {
      b = (*itb) == (*ite);
      itb++;
      ite--;
    }
  }
```

```
  cout << (b ? "yes":"no");
  return 0;
}
```

При заполнении вектора значениями использована функция добавления в конец `v.push_back(a)`, имеющая постоянную сложность. Если вектор заполнен (размер равен ёмкости), выполнение функции `push_back` автоматически приводит к увеличению ёмкости.

Если заранее известно количество вводимых элементов, то вместо `v.push_back(a)` можно использовать операцию обращения по индексу к элементу массива.

```
vector<int> v(10);
for (i=0; i<v.size(); ++i) cin>>v[ i ];
```

В приведённом решении продемонстрировано использование итераторов произвольного доступа, которые можно перемещать как в прямом (++, +=), так и в обратном направлении (`--` , -=).

```
auto itb = v.begin();
auto ite = v.end() - 1;
```

Поскольку функция `end()` возвращает итератор за пределами вектора, для доступа к последнему элементу необходимо сместить его на одну позицию назад. Однако, в случае пустого вектора (`v.size()==0`) эта операция приведёт к ошибке. Поэтому необходимо проверить вектор на непустоту.

```
if (v.size() > 0)
```

Оба итератора имеют одинаковый тип, поэтому допустимо использование операций сравнения.

```
while (b && itb < ite)
```

Тип переменных `itb` и `ite` для итераторов вектора определяется автоматически, поскольку использовано `auto`. Кроме повышения безопасности это удобно ещё и тем, что упрощает написание кода. Поскольку тип итератора является вложенным типом для соответствующего

контейнера. Для доступа к таким типам вне класса нужно разрешить область видимости.

```
//равносильные объявления
auto itb = v.begin();
vector <int>::iterator itb = v.begin();
```

Понятно, что первый способ компактнее и проще. При таком подходе возникает меньше ошибок.

Для вектора допустимо использование обратного (реверсивного) итератора `v.rbegin()`.

```
bool b = true;
auto itb = v.begin();
auto itrb = v.rbegin();
while (b && itb < itrb.base()) {
  b = (*itb) == (*itrb);
  itb++;
  itrb++;
}
cout << (b ? "yes":"no");
```

Для реверсивного итератора функция `rbegin()` указывает на последний элемент контейнера, с которого начнётся просмотр элементов в обратном порядке. Функция `rend()` указывает соответственно на позицию перед первым элементом контейнера. Операции `++` и `+=` для обратного итератора соответствуют операциям `--` и `-=` для прямого (базового) итератора, и наоборот.

```
//равносильные объявления
auto itrb = v.rbegin();
vector<int> :: reverse_iterator itrb = v.rbegin();
```

Использование обратного итератора позволяет избежать дополнительной проверки на непустоту вектора. Так как для пустого вектора значения итераторов `itb` и `itrb` окажутся равными.

```
while (b && itb < itrb.base())
```

Типы итераторов `itb` и `itrb` — разные, поэтому сравнение для них неприменимо. Но обратный итератор можно привести к базовому, используя функцию `base()`. И тогда итераторы можно сравнивать.

Пример 101. Дан вектор. Удалить элементы вектора, начиная с последнего нуля. При этом необходимо, чтобы ёмкость вектора соответствовала количеству элементов.

```
vector<int> v0 {0};
cout << v.size() << "  " << v.capacity() << endl;
auto it = find_end(v.begin(), v.end(), v0.begin(), v0.end());
v.erase(it, v.end());
cout << v.size() << "  " << v.capacity() << endl;
vector<int> v1(v);
v.swap(v1);
cout << v.size() << "  " << v.capacity() << endl;
```

STL содержит большое количество различных обобщённых алгоритмов, но подходящего для нашей задачи алгоритма нет. Поэтому для нахождения последнего нулевого элемента в векторе используем алгоритм поиска последнего вхождения подпоследовательности элементов. Для этого необходимо подключить заголовочный файл algorithm.

Входные параметры алгоритма — две пары итераторов, одна определяет диапазон вектора, в котором выполняется поиск, вторая — диапазон подпоследовательности, которую ищем. Для этого определим вектор из одного элемента, равного 0:

```
vector<int> v0 {0};
```

Результат поиска — итератор, указывающий на позицию первого элемента последнего вхождения подпоследовательности или значение `v.end()`, если подпоследовательность не найдена.

```
auto it = find_end(v.begin(), v.end(), v0.begin(), v0.end());
```

Для удаления элементов из вектора используется функция-член класса `erase()` с параметрами — парой итераторов, задающих диапазон удаляемых элементов.

```
v.erase(it, v.end());
```

Итератор может стать недействительным, и, таким образом, небезопасным, если элемент, на который указывал итератор, был

уничтожен. Если к недействительному итератору попытаться обратиться, то возникает ошибка времени выполнения.

```
v.erase(it, v.end());
//cout << *it << endl; //ошибка времени выполнения
```

Для дальнейшего использования переменной `it` потребуется её повторная инициализация.

Для контроля за ёмкостью и размером вектора использованы функции `size()` и `capacity()` шаблона `vector`

```
cout << v.size() << "  " << v.capacity() << endl;
```

Можно увидеть, что после выполнения функции `erase()`, размер вектора уменьшился, а ёмкость осталась прежней.

У вектора различают ёмкость (количество выделенной памяти) и размер (количество актуальных элементов). Для управления этими параметрами у вектора есть две функции: `reserve()` изменяет ёмкость, а `resize()` — размер.

Если функция `resize()` вызывается со значением большим чем текущий размер, то создаются дополнительные элементы конструктором без параметров. Если новый размер должен стать меньше, чем текущий размер, то лишние элементы удаляются из вектора, а ёмкость при этом не изменяется.

Функция `reserve()` может только увеличить ёмкость до заданного числа элементов (выделит память, но никакие конструкторы для элементов вызываться не будут), т. е. вызов `reserve()` не влияет на `size`. Уменьшить ёмкость с помощью функции `reserve()` невозможно. Для уменьшения ёмкости начиная с C++11 можно использовать вызов `shrink_to_fit()`. В нашем примере был использован не привязанный к версии универсальный подход с использованием функции `swap()`.

Для этого с помощью конструктора копии создаётся вспомогательный вектор нужных размера и ёмкости. Затем посредством функции `swap()` происходит обмен векторов. Функция `swap()` выполняется за постоянное время, независимо от размера контейнера. Это связано с тем, что меняются указатели на области памяти.

```
vector<int> v1(v);
v.swap(v1);
```

Если предполагается большое количество операций удаления/вставки в произвольных местах контейнера, то рекомендуется использовать двунаправленный список `list` или однонаправленный `forward_list`.

Пример 102. Дана строка, содержащая слова, разделённые запятыми, и завершающаяся точкой. После каждой запятой вставить символ пробела.

```
#include <list>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;
int main() {
  list<char> text;
  char c;
  do {
    cin >> c;
    text.push_back(c);
  } while (c != '.');

copy(text.begin(),text.end(),ostream_itertor<char>(cout,""));
  cout << endl;
  auto ite = text.end();
  auto itb = find(text.begin(),ite,',');
  while (itb != ite) {
    itb=text.insert(++itb, ' ');
    itb = find(itb,ite,',');
  }

copy(text.begin(),text.end(),ostream_itertor<char>(cout,""));
}
```

Поскольку предполагается выполнение большого количества операций вставки внутри строки, используем для хранения символов контейнер `list`.

Для поиска символа «запятая» использован алгоритм `find` поиска заданного значения в диапазоне элементов контейнера. Диапазон определяется двумя итераторами:

```
itb = find(itb,ite,',');
```

В случае неудачи алгоритм возвращает указатель `text.end()`. Поэтому условие продолжения преобразования строки выглядит так:

```
while (itb != ite)
```

Если же запятая найдена, то возвращается итератор на её позицию в списке. Функция `insert(++itb, ' ')` выполняет вставку в позицию перед итератором, поэтому необходимо сдвинуть итератор `++itb`. Функция возвращает итератор на вставленный элемент.

Для контроля работы алгоритма выводится список до и после преобразования. Для этого используем алгоритм `copy`, параметрами которого являются итераторы на начало и конец копируемого контейнера и итератор на начало контейнера, куда производится копирование. Чтобы использовать эту функцию для вывода в поток третьим параметром должен быть итератор вывода в поток `ostream`.

```
ostream_iterator<char>(cout,""))
```

Итератор создаётся по шаблону с параметром типа выводимых элементов. В нашем случае это `char`. В конструктор итератора передаётся поток `cout` и символ-разделитель для выводимых элементов. Поскольку в примере при выводе разделители не нужны, второй параметр конструктора — пустая строка.

```
copy(text.begin(),text.end(),ostream_iterator<char>(cout,""));
```

☺ Для вывода в поток содержимого контейнеров рекомендуется использовать алгоритм `copy` вместо цикла по контейнеру.

9.4. Ассоциативные контейнеры

В то время как последовательные контейнеры хранят свои данные линейно, с сохранением относительных позиций, в которые были вставлены элементы, ассоциативные контейнеры определяют порядок размещения элементов на основе ключей, которые хранятся в элементе (или, в некоторых случаях, представляют собой сам элемент). Общий подход к организации хранения ключей состоит в том, что они отсортированы в соответствии с некоторым отношением порядка. Это позволяет обеспечить максимальную скорость выборки на основе ключей.

В качестве структур хранения ассоциативных контейнеров используются сбалансированные деревья поиска и хеш-таблицы.

Ассоциативными контейнерами STL являются классы для представления множеств `set<K>`, `multiset<K>`, `unordered_set<K>`, `unordered_multiset<K>` и для представления отображений `map<K,V>`, `multimap<K,V>`, `unordered_map<K,V>`, `unordered_multimap <K,V>`.

В случае множеств и мультимножеств элементами данных являются сами ключи, причём мультимножество допускает наличие одинаковых ключей, а множество — нет. В отображениях и мультиотображениях элементы данных представляют собой пары, состоящие из ключей и собственно данных некоторого другого типа, причём мультиотображение допускает наличие одинаковых ключей, а отображение — нет.

С помощью бинарных деревьев поиска организованы контейнеры `map`, `set`, `multimap`, `multiset`. Для них скорость доступа к элементу по ключу равна O(*log n*). Контейнеры `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset` организованы как хеш-таблицы с константной скоростью доступа.

У каждой реализации есть свои достоинства и недостатки. Важно, чтобы базовые операции для ассоциативных контейнеров (вставка `insert`, удаление `erase`, поиск `find`) выполнялись как в среднем, так и в худшем случае за время O(*log n*). Для сбалансированных деревьев поиска (в том числе для красно-чёрных деревьев) это условие выполнено.

В реализациях, основанных на хэш-таблицах, среднее время оценивается как O(1), что лучше, чем в реализациях, основанных на деревьях поиска. Но при этом не гарантируется высокая скорость выполнения отдельной операции: время операции вставки в худшем случае оценивается как O(n). Она выполняется долго, когда коэффициент заполнения становится высоким и необходимо перестроить индекс хэш-таблицы.

Хэш-таблицы плохи также тем, что на их основе нельзя реализовать быстро работающие дополнительные операции `min`, `max` и алгоритм обхода всех хранимых пар в порядке возрастания или убывания ключей.

Ассоциативные контейнеры обладают многими свойствами, присущими последовательным контейнерам, поскольку они поддерживают обход элементов данных в виде линейной последовательности. Они предоставляют двунаправленные итераторы, обход с использованием которых даёт отсортированный порядок элементов. В некоторых случаях (например, когда элементы данных представляют собой большие структуры) сортировка последовательности элементов может быть выполнена более эффективно путём вставки их в мультимножество и обхода мультимножества, чем обобщённым алгоритмом сортировки или соответствующей функцией-членом списка.

Пример 103. Даны три множества, содержащих сведения о студентах, неуспевающих по предметам: «Языки программирования», «Математический анализ», «Дифференциальные уравнения». Сведения

содержат фамилию, имя, отчество и номер зачётной книжки, используемый в качестве уникального ключа. Найти студентов, неуспевающих по всем трём дисциплинам для формирования приказа на отчисление.

```
#include <iostream>
#include <set>
#include <algorithm>
#include <string>
#include <iterator>

using namespace std;
class Student{
private:
  string name;
// номер зачётной книжки (уникальное значение)
  string recordBook;
public:
  Student(string nm="noname", string rB="********"):
          name(nm),recordBook(rB) {}
  friend bool operator<(const Student & p1,const Student &
p2){
    return p1.recordBook < p2.recordBook;
  }
  string Name() { return name; }
};

int main(){
  set<Student> MA, DE, PL;
  /*
   здесь заполняются множества задолжников по:
   математическому анализу — MA
   дифференциальным уравнениям — DE
   языкам программирования— PL
   например, с использованием операции insert:
   MA.insert(Student("First Student", "MM11111"));
  */
  set<Student> expelledProj;
  insert_iterator<set<Student>> it_ex(expelledProj,
                                expelledProj.begin());
  set_intersection(MA.begin(),MA.end(),DE.begin(),DE.end(),it_ex);
  for(Student t :expelledProj) {
    cout << t.Name() << " ";
  }
  cout << endl;
  set<Student> expelled;
  set_intersection(expelledProj.begin(), expelledProj.end(),
                PL.begin(), PL.end(),
                inserter(expelled, expelled.begin()));
```

```
  for (Student t : expelled) {
    cout << t.Name() << " ";
  }
  return 0;
}
```

Шаблон `set` для пользовательских типов данных, используемых в параметре, накладывает требование — определить операцию `operator<`. Это связано с упорядоченностью множества.

В упрощённом классе `Student` определена перегруженная операция `operator<`.

```
friend
  bool operator<(const Student & p1,const Student & p2){
  return p1.recordBook < p2.recordBook;
}
```

Множество `set` не должно содержать повторяющихся значений. Тем не менее, реализовывать операцию сравнения ключей на равенство не требуется, поскольку проверка на совпадение реализуется через две операции `operator<`.

Определение эквивалентности `x == y` сводится к выражению `!((x < y)||(y < x))` или `!(x < y)&&!(y < x)`.

Бывают случаи, когда необходимы сравнения по разным ключам, а операция `operator<` реализует только одно сравнение. В этих случаях есть возможность определить класс для сравнения значений пользовательского типа. Такие классы называются *компараторами*.

Например, если сведения о студенте содержат баллы рейтинга `rating`, можно определить компаратор для упорядочения по баллам:

```
class StudentRating{
public:
  bool operator() (const Student & p1, const Student & p2){
    return p1.getrating() < p2.getrating();
  }
}
```

Поскольку у разных студентов могут быть одинаковые баллы, для упорядочения по рейтингу используем multiset с двумя параметрами: тип элементов множества и компаратор.

```
multiset<Student, StudentRating > s1;
```

Функция-член insert классов set и multiset получает единственный аргумент и вставляет копию этого аргумента.

```
MA.insert(Student("First Student", "MM11111"));
```

В случае multiset элемент всегда вставляется и результатом вставки является позиция итератора iterator, указывающая во множестве на элемент с заданным значением ключа. Для класса set дополнительно указывается, был ли уже этот элемент во множестве или он был добавлен в текущей операции. Поэтому в классе set тип возвращаемого значения имеет вид pair<iterator, bool>.

Использовать такой результат можно следующим образом:

```
set <int> s;
pair<it, bool> a = s.insert(1);// можно auto a = s.insert(1);
if (a.second)  // проверяем, вставлено или нет
  cout <<"добавлено "<< *(a.first);
else
  cout <<"уже есть во множестве "<<(a.first);
```

Алгоритм set_intersection строит отсортированное пересечение элементов из двух диапазонов. Поэтому пересечение трёх множеств находим за два шага.

Пятым параметром алгоритма set_intersection должен быть итератор вставки в результирующий контейнер. В нашем примере контейнером для результата является множество set<Student> expelledProj. В конструктор итератора вставки передаются контейнер и итератор, определяющий позицию вставки.

```
set<Student> expelledProj;
insert_iterator<set<Student>> it_ex(expelledProj,
                                    expelledProj.begin());
set_intersection(MA.begin(),MA.end(),DE.begin(),DE.end(),it_ex);
```

Чтобы не создавать дополнительную переменную для итератора вставки, можно использовать шаблонную функцию, возвращающую такой итератор `inserter(expelled,expelled.begin())`. Этот приём использован для нахождения второго пересечения.

```
set<Student> expelled;
set_intersection(expelledProj.begin(), expelledProj.end(),
                 PL.begin(), PL.end(),
                 inserter(expelled, expelled.begin()));
```

В библиотеке имеются алгоритмы для выполнения теоретико-множественных операций: `includes`, `set_union`, `set_intersection`, `set_difference`, `set_symmetric_difference`. Эти операции работают с любыми отсортированными контейнерами, включая ассоциативные.

Проиллюстрируем использование `set_intersection` для упорядоченных векторов.

```
int a[] = {1,2,3,4,5};
int b[] = {1,3,5,7};
vector<int> v1(a,a+5);
vector<int> v2(b,b+4);
vector<int> v;
set_intersection(begin(v1), end(v1), begin(v2),end(v2),
                 back_inserter(v));
```

В случае неупорядоченных контейнеров алгоритмы для теоретико-множественных операций завершаются ошибкой времени выполнения.

Пример 104. Использовать `map` для подсчёта частоты вхождения слов в текст.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;

int main() {
  ifstream ifs("source.txt");
  if (!ifs.is_open()) {
    cout<<"Не могу открыть файл словаря "<<"source.txt"<<endl;
```

```
    exit(1);
  }
  typedef istream_iterator<string> string_input;
  vector<string> text;
  map<string, int> dictionary;
  copy(string_input(ifs),string_input(),back_inserter(text));
  for (string s: text) {
    ++dictionary[s];
  }
  for (auto s : dictionary) {
    cout << s.first<<" - "<<s.second;
  }
  return 0;
}
```

Для удобства анализа текста читаем слова из текстового файла `ifs` сразу в вектор `text` с элементами типа `string`. Для этого используем уже рассмотренную функцию `copy`.

```
copy(string_input(ifs), string_input(), back_inserter(text));
```

Только теперь первые два параметра — это итераторы на начало и конец потока для текстового файла `ifs`, создаваемые конструкторами `string_input(ifs)` и `string_input()`. Третий параметр — функция `back_inserter(text)`, возвращающая итератор вставки в конец вектора `text`.

Контейнер `map` — ассоциативный контейнер, содержащий пары ключ-значение с неповторяющимися ключами. Порядок ключей задаётся операцией меньше или компаратором. Операции поиска, удаления и вставки имеют логарифмическую сложность. Данный тип, как правило, реализуется как красно-чёрное дерево [13].

Операция обращения по индексу позволяет работать с `map` как с массивом, используя в качестве индекса ключ. Если элемента с заданным ключом в контейнере `map` нет, то он добавляется. При этом ассоциированное по этому ключу значение заполняется значением по умолчанию. Для типа `int` это 0. Причём это не зависит от того, обращение по индексу входит в lvalue или rvalue выражения.

В нашем примере это позволяет не делать проверку на существование элемента в контейнере. В любом случае используется только

```
++dictionary[s];
```

Если элемента в контейнере ещё не было, то он добавится с ассоциированным значением, равным 0. Далее операция инкремента увеличит его на единицу.

Поскольку при обращении по индексу возможно неявное добавление элемента в `map` с указанием только ключа, то для типа ассоциированного значения необходим конструктор по умолчанию.

Просмотр содержимого частотного словаря организуется с помощью цикла `foreach` по словарю. Автоматически определяемый тип для параметра `s` представляет пару `pair<string, int>`. Доступ к данным в паре осуществляется через поля `first` и `second`.

```
for (auto s : dictionary) {
  cout << s.first<<" - "<<s.second;
}
```

Пример 105. Разработать структуру данных для хранения информации о лекторах, дисциплинах, которые они читают, и форме отчётности по этой дисциплине.

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

typedef map <string, map <string, string>> plan;
int main() {
  plan A;
  map<string,string> p ;
  p["ALG"] = "exam";
  p.insert(pair<string, string>("LOG", "exam"));
  p.insert(pair<string, string>("ML", "credit"));
  A.insert(pair<string, map<string, string>>("Hoar", p));
  A["Winner"]["PL"] = "exam";
  A["Winner"]["FL"] = "credit";
  A["Turing"]["TPL"] = "exam";
  A["Turing"]["TML"] = "credit";
  A["Virt"] = p;
```

```
  for (auto t : A) {
    cout << t.first<<endl;
    for (auto s : t.second) {
      cout << s.first << " " << s.second << endl;
    }
  }
  return 0;
}
```

Удобно каждому преподавателю сопоставить набор пар (дисциплина, отчётность). Тогда пары (дисциплина, отчётность) можно организовать как ассоциативный контейнер `map`.

```
map<string,string> p;
```

А все данные о лекторах и дисциплинах с отчётностями поместить в `map` следующей структуры

```
typedef map <string, map <string, string>> plan;
plan A;
```

На примере словарей `p` и `A` продемонстрированы различные способы добавления значений в `map`. Можно отметить, что использование операции доступа по индексу существенно упрощает работу со словарём.

Аналогия с двумерным массивом проявляется и в организации цикла просмотра всех элементов контейнера `map<string,map<string,string>>`

```
for (auto t : A) {
  cout << t.first<<endl;
  for (auto s : t.second) {
    cout << s.first << " " << s.second << endl;
  }
}
```

9.5. Итераторы

Итераторы занимают центральное место в организации STL благодаря их роли посредников между контейнерами и обобщёнными алгоритмами [9]. Они позволяют создавать обобщённые алгоритмы без учёта того, как именно хранятся последовательности данных, а контейнеры — без необходимости написания большого количества исходного текста работающих с ними алгоритмов. Однако по причинам эффективности

невозможно обеспечить возможность работы каждого обобщённого алгоритма с каждым контейнером. Чтобы определить применимость алгоритма к контейнеру, необходимо знать тип итератора, удовлетворяющего минимальным требованиям алгоритма, и тип итератора, предоставляемый контейнером.

Существуют пять основных типов итераторов: итераторы ввода, итераторы вывода, однонаправленные (последовательные) итераторы, двунаправленные итераторы и итераторы произвольного доступа.

Тип итератора определяет набор допустимых для него операций. Для всех типов итераторов определены операции ++ и *. Операция ++ имеет одинаковую семантику. Операция * для итераторов ввода возвращает значение элементов коллекции, не допуская их изменение. Для итератора вывода операция * предназначена для изменения элементов коллекции, не допуская их прочтение. Для остальных типов итераторов операция * позволяет выполнять оба действия над элементами коллекции. Двунаправленные и произвольного доступа дополнительно имеют операцию – –. Итераторы произвольного доступа, кроме того, имеют операцию смещения на произвольное количество элементов. Таким образом, получается каждый итератор, имеющий больший набор операций может использоваться вместо более ограниченного в операциях итератора. Алгоритмы библиотеки STL поддерживают принцип наименьших требований.

Важной причиной включения итераторов ввода/вывода в STL является возможность выделения алгоритмов, которые могут использоваться с итераторами, связанными с потоками ввода-вывода. Такие итераторы предоставляются классами STL `istream_iterator` (для ввода из контейнера) и `ostream_iterator` (для вывода).

Итераторы ввода предназначены для получения информации из последовательности в алгоритм, при этом значение в последовательности не может быть изменено. Под последовательностью в данном случае будем понимать или контейнер или поток ввода. Итераторы ввода поддерживают операцию ++ и используются в однопроходных алгоритмах.

☺ Для входных итераторов из `a == b` не следует `++a == + +b`. Алгоритмы, работающие с входными итераторами, не должны пытаться скопировать значение итератора и использовать его для повторного прохода по одной и той же позиции. Тип значения, возвращаемого итератором, не обязан быть ссылочным, поскольку алгоритмы, работающие со входными итераторами, не должны пытаться посредством них выполнять присваивания.

Их использование можно продемонстрировать на алгоритме `find`, для которого достаточно итератора ввода.

Пример 106. Выполнить поиск первого вхождения заданного значения в массиве, списке и потоке ввода `istream`. Вывести для каждой коллекции или элемент, следующий за найденным, или сообщение, что найденный — последний, или сообщение, что элемент не найден.

```
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main(){
  // Инициализация массива 10 целыми числами:
  int a[10] = { 12, 3, 25, 7, 11, 213, 7, 123, 29, -3 };
  // Поиск в массиве первого элемента, равного 7:
  int* ptr = find(a, a+10, 7);
  if (ptr< a+10 )
    if ((++ptr) < a+10)
      cout << *(ptr) << endl;
    else
      cout << "Last found\n";
  else
    cout << "Not found\n";
```

```
  // Инициализация списка числами из массива,начиная с a[4]:
  list<int> listl(a+4, a+10);
  // Поиск в списке первого элемента, равного 7:
  auto pi = find(listl.begin(), listl.end(), 7);
  if (pi != listl.end())
    if ((++pi) != listl.end()) cout << *(pi) << endl;
    else cout << "Last found\n";
  else cout << "Not found\n";

  // Поиск первого символа во входном потоке за символом 7
  istream_iterator<char> in(cin);
  istream_iterator<char> eos;
  in = find(in, eos, '7');
  if ((in) != eos)
    if ((++in) != eos)
      cout << *(in) << endl;
    else cout << "Last found\n";
  else cout << "Not found\n";
  int c;
}
```

Алгоритм `find` возвращает итератор на найденный элемент или признак конца контейнера. В первой части программы алгоритм `find` применяется с обычными указателями C++. Во второй части `find` используется с итераторами списка, которые также удовлетворяют требованиям алгоритма `find`. В последней части программы используются специальные входные итераторы `istream` для чтения значений из входного потока.

Для поиска символа во входном потоке используются итераторы ввода `in` и `eos` класса `istream_iterator`. Объекты `istream_iterator` могут читать данные только в одном направлении, запись данных при помощи этих итераторов невозможна.

```
istream_iterator<char> in(cin);
istream_iterator<char> eos;
```

Конструктор `istream_iterator<T> (istream&)` создаёт входной итератор для значений типа `T` из входного потока, такого как стандартный поток ввода `cin` в приведённом примере.

Конструктор `istream_iterator<T> ()` генерирует входной итератор, который работает в качестве маркера конца для итераторов `istream`. Это просто значение, которому итераторы `istream` становятся равны, когда связанный с ними входной поток сообщает о достижении конца потока.

В отличие от файлов, входной поток `cin` не имеет явного признака конца потока. Для сигнализации об этом состоянии необходимо завершить ввод (`ENTER`) и ввести комбинацию `Ctrl+Z`. Удобнее выполнять запуск из окна командной строки (рис. 9.1).

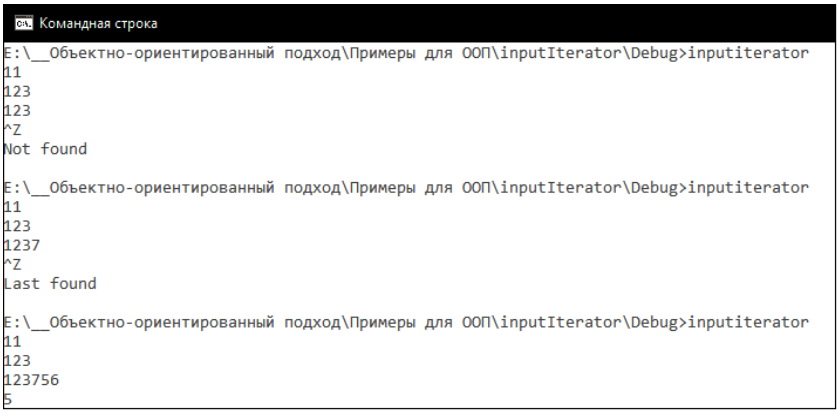

Рис. 9.1. Окно вывода примера 106

В C++11 синтаксис функций получения итераторов на начало и конец коллекции изменился. Вместо член-функции класса коллекции `begin()` рекомендуется использовать внешнюю функцию `begin()`, в которую коллекция передаётся в качестве параметра. Тогда строка

```
auto pi = find(list1.begin(), list1.end(), 7);
```

изменится на

```
auto pi = find(begin(list1),end(list1), 7);
```

Выходные итераторы (итераторы вывода) обладают противоположной входным итераторам функциональностью: они

позволяют записывать значения в последовательность. Под последовательностью понимается либо контейнер, либо выходной поток. Как и в случае итераторов ввода, алгоритмы, использующие итераторы вывода, должны быть однопроходными. Операции равенства и неравенства для таких итераторов могут быть не определены.

Алгоритмы, использующие итераторы вывода, могут работать с выходными потоками для размещения данных посредством класса `ostream_iterator`, а также с итераторами и указателями вставки. Единственное корректное применение оператора `operator*` к итераторам вывода — в левой части операции присваивания.

В примере 102 уже использовался выходной итератор в алгоритме `copy` для вывода символов в поток `cout`.

```
copy(text.begin(),text.end(),ostream_iterator<char>(cout,""));
```

В конструкторе `ostream_iterator<char>` первым параметром указывается поток, в который происходит вывод (`cout`), вторым параметром задаётся разделитель между выводимыми элементами. В примере 102 символы выводятся без разделителей, поэтому используется пустая строка.

Однонаправленный итератор — это итератор, который объединяет свойства входного и выходного итераторов, тем самым обеспечивая возможность чтения и записи. При этом обход элементов контейнера происходит в одном направлении. Однонаправленные итераторы обладают также возможностью сохранить значение итератора и использовать сохранённое значение для повторного прохода из той же позиции. Это позволяет использовать однонаправленные итераторы в многопроходных алгоритмах.

Контейнер `forward_list` имеет однонаправленный итератор `forward_list<T>::iterator`. Последовательные контейнеры `vector`,

list, deque, а также массивы, имеют двунаправленные итераторы, которые обладают свойствами однонаправленных. Поэтому к ним тоже применимы алгоритмы, использующие однонаправленные итераторы.

Одним из простых примеров таких алгоритмов является replace, который заменяет в диапазоне все элементы с заданным значением другим значением.

```
int a[10];
for (int i=0; i<10; ++i)
  cin >> a[i];
// Замена всех элементов массива, равных 5, на 6:
replace(a, a+10, 5, 6);
```

Поскольку тип deque<T>::iterator также удовлетворяет всем требованиям к однонаправленным итераторам, этот алгоритм можно использовать и для замены элементов в деке:

```
deque<char> deq;
char c;
for (int i=0; i<10;++i) {
  cin>>c;
  deq.push_back(c);
  deq.push_front(c);
}
// Замена всех 'e' в deq на 'o':
replace(deq.begin(), deq.end(), 'e', 'o');
```

Двунаправленный итератор аналогичен однонаправленному, кроме того, он допускает обход в обоих направлениях. То есть, двунаправленные итераторы должны поддерживать все операции однонаправленных итераторов, а кроме них — операцию --, делая возможным обход последовательности в обратном направлении. И префиксный, и постфиксный operator-- должны выполняться за константное время.

Возможность обхода структуры данных в обратном порядке важна потому, что без неё некоторые алгоритмы не могут эффективно работать. Например, алгоритм STL reverse может использоваться для обращения

порядка элементов в последовательности только при наличии двунаправленных итераторов.

Контейнер list<T> предоставляет двунаправленные итераторы, так что его можно использовать с алгоритмом reverse:

```
list<int> listl;
//  Код для вставки значений в listl
//  Обращение порядка значений в списке:
reverse(listl.begin(), listl.end());
```

Возможность предоставления контейнером двунаправленного итератора зависит от внутренней организации контейнера. Контейнер list<T> реализован как двусвязный список, поэтому предоставляет возможность итератору реализовать операцию operator-- за константное время. В случае односвязного списка эффективно реализовать operator-- (с константным временем работы) невозможно.

Имеются некоторые алгоритмы, для эффективной работы которых требуется, чтобы любой элемент последовательности был достижим из любого другого за константное время.

Например, обобщённый алгоритм бинарного поиска:

```
binary_search(first, last, value);
```

Итераторы first и last задают диапазон элементов контейнера, упорядоченного по возрастанию. Алгоритм возвращает true, если в последовательности имеется значение value, и false в противном случае.

Для эффективной работы таких алгоритмов необходимы итераторы произвольного доступа. Итераторы произвольного доступа должны поддерживать все операции двунаправленных итераторов, плюс нижеперечисленные (через r и s обозначены итераторы с произвольным доступом, а через n — целочисленное выражение).

- ✓ Прибавление и вычитание целых чисел: r+n, n+r, r-n, r+=n и r-=n.

- ✓ Доступ к n-му элементу при помощи выражения `r[n]`, которое означает `*(r+n)`.
- ✓ Вычитание итераторов `r-s`.
- ✓ Сравнение итераторов `r<s`, `r>s`, `r<=s` и `r>=s`.

Алгоритм `binary_search` на таких контейнерах как векторы, деки и массивы, имеет наибольшую эффективность — время работы O(log N), т. к. их итераторы являются итераторами произвольного доступа. Однако требование алгоритма — наличие однонаправленного итератора, позволяет использовать его и для других контейнеров, например списков. Но время работы увеличится, т. к. имея указатели на начало и конец списка, единственный способ добраться до элемента посредине — это обойти элементы один за другим. Это займёт время, пропорциональное длине списка.

9.6. Адаптеры итераторов

Адаптеры — шаблонные классы, которые обеспечивают изменение поведения при сохранении интерфейса.

Например, для двунаправленных итераторов и итераторов произвольного доступа существуют адаптеры — обратные (реверсивные) итераторы — `reverse_iterator`. Они переопределяют операции увеличения (уменьшения) таким образом, что перемещение происходит в противоположном направлении.

Во всех контейнерах для инициализации реверсивных итераторов используются операции `rbegin()` и `rend()`.

В примере 100 используется обратный итератор для проверки контейнера на симметричность.

```
auto itb = v.begin();
auto itrb = v.rbegin();
while (b && itb < itrb.base()) {
```

```
  b = (*itb) == (*itrb);
  itb++;
  itrb++;
}
```

При этом демонстрируется, что нельзя сравнивать значения обычного и реверсивного итераторов. Для преобразования реверсивного итератора в обычный используется метод `base()`. С одной стороны, это может упрощать алгоритм, но, с другой стороны, следует постоянно помнить про необходимость корректного преобразования. Кроме того, инвертирование семантики операторов инкремента и декремента может внести путаницу, но зато позволяет программисту передавать алгоритмам пару обратных итераторов вместо прямых, упрощая разработку кода. Например, чтобы изменить порядок сортировки вектора с возрастания на убывание, можно вместо написания своего компаратора использовать реверсивные итераторы.

```
// сортирует вектор в порядке возрастания
sort( v.begin(), v.end() );
// сортирует вектор в порядке убывания
sort( v.rbegin(), v.rend() );
```

Другим примером адаптеров итераторов являются итераторы вставки. Главное отличие этих итераторов состоит в том, что выражение `*it = value` трактуется как вставка в контейнер нового элемента со значением `value`. Позиция вставки при этом определяется типом итератора `it`. Существует три типа итераторов вставки, параметризованные типом контейнера:

- ✓ `insert_iterator<Container>` — использует функцию `insert()` (вставка по позиции итератора) класса `Container`;
- ✓ `back_insert_iterator<Container>` — использует функцию `push_back()` (вставка в конец контейнера);
- ✓ `front_insert_iterator<Container>` — использует функцию `push_front()` (вставка в начало контейнера).

Для итераторов вставки также определена операция ++, которая при этом не имеет никакого эффекта. Для итераторов ставки позиция итератора всегда определяется его типом. Поэтому для back_insert_iterator и front_insert_iterator позиция итератора всегда в конце или в начале контейнера. Итератор insert_iterator автоматически сдвигается на одну позицию вперёд после каждой вставки элемента, выполняемой посредством операции *.

Объявление итераторов вставки достаточно сложное. Например, если v — вектор целых чисел, то объявление для него итератора вставки перед вторым элементом будет выглядеть следующим образом.

```
insert_iterator<vector <int> > it(v,next(v.begin()));
```

Чтобы было удобнее использовать вводятся шаблоны функций, возвращающие нужные итераторы (inserter, back_inserter и front_inserter соответственно). Так для нашего примера объявление будет выглядеть следующим образом

```
auto it = inserter(v, next(v.begin()));
```

Итераторы вставки особенно полезны, когда необходимо передать заранее неизвестное количество элементов из одного контейнера или потока в другой.

Пример 107. Создать пустой вектор и добавить в него элементы, вначале используя вставку в конец, а затем вставляя новые значения между первым и вторым элементом. Сформировать из полученного вектора список, содержащий элементы вектора в обратном порядке.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <list>

using namespace std;
int main() {
  vector <int> v;
```

```
  int n;
  cin >> n;
  auto it1 = back_inserter(v);
  for (int i = 0; i < n; ++i) {
    *it1 = i;
  }
  auto it = inserter(v, next(v.begin()));
  for (int i = 0; i > -n; --i) {
    *it = i;
  }
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
  list <int> L1;
  copy(v.begin(), v.end(), front_inserter(L1));
  copy(L1.begin(), L1.end(), ostream_iterator<int>(cout," "));
  return 0;
}
```

В результате работы, при введённом значении n=5, вектор v содержит элементы в таком порядке {0, 0, –1, –2, –3, –4, 1, 2, 3, 4}, а список L1 содержит элементы вектора v в обратном порядке.

Итератор insert_iterator использует функцию-член контейнера insert, которая имеется у всех типов STL-контейнеров. Поэтому функцию inserter, возвращающую insert_iterator, можно применять к любому контейнеру. Это наиболее общий итератор вставки.

Итератор back_insert_iterator и функция back_inserter могут использоваться с контейнерами vector, list и deque, так как они имеют функцию push_back.

Для итератора front_insert_iterator и функции front_inserter доступно использование для контейнеров list и deque.

Семантика перемещения, появившаяся в стандарте C++11, нашла своё отражение и в итераторах. Появился ещё один адаптер — move_iterator. Он предоставляет такое же поведение, что и инкапсулируемый в нем базовый итератор, за исключением того, что операция разыменования возвращает ссылку rvalue (&&), поэтому операция копирования заменяется перемещением. Основное назначение перемещающего итератора —

преобразование одного контейнера в другой не используя выделение и освобождение памяти.

Возможная реализация операции разыменования для перемещающего итератора представлена ниже.

```
value_type&& operator*() const{
  return move(*current);
}
```

Как и для итератора вставки есть шаблон фабричных функций для создания перемещающего итератора `make_move_iterator`.

Пример 108. Проиллюстрировать различие между использованием в конструкторе копии в качестве параметров обычных итераторов и перемещающих итераторов.

```
#include <list>
#include <vector>
#include <string>
#include <iterator>
using namespace std;
int main(){
  list<string> s{"one", "two", "three"};
  vector<string> v1(s.begin(), s.end()); // copy
  cout << endl << "vector v1" << endl;
  copy(v1.begin(),v1.end(),ostream_iterator<string>(cout," "));
  cout << endl<<"list s after copy"<<endl;
  copy(s.begin(),s.end(),ostream_iterator<string>(cout," "));
  cout << endl;
  vector<string> v2(make_move_iterator(s.begin()),
                    make_move_iterator(s.end())); // move
  cout << endl << "vector v2" << endl;
  copy(v2.begin(),v2.end(),ostream_iterator<string>(cout," "));
  cout << endl << "list s after move" << endl;
  copy(s.begin(),s.end(),ostream_iterator<string>(cout," "));
  cout << endl;
  return 0;
}
```

В окне вывода (рис.9.2) можно заметить, что после создания вектора `v1` посредством конструктора копии с обычным итератором список `s` сохраняется в памяти. Применение же в конструкторе копии в качестве параметров перемещающих итераторов приводит к тому, что список `s` оказывается пуст.

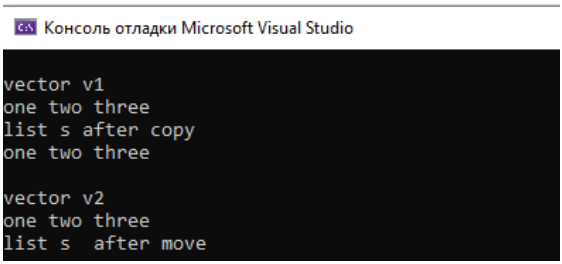

Рис.9.2. Окно вывода примера 108

Важно отметить наличие в языке C++ большого количества видов итераторов. При этом каждый алгоритм предъявляет минимальные требования к используемому виду итераторов. Чтобы не потеряться в большом количестве видов нужно использовать документацию [19], обращая внимание на мнемоническое обозначение допустимых видов итераторов.

Например, чаще всего итератор ввода обозначается `InputIterator`, итераторы произвольного доступа — `RandomAccessIterator` и т.п.

9.7. Категории алгоритмов

STL предоставляет программистам большой выбор алгоритмов, работающих со структурами данных, определенных в рамках схемы STL. Основная цель заключается в том, чтобы сделать алгоритмы STL не зависящими от структур данных, с которыми они работают. Вместо этого алгоритмы и контейнеры STL комбинируются следующим образом.

- ✓ Для каждого контейнера мы указываем, какие категории итераторов он предоставляет.
- ✓ Для каждого алгоритма мы указываем, с какими категориями итераторов он работает.

Определение алгоритмов в терминах итераторов, а не непосредственно структур данных, делает возможной независимость

алгоритмов от последних. Так алгоритмы становятся особенно «обобщёнными», способными работать с большим количеством разных структур данных. Поэтому алгоритмы могут работать с определяемыми пользователем структурами данных, если они имеют итераторы нужного вида.

Алгоритмы представляют собой внешние функции. Большинство алгоритмов определено в заголовочном файле algorithm. Некоторые численные алгоритмы определены в заголовочном файле numeric. Так же как и в случае с итераторами рекомендуется подробно знакомиться с алгоритмами, работая с документацией [19].

Алгоритмы можно разбить на следующие группы:

- ✓ алгоритмы, не модифицирующие последовательность;
- ✓ алгоритмы, модифицирующие последовательность;
- ✓ алгоритмы сортировки, поиска, слияния, а также алгоритмы, выполняемые над множествами;
- ✓ числовые алгоритмы.

Вне зависимости от того, к какой категории относится алгоритм, он может иметь несколько вариаций. Можно выделить версии, вносящие изменения в исходный контейнер, и копирующие версии. В названиях последних обязательно будет присутствовать суффикс `_copy`. Алгоритмы, которые могут принимать в качестве параметра указатель на логическую функцию (предикат) имеют в названиях суффиксы `_if` или `_if_not`.

Пример 109. Проиллюстрировать различие между использованием версий модифицирующего алгоритма замены элементов контейнера (`replace`, `replace_copy`, `replace_if`, `replace_copy_if`).

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

```
bool raise(int v) {return v >= 85;}

void print_vector(vector<int> v) {
  copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
  cout << endl;
}

int main() {
  int myints[] = { 100, 80, 90, 95, 60, 70, 85, 75 };

  vector<int> myvector1(myints, myints + 8);
  replace(myvector1.begin(), myvector1.end(), 95, 100);
  print_vector(myvector1);
  vector<int> myvector2(myints, myints + 8);
  vector<int>  myvector3(8);
  replace_copy(myvector2.begin(), myvector2.end(),
               myvector3.begin(), 95, 100);
  print_vector(myvector2);
  print_vector(myvector3);

  vector<int> myvector4(myints, myints + 8);
  replace_if(myvector4.begin(), myvector4.end(), raise, 100);
  print_vector(myvector4);
  vector<int> myvector5(myints, myints + 8);
  vector<int>  myvector6(8);
  replace_copy_if(myvector5.begin(), myvector5.end(),
                  myvector6.begin(), raise, 100);
  print_vector(myvector5);
  print_vector(myvector6);
  return 0;
}
```

9.8. Алгоритмы с функциональными параметрами

В примере 109 было продемонстрировано использование функциональных параметров для задания критерия выбора элементов контейнера, т. е. предикат. Функциональные параметры могут быть не только логического типа. Функциональные параметры могут быть унарными и бинарными. В C++ в качестве функционального параметра может выступать как обычная функция, так и объект класса (или структуры), перегружающей операцию вызова функции `operator()`. Объекты, для которых перегружена операция вызова функции, называются функторами.

Например, для функции без параметров, не возвращающей результат, тип функционального объекта должен быть определён одним из двух следующих способов:

```
class FunctionObjectType {
  public:
  void operator() () {
    // Do some work
  }
};
```

```
struct FunctionObjectType {
  void operator() () {
    // Do some work
  }
};
```

В этом случае в следующем коде выражение `func()` является вызовом `operator()` функционального объекта `func`, а не вызовом некоторой функции с именем `func`.

```
FunctionObjectType func;
func();
```

В общем случае вид перегружаемого оператора вызова функции может быть любым: с разными списками параметров и типами возвращаемого значения.

Поскольку функтор определяется через класс, а в определении класса могут быть поля и конструкторы, то функциональный объект может иметь состояние, в отличие от функции. Более того, можно создать несколько функциональных объектов одного типа, каждый со своим состоянием.

```
class Hello {
  private:
    string name;
  public:
  Hello (string s="incognito"):name(s){}
  void operator() () {
    cout<<"Hello, "<<name<<endl;
  }
};

Hello p1;
Hello p2("C++");
Hello p3("student");
p1();
p2();
p3();
```

В ряде случаев может потребоваться определить функтор, реализующий стандартную встроенную операцию. Такие функторы (табл. 10) уже включены в стандартную библиотеку. Они объявлены в заголовочном файле <functional>.

Таблица 10

Встроенные функторы

Тип операции	Имя функтора	Арность функтора	Тип результата	Реализуемая операция
Сравнения	equal_to	Бинарный	bool	x == y
	not_equal_to	Бинарный	bool	x != y
	greater	Бинарный	bool	x > y
	less	Бинарный	bool	x < y
	greater_equal	Бинарный	bool	x >= y
	less_equal	Бинарный	bool	x <= y
Логические	logical_and	Бинарный	bool	x && y
	logical_or	Бинарный	bool	x || y
	logical_not	Унарный	bool	!x
Арифметические	plus	Бинарный	T	x + y
	minus	Бинарный	T	x - y
	multiplies	Бинарный	T	x * y
	divides	Бинарный	T	x / y
	modulus	Бинарный	T	x % y
	negate	Унарный	T	-x
Битовые (C++11)	bit_and	Бинарный	T	x & y
	bit_or	Бинарный	T	x | y
	bit_xor	Бинарный	T	x ^ y
	bit_not	Унарный	T	~x

К встроенным функторам относятся базовые арифметические операции (+-*/%), базовые логические операции (&&, ||, !) и операции сравнения (==, !=, >, <, >=, <=). Начиная с C++11, также были добавлены некоторые битовые операции (and, or, xor, not). Наличие встроенных функторов позволяет пользователю не писать собственные аналоги. Их можно использовать со встроенными типами и с любым пользовательским типом, в котором перегружены соответствующие операции.

Большинство обобщённых алгоритмов STL принимает в качестве параметра функциональный объект. Алгоритмы поиска `find`, удаления `remove`, замены `replace`, копирования `copy` и т. п. в версиях с суффиксом `if` используют функциональный параметр — одноместный предикат. Алгоритмы сортировки, слияния, а также алгоритмы, выполняемые над множествами (`sort`, `partition` и др.) используют компараторы (двуместные предикаты). Алгоритм `transform` имеет две перегруженных версии, одна из которых использует унарную операцию преобразования элементов, а вторая — бинарную.

Для ряда преобразований удобно использовать алгоритм `transform` в сочетании со стандартными функторами. Например, следующий вызов позволяет поменять знаки всех элементов контейнера `v1` на противоположные, используя стандартный функтор `negate<int>()`.

```
transform(begin(v1), end(v1), begin(v1), negate<int>());
```

Используя функтор `plus<int>()` можно просуммировать элементы двух контейнеров `v1` и `v2` попарно, помещая сумму в третий контейнер `v3`.

```
transform(begin(v1),end(v1),begin(v2),begin(v3),plus<int>());
```

К алгоритмам с функциональными параметрами относится и `for_each`. Он интересен тем, что его, в зависимости от функционального параметра, можно отнести к разным категориям алгоритмов: немодифицирующим и модифицирующим элементы контейнера. Если

функциональный параметр не меняет значение элемента контейнера, for_each можно отнести к категории немодифицирующих алгоритмов. В документации for_each относится именно к категории немодифицирующих алгоритмов. Но ограничений на использование вида функционального параметра нет, что позволяет использовать for_each и как модифицирующий алгоритм.

Возможная реализация алгоритма for_each, представленная в документации [19], выглядит следующим образом.

```
template<class InputIter, class Function>
Function for_each(InputIter first, InputIter last, Function fn){
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;      // or, since C++11: return move(fn);
}
```

Тип Function определяет функцию с одним параметром, тип которого должен совпадать с типом элементов контейнера. Но передаваться он может как по значению, так и по ссылке.

Пример 110. Дан список целых чисел. Реализовать вывод элементов списка и их квадратов в виде таблицы. Затем увеличить каждый элемент в два раза и ещё раз вывести в виде таблицы.

```
#include <iostream>
#include <list>
#include <algorithm>
#include <iomanip>

using namespace std;

// Функция печати строки таблицы
void print(int i) {
  cout << setw(5)<< i << setw(5) << i*i<<endl;
}

void twice (int & x){
  x *= 2;
}
```

```
int main() {
  list<int> v{ 1, 3, 5 };
  for_each(begin(v), end(v), print);
  for_each(begin(v), end(v), twice);
  for_each(begin(v), end(v), print);
  return 0;
}
```

Данный пример иллюстрирует оба варианта использования `for_each`, принадлежащего как категории немодифицирующих алгоритмов, так и категории модифицирующих.

Если вместо функций использовать функторы, то код будет выглядеть следующим образом.

```
struct print {
  void operator()(int i) {
    cout << setw(5) << i << setw(5) << i * i << endl;
    }
};
struct twice {
  void operator()(int& x) {
    x *= 2;
  }
};

int main() {
  list<int> v{ 1, 3, 5 };
  print p;
  twice t;
  for_each(begin(v), end(v), p);
  for_each(begin(v), end(v), t);
  for_each(begin(v), end(v), p);
  return 0;
}
```

Тот же код можно записать с использованием лямбда-выражений:

```
int main() {
  list<int> v{ 1, 3, 5 };
  for_each(begin(v),end(v),[](int
i){setw(5)<<i<<setw(5)<<i*i<<endl;});
  for_each(begin(v), end(v), [](int& x){x *= 2;})
  for_each(begin(v),end(v),[](int
i){setw(5)<<i<<setw(5)<<i*i<<endl;});
  return 0;
}
```

9.9. Лямбда-выражения

Лямбда-выражения пришли из функциональных языков, впоследствии они появились и в императивных языках, в том числе и в C++. Они являются одним из важных нововведений, появившихся в C++11.

Лямбда-выражениями называются безымянные (анонимные) локальные функции, которые можно создавать прямо внутри какого-либо выражения. Поэтому их также называют лямбда-функциями. Лямбда-выражения в C++ являются синтаксическим сахаром. Компилятор автоматически преобразует их в анонимные функторы.

В общем случае синтаксис лямбда-выражения выглядит следующим образом:

[список захвата] (список параметров)→тип возвращаемого значения {код;}

Не все элементы объявления обязательны. В примере 110 используется простой вариант. Список захвата пуст. Тип возвращаемого значения не указан, поскольку тип результата выводится компилятором из кода. В обоих случаях в примере 110 тип результата — `void`.

```
[](int i){setw(5)<<i<<setw(5)<<i*i<<endl;}
[](int& x){x *= 2;}
```

Пример 111. Дан вектор вещественных чисел. Заменить отрицательные числа нулём.

```
int main(){
  list<double> v{ 1, -3, 5, -7, 0 };
  copy(begin(v),end(v), ostream_iterator<double>(cout, " "));
  cout << endl;
  transform(begin(v),end(v),begin(v),[](double d) {return d<0? 0:d;});
  copy(begin(v), end(v), ostream_iterator<double>(cout," "));
  cout << endl;
  return 0;
}
```

В простейших случаях тип возвращаемого значения лямбда-функции выводится компилятором. Однако в лямбда-выражениях могут возникнуть неоднозначности, когда возвращаемый тип не может быть выведен компилятором, например, непонятно, какой тип будет выведен в случае возвращения нуля — int или double:

```
transform(begin(v), end(v), begin(v),[](double d) {
      if (d < 0.0001) {return 0;}
      else {return d;}
    }
);
```

Чтобы решить эту проблему, нужно указать возвращаемый тип лямбда-выражения.

```
transform(begin(v), end(v), begin(v),[](double d) -> double {
      if (d < 0.0001) {return 0;}
      else {return d;}
    }
);
```

Пример 112. Даны список целых чисел и целое число — множитель. Умножить каждый элемент списка на заданный множитель.

При умножении на конкретное заранее известное число, например 2, код выглядит следующим образом.

```
for_each(begin(v), end(v), [](int& x){x *= 2;})
```

Но теперь нам нужно лямбда-выражение, в котором элементы контейнера будут умножаться на произвольное число — заданный множитель. Для этого необходимо использовать захват переменных из внешнего контекста, т. е. необходимо указать в списке захвата, что множитель пришёл в функцию извне. Изменим код следующим образом:

```
int a = 5; // множитель
for_each(begin(v), end(v), [a](int& x){x *= a;})
```

Рассмотрим правила синтаксиса захвата переменных. Можно указывать несколько захватываемых переменных через запятую. Можно захватывать ссылку на переменную. Помимо имён захватываемых переменных, можно указывать «правила захвата», обозначаемые «&» и «=»:

- ✓ [&x] — захват ссылки на переменную;
- ✓ [x, &y] — захват переменной x по значению, переменной y — по ссылке;
- ✓ [=] — захват всех переменных происходит по значению;
- ✓ [&] —захват всех переменных происходит по ссылке;
- ✓ [&, x] — по умолчанию захват переменных происходит по ссылке, а переменная x будет захвачена по значению;
- ✓ [=, &x] — по умолчанию захват переменных происходит по значению, а для переменной x захват производится по ссылке.

Таким образом лямбда-функция может использовать как собственные параметры, так и переменные, захваченные извне. Такие захваченные переменные называют доступными в контексте функции. То есть переменные, доступные в том же контексте, где и описана лямбда-функции, доступны внутри лямбда-функции. Такая функция называется замыканием.

> 🕮 *Замыкание* (*closure*) в программировании — функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции и не в качестве её параметров (а в окружающем коде). Говоря другим языком, замыкание — функция, которая ссылается на свободные переменные в своём контексте.

Напомним, что лямбда-функция компилятором превращается в функтор — класс, перегружающий операцию вызова функции, не имеющий полей, с конструктором без параметров, определяемый по умолчанию.

Операция вызова функции имеет доступ к параметрам и переменным из области видимости, определяемой этим классом. Механизм замыкания позволяет добавить в эту область видимости захваченные переменные.

Пример 113. Дан вектор целых чисел. Используя алгоритм `for_each` найти сумму элементов вектора.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
  vector<int> v { 1, 2, 3, 4, 5 };
  int sum = 0;
  for_each(begin(v), end(v), [&sum](int x) {sum += x; });
  cout << sum;
}
```

В списке захвата появилась ссылка, поэтому изменение захваченной переменной `sum` в функции будет сохранено после выхода из функции.

Лямбда-выражения, как и другие функции, можно присваивать переменным с целью их повторного использования. В этом случае тип переменной определяется через шаблонный класс `std::function` (начиная с C++11), который является полиморфной обёрткой для функций с разными сигнатурами.

Объекты класса `std::function` могут хранить, копировать и вызывать произвольные вызываемые объекты — функции, лямбда-выражения и другие функциональные объекты. В любом месте, где необходимо использовать указатель на функцию для её отложенного вызова, вместо него может быть использован `std::function`. Соответствующий класс определён в заголовочном файле `<functional>`.

Например, лямбда-выражению, возвращающему квадрат целого числа, будет соответствовать следующий тип переменной

```
std::function<int(int)> g = [](int x){return x*x;};
```

А воспользоваться этой переменной можно следующим образом: выполнив одиночный вызов или передав лямбду в алгоритм, например, `transform`.

```
cout << g(5);
vector<int> v {1, 2, 3, 4, 5};
transform(begin(v), end(v), begin(v), g);
```

Для упрощения определения переменной можно воспользоваться ключевым словом `auto`.

```
auto g = [](int x){return x*x;}
```

Для переменных `auto` указывает, что тип переменной `g` будет автоматически выведен из её инициализатора.

9.10. Обобщённые численные алгоритмы

В STL выделена группа обобщённых численных алгоритмов. Для них необходимо подключить заголовочный файл <numeric>. Рассмотрим часто используемые четыре алгоритма: `accumulate`, `partial_sum`, `adjacent_difference` и `inner_product`. Каждый из них имеет по два шаблона. Один из шаблонов предполагает использование в алгоритме предопределённых для этого алгоритма операций. Второй шаблон позволяет настроить алгоритм, заменив предопределённую операцию на другой функтор.

Алгоритм `accumulate` суммирует все значения из диапазона.

Первый шаблон

```
template <typename InIter, typename T>
T accumulate (InIter first, InIter last, T init);
```

Второй шаблон

```
template <typename InIter, typename T, typename BinOp>
T accumulate (InIter first, InIter last, T init, BinOp oper);
```

В процессе аккумуляции в первом шаблоне используется операция + (сложения). Вторая версия позволяет использовать вместо сложения любую другую бинарную операцию. Параметр `init` позволяет указывать любое начальное значение для суммы или любой другой используемой бинарной операции.

Алгоритм `partial_sum` вычисляет все частичные суммы последовательности значений из диапазона.

Первый шаблон

```
template <typename InIter, typename OutIter>
OutIter partial_sum (InIter first, InIter last, OutIter res);
```

Второй шаблон

```
template <typename InIter, typename OutIter, typename BinOp>
OutIter partial_sum (InIter first, InIter last, OutIter res, BinOp oper);
```

Последовательность вычисленных сумм сохраняется в контейнере, задаваемом итератором. Вторая версия позволяет использовать вместо сложения любую другую бинарную операцию.

Алгоритм `adjacent_difference` вычисляет разности между соседними значениями элементов (для каждого элемента, кроме первого, вычисляется разность между текущим и предыдущим; первый возвращается без изменений).

Первый шаблон

```
template <typename InIter, typename OutIter>
OutIter adjacent_difference (InIter first, InIter last, OutIter res);
```

Второй шаблон

```
template <typename InIter, typename OutIter, typename BinOp>
OutIter adjacent_difference (InIter first, InIter last, OutIter res,
        BinOp oper);
```

Последовательность вычисленных разностей сохраняется в контейнере, задаваемом итератором. Вторая версия позволяет использовать вместо вычитания любую другую бинарную операцию.

Алгоритм `inner_product` вычисляет суммы попарных произведений элементов из двух диапазонов.

Первый шаблон

```
template <typename InIter1, typename InIter2,typename T>
T inner_product (InIter1 first1, InIter1 last, InIter2 first2, T init);
```

Второй шаблон

```
template    <typename    InIter1,    typename    InIter2,    typename    T,
            typename BinOp1, typename BinOp2>
T inner_product (InIter1 first, InIter1 last, InIter2 first2, T init,
                BinOp1 oper1, BinOp2 oper2);
```

Последовательность вычисленных разностей сохраняется в контейнере, задаваемом итератором. Вторая версия позволяет использовать вместо сложения любую другую бинарную операцию `BinOp1 oper1`, а вместо умножения `BinOp2 oper2`.

Пример 114. В примере демонстрируется использование рассмотренных обобщённых численных алгоритмов. Для каждого из алгоритмов демонстрируется использование как шаблона с операцией по умолчанию, так и шаблона с настраиваемыми операциями.

```
#include <iostream>
#include <vector>
#include <numeric>
#include <functional>
#include<iterator>
using namespace std;

int main() {
  vector <int> val = { 10,20,30 };
```

```
//Накопление суммы всех элементов вектора val
  cout << accumulate(begin(val), end(val), 0) << endl;

//Использование accumulate с операцией minus и начальным значением 100
  cout << accumulate(begin(val), end(val), 100, minus<int>())<<endl;

//Использование accumulate с операцией «минус по условию», реализованной
//через лямбда-функцию
  cout << accumulate(begin(val), end(val), 0, [](int x, int y) {
    if (y % 3 == 0) return x - y; return x; }) << endl;

// Накопление суммы квадратов элементов вектора val
  cout << accumulate(begin(val), end(val), 0, [](int x, int y) {
    return x + y*y;}) << endl;

  vector<int> result(3);
//Вычисление частичных сумм. Результат сохраняется в новый вектор
  partial_sum(begin(val), end(val), begin(result));
  copy(begin(result), end(result), ostream_iterator<int>(cout, " "));
  cout << endl;

//Вычисление частичных сумм. Результат выдаётся в поток
  partial_sum(begin(val), end(val), ostream_iterator<int>(cout, " "));
  cout << endl;

//Вычисление частичных произведений. Результат выдаётся в поток
  partial_sum(begin(val), end(val), ostream_iterator<int>(cout, " "),
              multiplies<int>());
  cout << endl;

//Вычисление разности между соседними элементами
//Результат сохраняется в новый вектор
  adjacent_difference(begin(val), end(val), begin(result));
```

```
  copy(begin(result), end(result), ostream_iterator<int>(cout, " "));
  cout << endl;

//Вычисление попарных произведений соседних элементов
  adjacent_difference(begin(val), end(val), begin(result),
                      multiplies<int>());
  copy(begin(result), end(result), ostream_iterator<int>(cout, " "));
  cout << endl;

//Вычисление остатков от деления очередного элемента на предыдущий
  adjacent_difference(begin(val), end(val), begin(result),
                     [](int x, int y) { return x % y; });
  copy(begin(result), end(result), ostream_iterator<int>(cout, " "));
  cout << endl;
  vector <int> a = { 1,2,3,4 };
  vector <int> b = { 4,3,2,1 };

//Вычисление скалярного произведения двух векторов a и b
  cout << inner_product(begin(a), end(a), begin(b), 0) << endl;

//Вычисление скалярного произведения двух векторов a и a
  cout << inner_product(begin(a), end(a), begin(a), 0) << endl;

//Вычисление произведения попарных сумм двух векторов a и b
  cout << inner_product(begin(a), end(a), begin(b), 1, multiplies<int>(),
                        plus<int>()) << endl;

//Вычисление суммы модулей попарных разности двух векторов a и b
  cout << inner_product(begin(a), end(a), begin(b), 0, plus<int>(),
                       [](int x, int y) {return abs(y - x); }) << endl;

//Последовательно вычисляются модули попарных разностей элементов двух
//векторов a и b.Накопление производится по правилу:
```

```
//результат умножается на нечётный модуль и суммируется с чётным.
  cout << inner_product(begin(a), end(a), begin(b), 1,
         [](int x, int y) {if (y % 2 != 0) return x * y; return x + y; },
         [](int x, int y) {return abs(y - x); }) << endl;

  return 0;
}
```

ГЛАВА 10. УМНЫЕ УКАЗАТЕЛИ

Указатели в C/C ++ с одной стороны дают больше гибкости, а с другой — если их использовать некорректно, то это вызовет различные проблемы в программе, которые трудно обнаруживать.

Начиная с C++11 стандартная библиотека включает в себя так называемые *умные указатели* (*smart pointers*). Эти указатели предназначены для того, чтобы сделать работу с указателями безопасной.

10.1. Семантика умных указателей

Умный указатель (smart pointer) — класс (обычно шаблонный), имитирующий интерфейс обычного указателя и добавляющий некую новую функциональность, например проверку границ при доступе или очистку памяти.

Умный указатель хранит обычный указатель на объект в динамической памяти и освобождает память, занимаемую объектом, в своём деструкторе. Такая технология называется RAII, Resource Acquisition Is Initialization — «Захват ресурса есть инициализация». RAII — это популярный паттерн проектирования, применяемый во многих объектно-ориентированных языках, смысл которого заключается в том, что захват ресурса совмещается с инициализацией объекта, а освобождение — с финализацией (уничтожением) объекта. Впервые в C++ такое поведение было реализовано в шаблоне `auto_ptr` стандартной библиотеки. Начиная с C++11 этот шаблон считается устаревшим.

В новом стандарте появились следующие умные указатели: `unique_ptr`, `shared_ptr` и `weak_ptr`. Все они объявлены в заголовочном файле `<memory>`.

Чаще всего умный указатель инкапсулирует семантику владения ресурсом. В таком случае он называется *владеющим указателем*.

Владеющие указатели применяются для борьбы с утечками памяти и висячими ссылками. Утечкой памяти называется ситуация, когда в программе нет ни одного указателя, хранящего адрес объекта, созданного в динамической памяти. Висячей ссылкой называется указатель, ссылающийся на уже удалённый объект. Семантика владения для динамически созданных объектов означает, что удаление или присвоение нового значения указателю будет согласовано с временем жизни объекта.

Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах. Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты. В случае обычного указателя — уничтожится переменная-указатель, при этом ресурс останется не освобождённым. В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

В каждом из умных указателей `unique_ptr`, `shared_ptr` и `weak_ptr` есть свои особенности, которые и определяют их предназначение.

10.2. Стратегия одного владельца

Умный указатель `unique_ptr` реализует стратегию «одного владельца», запрещая копирование.

Любая попытка использовать явное или неявное копирование приводит к ошибке компиляции.

```
std::unique_ptr<myClass> x_ptr(new myClass(5));
std::unique_ptr<myClass> y_ptr;
y_ptr = x_ptr; // ошибка при компиляции
std::unique_ptr<myClass> z_ptr(x_ptr); // ошибка при компиляции
```

Но передать права владения объектом можно с помощью вспомогательной функции `std::move` (которая является частью механизма

перемещения). При этом права владения переходят к новому владельцу, а старый хранит значение nullptr.

```
std::unique_ptr<myClass> x_ptr(new myClass(5));
std::unique_ptr<myClass> y_ptr;
y_ptr = std::move(x_ptr); //права владения переходят к y_ptr
```

Если по каким-то причинам необходимо вернуться от unique_ptr к обычному (сырому) указателю, следует воспользоваться методами get() и reset(). Метод get() возвращает *сырой* (классический) указатель, а метод reset() сбрасывает права владения объектом. Сбрасывать права необходимо, чтобы и дальше поддерживать идеологию уникального владельца объекта.

```
std::unique_ptr<myClass> ptr = std::unique_ptr<myClass>(new myClass());
myClass *mco = ptr.get(); // получаем классический указатель
mco ->myfunc();
ptr.reset(); // сбрасываем права владения
```

Для создания unique_ptr можно использовать функцию std::make_unique(). Эта шаблонная функция создаёт объект шаблонного типа и инициализирует его аргументами, переданными в эту функцию.

```
//Динамически размещаемая дробь Fraction с числителем 3 и знаменателем 5.
auto f1{ std::make_unique<Fraction>(3, 5) };
//Динамически размещаемый массив дробей Fraction длиной 4
  auto f2{ std::make_unique<Fraction[]>(4) };
```

Для умного указателя в случае управления им динамическим массивом перегружена операция обращения по индексу.

Пример 115. В примере демонстрируется использование умного указателя unique_ptr.

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

```
class DemoSmartPtr {
public:
  string name;
  DemoSmartPtr(string s = "") : name(move(s)) {
    cout << "CTOR " << name << '\n';
  }
  ~DemoSmartPtr() {
    cout<<"DTOR "<<name<< '\n';
  }
};

void process_item(unique_ptr<DemoSmartPtr> p) {
  if (!p)
    return;
  cout<< "Processing "<<p->name<< '\n';
}

int main() {
  cout << " Create objects in block \n"; {
    unique_ptr<DemoSmartPtr> p1{ new DemoSmartPtr("first") };
    auto p2{ make_unique<DemoSmartPtr>("second") };
  }
  cout << " Out of block \n";
  process_item(make_unique<DemoSmartPtr>("3'd"));
  auto p3{ make_unique<DemoSmartPtr>("moved") };
  auto p4{ make_unique<DemoSmartPtr>("last") };
  process_item(move(p3));
  cout << "END OF main \n";*/
  return 0;
}
```

10.3. Стратегия совместного доступа к ресурсу

Умный указатель shared_ptr поддерживает стратегию совместного доступа к ресурсу, которая реализуется через подсчёт ссылок на ресурс. Ресурс освобождается тогда, когда счётчик ссылок на него будет равен 0. Как видно, система реализует одно из основных правил сборщика мусора.

```
std::shared_ptr<myClass> x_ptr(new myClass(42));
std::shared_ptr<myClass> y_ptr(new myClass(13));
y_ptr = x_ptr;
```

После выполнения операции присваивания ресурс myClass(13), на который указывал ранее y_ptr, освободится, поскольку счётчик ссылок на него станет равен 0 и будет вызван деструктор.

А на ресурс myClass(42) будут ссылаться оба указателя, и счётчик ссылок будет равен двум. Ресурс myClass(42) освободится лишь при уничтожении последнего ссылающегося на него указателя.

Также как и класс unique_ptr, данный класс предоставляет методы get() и reset().

Увеличение счётчика может происходить при выполнении операций присваивания указателей, создании нового указателя с помощью конструкции копии, а также при передаче параметров-указателей по значению. Если передать указатель в качестве параметра по ссылке, то счётчик не будет увеличен.

При работе с умным указателем, следует опасаться их создания на лету. Например, следующий код может привести к утечке памяти.

```
someFunction(shared_ptr<myClass>(new myClass(42)), ());
```

Это связано с тем, что стандарт C++ не определяет порядок вычисления аргументов. Может случиться так, что сначала выполнится new myClass(42), затем problemFunc() и лишь затем конструктор shared_ptr. Если же функция problemFunc() бросит исключение, до

конструктора shared_ptr дело не дойдёт, хотя ресурс (объект myClass) был уже выделен. Вызов конструктора new myClass возвращает временный объект. Однако, временным является указатель на выделенный ресурс, и в случае исключения — уничтожится указатель, при этом ресурс останется выделенным.

В случае с shared_ptr есть выход — использовать фабричную функцию make_shared<>, которая создаёт объект заданного типа и возвращает shared_ptr, указывающий на него.

```
someFunction(make_shared<myClass>(42), problemFunc ());
```

Поскольку результатом вызова фабричной функции make_shared является временный объект, то он будет уничтожен в случае возникновения исключений. Стандарт C++ чётко декларирует, что временные объекты уничтожаются в случае появления исключения.

Чтобы понять какой из умных указателей нужно использовать, рекомендуется проанализировать ситуации, в которых он может потребоваться.

Если объект нужен только в одном месте, то следует использовать unique_ptr, чтобы защититься от непреднамеренного копирования. Если объект понадобился в нескольких местах, то — shared_ptr.

Если рассмотреть реализацию умных указателей, то выяснится, что unique_ptr по своей эффективности близок к обычным указателям. В то время как shared_ptr предоставляет больше возможностей, но требует больше накладных расходов (памяти и времени доступа).

Пример 116. В примере демонстрируется использование умного указателя shared_ptr.

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

```
class DemoSmartPtr {
public:
  string name;
  DemoSmartPtr(string s = "") : name(move(s)) {
    cout << "CTOR " << name << '\n';
  }
  ~DemoSmartPtr() {
    cout<<"DTOR "<<name<< '\n';
  }
};

void f(shared_ptr<DemoSmartPtr>sp) {
  cout << "in F: "<<sp->name<<" counter = " << sp.use_count() << '\n';
}

int main() {
  shared_ptr <DemoSmartPtr> sh1;
  cout << " Inner scope begin \n";
  shared_ptr<DemoSmartPtr> sp1{ new DemoSmartPtr("first") };
  auto sp2{ make_shared<DemoSmartPtr>("second") };
  cout << sp1->name<<" counter :" << sp1.use_count() << '\n';
  sh1 = sp1;
  cout << sp1->name<<"  counter :" << sp1.use_count() << '\n';
  f(sh1);
  cout << sp1->name << "  counter :" << sp1.use_count() << '\n';
  cout << " Back to outer scope \n";
  cout <<sh1->name<< " counter " << sh1.use_count() << '\n';
  f(sh1);
  if (!sh1)
    cout << " we no longer own \n";
  else
    cout << sh1->name << " counter " << sh1.use_count() << '\n';
  f(move(sh1));
```

```
  if (!sh1)
    cout << " we no longer own \n";
  else
    cout << sh1->name << " counter " << sh1.use_count() << '\n';
  cout << "END OF main \n";
return 0;
}
```

10.4. Указатель, не владеющий объектом

Представим себе случай, когда нам нужно передать указатель, не передавая права владения объектом. Использовать для этой цели `shared_ptr` нельзя, т. к. его копирование увеличит счётчик ссылок, и в результате происходит разделение прав владения. Использование обычного указателя также нежелательно, т. к., как уже говорилось в самом начале, невозможно проверить, ссылается указатель на существующий объект или нет.

Специально для такого случая предусмотрен ещё один вид умных указателей: `weak_ptr`. Это «слабый», не владеющий экземпляром объекта умный указатель.

Указатель `weak_ptr` ссылается на экземпляр, которым владеет `shared_ptr`, однако не разделяет прав владения этим экземпляром. Это гарантирует, что, если экземпляр уже уничтожен, мы сможем это надёжно и безопасно проверить.

Рассмотрим классический пример циклических или перекрёстных ссылок. Одним из типичных является случай, когда один объект владеет коллекцией других объектов.

```
struct Container {
  shared_ptr<Container> otherContainer;
};
```

```
void func() {
    shared_ptr<Container> a(new Container);
    shared_ptr<Container> b(new Container);
    a->otherContainer = b;
    // В этой точке у второго объекта счетчик ссылок = 2
    b->otherContainer = a;
    // В этой точке у обоих объектов счетчик ссылок = 2
}
```

Что произойдёт при завершении функции и выходе объектов `a` и `b` из области определения? В деструкторе уменьшатся ссылки на объекты. У каждого объекта будет счётчик равен 1 (объект, на который ссылался `a`, всё ещё указывает на объект, на который ссылался `b`, и наоборот). Объекты «держат» друг друга, но нет возможности получить к ним доступ, чтобы их уничтожить.

Аналогичная ситуация может возникнуть при использовании `shared_ptr` для коллекции.

```
struct RootContainer {
    list<shared_ptr<Container> > Containers;
};
struct Container {
    shared_ptr<RootContainer> parent;
};
```

При таком устройстве каждый `Containers` будет препятствовать удалению `RootContainer` и наоборот.

Для решения этой проблемы существует `weak_ptr`. При этом возникает вопрос: «Какой из указатель сделать слабым?». Очевидно, что именно `RootContainer` в данном случае владеет объектами `Container`, а не наоборот.

Поэтому нужно модифицировать `struct Container`, используя указатель, не владеющий экземпляром объекта:

```
struct Container {
  weak_ptr<RootContainer > parent;
};
```

Слабые указатели не препятствуют удалению объекта. Они могут быть преобразованы в сильные двумя способами: используя конструктор или метод `lock`.

```
weak_ptr<Container> w = ...;
shared_ptr<Container> p( w );
```

В случае, если объект уже удалён, в конструкторе `shared_ptr` будет сгенерировано исключение

```
if( shared_ptr<Widget> p = w.lock() ) {
...
}
```

В случае, если объект уже удалён, то `p` будет пустым указателем. Работать с объектом `shared_ptr` можно, только если он не был удалён.

Пример 117. В примере демонстрируется использование умного указателя `weak_ptr`.

```
#include <iostream>
#include <memory>
using namespace std;
class DemoSmartPtr {
public:
  string name;
  DemoSmartPtr(string s = "") : name(move(s)) {
    cout << "CTOR " << name << '\n';
  }
  ~DemoSmartPtr() {
    cout<<"DTOR "<<name<< '\n';
  }
};
```

```
void weak_ptr_info(const weak_ptr<DemoSmartPtr>& p){
  cout << "----------" << boolalpha
    << "\nexpired:  " << p.expired()
    << "\nusecount: " << p.use_count()
    << "\ncontent:  ";
  const auto sp(p.lock());
  if (sp)
    cout << sp->name << '\n';
  else
    cout << "<null>\n";
}
int main() {
weak_ptr <DemoSmartPtr> weak_p;
  weak_ptr_info(weak_p);
  {
    auto shared_p(make_shared<DemoSmartPtr>("Shared"));
    weak_p = shared_p;
    weak_ptr_info(weak_p);
  }
  weak_ptr_info(weak_p);
  cout << "END OF main \n";
  return 0;
}
```

Достоинства умных указателей позволяют рекомендовать использовать их вместо «сырых указателей» как можно чаще.

Хотя указатели по-прежнему являются важной частью языка C++, тем не менее всегда следует использовать наиболее подходящие типы данных, особенно если они более безопасны.

Рекомендуется использовать:

- ✓ умные указатели вместо традиционных указателей;
- ✓ объекты `std::vector` или `std::array` вместо традиционных массивов;

- ✓ объекты `std::string` вместо строк в стиле C.

Тем не менее традиционные указатели всё ещё нужны. Например:

- ✓ может возникнуть необходимость создать динамическую структуру, которая отсутствует в стандартной библиотеке;
- ✓ при передаче параметров функционального типа;
- ✓ если необходимо внутри функции проверять существование объекта, передаваемого в качестве параметра, сравнивая его с `nullptr`;
- ✓ при изучении, доработке и использовании устаревшего кода.

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. *Галовиц Я.* C++17 STL. Стандартная библиотека шаблонов. — СПб.: Питер, 2018. — 432 с.
4. *Дейтел Х. М., Дейтел П. Дж.* Как программировать на C++. — М.: ЗАО «Издательство БИНОМ», 2000. — 1024 с.
5. *Демяненко Я.М., Чердынцева М.И.* Методы процедурного программирования в C++. — Ростов-на-Дону: Издательство Южного федерального университета, 2014. — 207 с.
6. *Кениг Эндрю, Му Барбара.* Эффективное программирование на C++. Практическое программирование на примерах. — М.: Издательский дом «Вильямс», 2002. — 384 с.
7. *Керниган Б., Ритчи Д.* Язык программирования C. Второе издание, переработанное и дополненное. — М.: Вильямс, 2019. — 288 с.
8. *Лишнер Рэй.* C++. Справочник. — СПб.: Питер, 2005. — 907 с.
9. *Мюссер Дэвид Р., Дердж Жилмер Дж., Сейни Атул*. C++ и STL. Справочное руководство. – М.: Вильямс, 2010. – 432 с.
10. *Плаугер П. Дж., Степанов Александр, Ли Менг, Мюссер, Дэвид Р.* STL — стандартная библиотека шаблонов C++. — СПб.: БХВ-Петербург, 2004. – 656 с.
11. *Прата С.* Язык программирования C: Лекции и упражнения = C Primer Plus. — М.: Вильямс, 2006. — 960 с.

12. *Русанова Я.М., Чердынцева М.И.* C++ как второй язык в обучении приемам и технологиям программирования. — Ростов н/Д: Изд-во ЮФУ, 2010. — 200 с.
13. *Седжвик Р.* Фундаментальные алгоритмы на C. Анализ/Структуры данных/Сортировка/Поиск/Алгоритмы на графах. — СПб.: ООО «ДиаСофтЮП», 2003. — 1136 с.
14. *СкоттМайерс.* Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. — М.: ДМК Пресс, 2010 — 302 с
15. Стандарт C++20 (ISO/IEC JTC1 SC22 WG21 N 4860:2020) https://isocpp.org/files/papers/N4860.pdf — Дата последнего доступа 18.11.2024
16. *Страуструп Бьерн.* Язык программирования C++. Специальное издание. — М.: ООО «Бином-Пресс», 2004. — 1104 с.
17. *Эккель Брюс.* Философия C++. Введение в стандартный C++. — СПб.: Питер, 2004. — 572 с.
18. *Эккель Брюс, Эллисон Чак.* Философия C++. Практическое программирование. — СПб.: Питер, 2004. — 608 с.
19. cppreference.com https://ru.cppreference.com/w/ — Дата последнего доступа 21.05.2025
20. *David Gries*. The Science of Programming. Springer Verlag, New York, 1981, 350 pages. (Русское издание: Дэвид Грис. Наука программирования — М.: Мир, 1984. — 416 с.).

Учебное издание

Демяненко Яна Михайловна,
Чердынцева Марина Игорьевна

C++ как второй язык в обучении приемам и технологиям программирования

2-е издание, переработанное и дополненное

Подписано в печать 07.10.2025 г.
Бумага офсетная. Формат 60×84/16. Тираж 30 экз.
Усл. печ. лист 24,47. Уч.-изд.лист. 12,5. Заказ № 10143.

Издательство Южного федерального университета.

Отпечатано в отделе полиграфической, корпоративной и сувенирной продукции
Издательско-полиграфического комплекса КИБИ МЕДИА ЦЕНТРА ЮФУ.
344090, г. Ростов-на-Дону, пр. Стачки, 200/1, тел. (863) 243-41-66.)


9 785927 549719