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

Я. М. ДЕМЯНЕНКО, М. И. ЧЕРДЫНЦЕВА

МЕТОДЫ ПРОЦЕДУРНОГО ПРОГРАММИРОВАНИЯ В C++

Учебное пособие

Ростов-на-Дону
2014

УДК 004.43
ББК 32.973.2
Д 32

Печатается по решению Редакционно-издательского совета
Южного Федерального университета

Рецензенты:
доктор ф.-м.н., профессор КубГУ **А. И. Миков;**
доктор т.н., профессор РГУПС **М. А. Бутакова**

Д 32 Демяненко Я. М.

Методы процедурного программирования в C++ : учебное пособие / Я. М. Демяненко, М. И. Чердынцева; Южный федеральный университет. – Ростов-на-Дону: Издательство Южного федерального университета, 2014. – 207 с.
ISBN 978-5-9275-1486-1

Эта книга написана в качестве учебного пособия по курсу «Языки и методы программирования» бакалаврской программы по направлению «Прикладная математика и информатика». Книга может быть использована для самостоятельного изучения языка C++ магистрами по программе «IT in Biomechanics», разработанной в рамках проекта ICARUS. Она предназначена для студентов, имеющих базовые знания по основам программирования, например, на языке Паскаль. В данном учебном пособии уделяется внимание процедурным методам языка C++.

Публикуется в авторской редакции.

ISBN 978-5-9275-1486-1 УДК 004.43
ББК 32.973.2

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Эта книга написана в качестве учебного пособия по курсу «Языки и методы программирования» бакалаврской программы по направлению «Прикладная математика и информатика». Книга может быть использована для самостоятельного изучения языка C++ магистрами по программе «IT in Biomechanics».

В данном издании мы стремились достичь двух целей: во-первых, облегчить изучение языка C++ студентам, имеющим базовые знания по основам программирования, например, на языке Паскаль; во-вторых, продемонстрировать приемы решения задач с учетом особенностей языка C++.

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

ВВЕДЕНИЕ

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

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

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

Модуль 3 содержит материал, который позволит студентам освоить работу со списочными структурами и файлами. Изложение материала иллюстрируется подробным рассмотрением примеров решенных задач. Такой подход позволит студентам научиться применять теоретические знания при решении конкретных задач.

Некоторые вопросы, оставшиеся за рамками рассмотрения в первых трех модулях, вынесены в модуль 4.

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

Для облегчения работы с текстом в книге приняты следующие соглашения:

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

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

1.1. Используемые термины

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

Основные терминологические проблемы возникают вокруг трех понятий:

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

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

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

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

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

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

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

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

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

1.2. Языки C и C++

Язык C++ ведет историю своего происхождения от языка C. Поэтому он также как и язык C обладает такими характеристиками как эффективность, компактность, быстродействие и переносимость.

Поскольку синтаксис C++ унаследован от языка C, то одним из принципов разработки было сохранение совместимости с C. При этом C++ не является в строгом смысле расширением C.


Бьёрн Страуструп, придумавший C++, неоднократно выступал за максимальное сокращение различий между C и C++ для создания максимальной совместимости между этими языками. Существует и другая точка зрения: так как C и C++ являются двумя различными языками, то и совместимость между ними не так важна, хоть и полезна.

В дальнейшем C и C++ развивались независимо, что привело к росту несовместимости между ними. Редакция C99 добавила в язык несколько конфликтующих с C++ особенностей. Эти различия затрудняют написание программ и библиотек, которые могли бы нормально компилироваться и работать одинаково и в C и в C++.

Множество программ, соответствующих стандарту C, соответствуют и стандарту C++. Программы, написанные на языке C++, могут пользовать-

ся многими существующими библиотеками, написанными для C. Но при этом существуют средства языка C, которые не включены в стандарт C++.

С другой стороны некоторые возможности языка C++ были добавлены в стандарты, начиная с C99. Например, inline-функции, возможность приближать объявления переменных к месту их использования, более сильную проверку типов.

Язык C++ объединяет три парадигмы программирования: процедурное программирование, объектно-ориентированное программирование, обобщённое программирование. В данном учебном пособии рассмотрено процедурное программирование на языке C++ с использованием некоторых классов стандартной библиотеки.

1.3. Особенности языка

Прежде всего, отметим первое правило.

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

Ключевые слова – это идентификаторы, имеющие особое назначение и зарезервированные самим языком. В таблице 1 приведен список ключевых слов языка C++ (в соответствии со стандартом ISO/IEC 14882:2011).

Все идентификаторы, не являющиеся ключевыми словами, должны быть объявлены либо напрямую, либо косвенно через подключаемые заголовочные файлы.

Объявление — это термин, описывающий все, что можно сообщить компилятору о каком-либо идентификаторе.

Область видимости — это область программного кода, содержащая объявления. Каждое объявление добавляет новый идентификатор (имя) в область видимости, и каждое использование идентификатора заставляет

компилятор искать содержащую его область видимости. Иногда компилятору можно явно указать, в какой области видимости находится имя, в остальных случаях компилятор будет определять эту область самостоятельно. Если компилятору известна область видимости, то он может определить, что обозначает идентификатор (переменная, функция и т.д.) и как оно может быть использовано. Таким образом, область видимости можно рассматривать как словарь соответствий имен и объявлений.

Области видимости делятся на именованные и неименованные. К именованным областям видимости относятся классы и пространства имен, к неименованным — блоки инструкций, тела функций и неименованные пространства имен.

Таблица 1

Список ключевых слов

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Если идентификатор находится в именованной области видимости, его можно *квалифицировать*, указав имя области видимости. Обычно в программе большинство имен не квалифицируется.

Следует отметить следующую особенность языков C и C++. Перед тем как будет произведена компиляция программы, она обрабатывается препроцессором. *Препроцессор* C/C++ — это программный инструмент, изменяющий код программы для последующей компиляции и сборки. Работа препроцессора заключается в удалении комментариев и обработке команд препроцессора, называемых директивами. *Директивы* препроцессора начинаются со знака #. Каждая директива пишется на отдельной строке.

Язык C отличается от других языков высокого уровня широким использованием указателей. *Указатель* (англ. *pointer*) — переменная, значением которой может быть адрес ячейки памяти или специальное значение — *нулевой адрес*. Последнее используется для указания того, что в данный момент переменная указатель не хранит значение адреса. Указатели совместно с адресной арифметикой играют в C особую роль. Можно сказать, что они определяют лицо языка. Благодаря им C может считаться одновременно языком высокого и низкого уровня по возможностям работы с памятью.

В языке C++ активно поддерживается использование указателей и их роль возрастает. На указателях базируется один из принципов объектно-ориентированного программирования — полиморфизм.

1.4. Первые шаги

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

Программа «Hello, world!» в виде консольного приложения — типичный пример, упоминаемый практически во всех учебниках.

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

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

Первая строка содержит директиву препроцессора:

```
#include <iostream>
```

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

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

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

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

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

При обращении к объектам, объявленным в этом пространстве, имя пространства имен может быть указано явно. Для этого используется операция разрешения области видимости `::`. Такой способ удобно использовать, если к какому-то пространству имен потребуется одно-два обращения или если необходимо явно подчеркнуть, что используемое имя принадле-

жит конкретному пространству имен. В противном случае, чтобы не загромождать текст программы, рекомендуется выносить объявление необходимого пространства имен, используя оператор `using namespace`.

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

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

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

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

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

Функция `main()` должна вернуть управление операционной системе после своего завершения и сообщить код возврата. В случае аварийного завершения операционной системе будет сообщено ненулевое целочисленное значение — код ошибки (код возврата). В случае безошибочной работы нужно возвратить признак нормального завершения `0`. Именно поэтому в заголовке функции `main()` указано, что возвращаемый ею результат должен быть целого типа, а последним оператором тела функции `main()` является оператор возврата:

```
return 0;
```

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

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

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

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

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

Для ввода данных объект `cin` использует операцию >>. Справа от этого знака находится переменная, принимающая вводимую информацию. В процессе ввода последовательность символов, вводимых с клавиатуры, преобразуется к типу, соответствующему переменной, принимающей информацию. Если это невозможно, генерируется ошибочная ситуация.

Использование объектов `cin` и `cout` возможно, потому что в программе подключен файл <iostream> и объявлено пространство имен `std`,

где определены эти объекты. Напомним также, что операции >> и << могут быть применены последовательно многократно.

Далее рассмотрим фрагмент программы, в котором задействованы управляющие конструкции `if` и `for`.

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

Обычно рекомендуют объявлять переменные непосредственно перед их использованием.

Чтобы избежать некорректного использования переменной цикла `i` за его пределами рекомендуется объявлять ее в заголовке цикла. Время жизни такой переменной заканчивается за его пределами.

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

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

Первый вариант:

```
if (выражение)
  оператор
```

Второй вариант:

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

В программе представлен первый вариант:

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

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

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

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

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

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

Общая форма цикла `for` выглядит так:

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

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

1.5. Функции как строительные блоки программы

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

Объявления глобальных объектов должны располагаться вне определений функций, входящих в программу.

Выполнение C++ программы начинается с выполнения функции `main()`.

Среди функций должна быть одна и только одна с именем `main()`.В частном случае программа может состоять только из функции `main()` и подключения заголовочных файлов.

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

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

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

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

Разберем этот пример подробно. В программе продемонстрировано использование двух типов комментариев: первый является однострочным, он начинается с пары символов // и заканчивается концом строки; далее

в тексте программы помещен многострочный комментарий, который начинается с пары символов /* и заканчивается парой символов */. Многострочный комментарий может занимать одну или несколько строк, а также только часть строки.

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

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

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

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

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

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

Это правило относится и к именам функций.

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

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

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

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

В случае переменной объявление и определение часто совпадают.

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

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

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

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

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

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

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

✓ операторы объявления (или описания)

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

✓ операторы ввода

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

✓ операторы вывода

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

✓ операторы вызова функции

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

✓ оператор возврата результата выполнения функции

```
return 0;
```

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

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

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

Рассмотрим операторы вызова функции.

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

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

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

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

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

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

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

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

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

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

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

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

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

Вернемся к операторам вызова функции.

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

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

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

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

Рассмотрим определения этих двух перегруженных функций.

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

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

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

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

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

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

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

Пример 4. Реализовать перегруженные функции для нахождения максимального из нескольких значений, возможно, разных типов.

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

Рассмотрим нахождение максимального значения для следующих вариантов параметров: два целых числа; три целых числа; два вещественных числа.

Листинг файла func_overload.cpp

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

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

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

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

Вызовы функций приводят к накладным расходам, которые могут снижать производительность. В C++ для снижения этих накладных расходов — особенно для небольших функций — предусмотрен механизм *встраиваемых* (`inline`) *функций*, которые иногда еще называют подставляемыми.

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

Компилятор может игнорировать спецификацию `inline`. В таком случае копии кода не создаются, а осуществляется обычный вызов функции.


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

1.6. Аргументы функции по умолчанию

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

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

Пример 5. Описать функцию, вычисляющую расстояние между двумя точками на плоскости. Предусмотреть возможность использования значений аргументов по умолчанию. Вычилить с ее помощью периметры двух треугольников. У первого треугольника одна из вершин лежит в точке с координатами (0,0), две другие — произвольные точки на плоскости. У второго треугольника одна из вершин лежит в точке с координатами (0,0), вторая — на оси Ox, третья — на оси Oy.

```
#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;
  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;
  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;
}
```

Прототип функции `dist` содержит значения по умолчанию для трех параметров:

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

При таком объявлении эту функцию можно вызвать с одним, двумя, тремя или четырьмя параметрами. Опускать можно только параметры, расположенные подряд в конце списка параметров. Например, можно задать список фактических параметров одним из следующих способов:

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

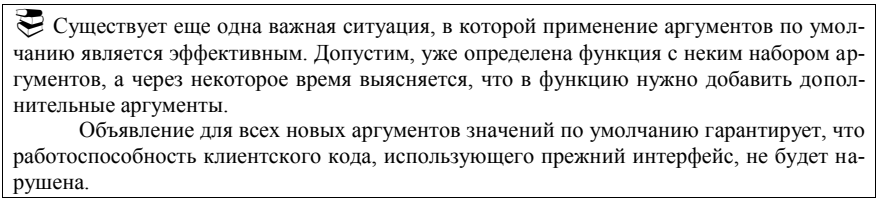
Значения по умолчанию указываются либо только в описании функции, либо только в определении.

☺ Рекомендуется включать значения по умолчанию в описание функции.

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

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

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


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

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

В примере 5 для корректного отображения символов национального алфавита используется оператор подключения кодовой страницы, определенной в настройках операционной системы (локали):

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

В C++ локаль — это объект типа `locale`. В нем хранятся свойства символов, настройки форматирования и прочая информация.

1.7. Многофайловый проект, включение заголовочных файлов

Напомним требования к структуре программы. Программа на языке C++ состоит из одной или нескольких функций, объявлений и определений глобальных объектов, подключения пространства имен и заголовочных файлов.

Все функции, составляющие программу, могут быть расположены в одном файле — исходном модуле.

Реальные программы на C++, как правило, состоят из нескольких исходных модулей. Это возможно благодаря тому, что C++ и C поддерживают раздельную компиляцию модулей.

Пример 6. Дано целое число. Определить количество и сумму цифр в десятичной записи этого числа. Выдать само число и число, получающееся из него при вычеркивании первой и последней цифр.

Оформим в виде функций каждое из действий:

- определение количества цифр в десятичной записи числа;
- вычисление суммы цифр числа;
- преобразование числа путем вычеркивания из него первой и последней цифры.

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

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

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

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

```
  return k;
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- ✓ цикла с предусловием

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

Оператор цикла с предусловием является универсальным оператором цикла:

```
while (условие)
  оператор;
```

- ✓ инкремента

```
k++;
```

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

- ✓ комбинированные или составные операторы присваивания

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

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

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

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

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

дуль по отдельности, то говорят, что компилятор поддерживает принцип раздельной компиляции.

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

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

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

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

```
#include <cstdlib>
```

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

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

```
#include <iostream>
using namespace std;
/*   объявления функций
     сами функции описаны после функции main()
*/
int count_dig(int a);//параметр передается по значению
int sum_dig(int a);
void del_last_dig(int& a);//параметр передается по ссылке
int del_first_dig(int& a);
int main() {
  int x;
  cin>>x;
  cout<<count_dig(x)<<endl;
  cout<<sum_dig(x)<<endl;
/*следующие две функции изменяют значение параметра, поэтому
  сохраним исходное число во вспомогательной переменной
*/
  int r=x;
  del_last_dig(r);
  cout<<x<<' '<<r<<endl;
  cout<<del_first_dig(r)<<' '<<r;
}
```

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

Очевидно, что хорошим решением в этом случае было бы наличие заголовочного файла, содержащего объявления всех функций, вынесенных в отдельный файл. Это удобно тем, что не нужно размещать в основном

файле, содержащем функцию `main()`, объявления всех функций. Достаточно подключить соответствующий h-файл.

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

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

Подключение заголовочного файла, используемого в проекте, выглядит так:

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

☺ Правило хорошего стиля требует в случае вынесения функций в отдельный cpp-файл создавать соответствующий ему h-файл, содержащий объявления соответствующих функций. Рекомендуется, чтобы имена cpp-файла и соответствующего ему h-файла совпадали.

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

1.8. Заголовочные файлы и библиотеки в C++

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

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

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

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

Пример 7. Найти все простые числа в диапазоне от 2 до N.

Рассмотрим программу для решения задачи поиска простых чисел.

Листинг Prime.cpp — первый вариант

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

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

После запуска программы на выполнение можно увидеть следующие результаты:

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

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

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

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

Проконтролируем значение переменной `y` после выполнения оператора

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

Запустим программу на выполнение при N=10.

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

Значит, условный оператор

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

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

В правильном варианте программа имеет следующий вид:

Листинг Prime.cpp — исправленный вариант

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

bool IsPrime (int x) {
  int z = int(sqrt((double)x));
```

```
  for (int i=2;i<=z; ++i) {
    int y=x%i;
    if (y==0)
      return false;
    }
  return true;
}

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

Обратите внимание, что вычисление значения, ограничивающего повторение цикла, вынесено из заголовка цикла.

```
int z = int(sqrt((double)x));
for (int i=2;i<=z; ++i) {
…
}
```

> ➢ В отличие от языка Паскаль в C++ выражение условия в операторе `for` вычисляется на каждой итерации. Поэтому вычисляемые выражения, значения которых не меняются при выполнении цикла, рекомендуется выносить из условия продолжения цикла.

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

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

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

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

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

Рассмотрим следующий фрагмент:

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

В современном варианте он выглядит так:

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

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

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

Пример 8. Для заданных натуральных чисел `n` и `k` определить, равно ли число `n` сумме `k`-х степеней своих цифр. Для решения задачи составить функцию целого типа для возведения целого числа в целую степень.

Поместим определение функции в отдельный cpp-файл и создадим для этого cpp-файла заголовочный h-файл.

Листинг файла functions.cpp

```
int power(int x, int k){
  int f=1;
  for (int i=1;i<=k; ++i)
    f*=x;
  return f;
}
```

Листинг файла task6.cpp

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

using namespace std;

int main() {
  int n,k;
  cout<<"Input n and k"<<endl;
  cin>>n>>k;
  int m=n;//сохраним копию числа
  int sum=0;
  while (m){
    int dig = m % 10;
    sum+=power(dig,k);
    m/=10;
  }
  cout<<"result = "<<(sum==n)<<endl;
  return 0;
}
```

Листинг файла functions.h.

```
int power(int, int);
```

Если сравнить содержимое заголовочного файла functions.h с содержимым заголовочных файлов стандартных библиотек языка C++, таких как iostream, то можно заметить, что в начале нашего файла отсутствуют директивы препроцессора. Рассмотрим, для чего они используются.

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

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

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

В каждом заголовочном файле следует сначала проверить, не был ли этот заголовочный файл уже включен в многофайловый проект. Это делается при помощи препроцессорного флага. Проверка «установлен ли флаг» задается директивой `ifndef имя флага`. Если флаг не установлен, значит, код заголовочного файла еще не включался. В этом случае флаг устанавливается директивой `define имя флага`, и включаются объявления, расположенные до директивы `endif`. Если флаг уже установлен, то код до директивы `endif` игнорируется.

Примерный формат заголовочного файла:

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

Имя флага может быть любым уникальным именем.

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

Например, для файла с именем functions.h можно, следуя данному правилу, получить имя `FUNCTIONS_H`.

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
int power(int, int);
#endif    // FUNCTIONS_H
```

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

1.9. Целочисленные типы данных

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

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

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

В порядке возрастания (а точнее, неубывания) размерности базовые целые типы — это: `char`, `short int`, `int` , `long int`, и `long long int`. Каждый из этих типов подразделяется на две разновидности: со знаком (`signed`) и без знака (`unsigned`). Для типов `short int`, `long int`, и `long long int` идентификатор `int` можно опускать. В таком случае имена типов будут выглядеть так: `short`, `long`, и `long long`.

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

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

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

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

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

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

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

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

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

Пример 9. Найти наибольший общий делитель (НОД) целых чисел A и B.

Для вычисления НОД используем алгоритм, основанный на соотношении:

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

где $a>0, b>0.$

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

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

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

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

Другой алгоритм, традиционно называемый алгоритмом Евклида, использует соотношение

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

Рассмотрим только, как изменится функция, вычисляющая НОД.

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

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

1.10. Поразрядные операции над целочисленными типами данных

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

Бинарные поразрядные операции, за исключением операций сдвига, являются коммутативными. При этом традиционно значение левого операнда принято считать проверяемым или подвергаемым изменению, а значение правого операнда задает принцип проверки или изменения левого операнда. В этом случае правый операнд называется *маской*.

В таблице 2 приведено краткое описание семантики поразрядных операций в предположении, что правый операнд является маской.

Таблица 2

Краткое описание семантики поразрядных операций

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

Пример 10. Написать программу, демонстрирующую использование поразрядных операций для получения двоичного представления содержимого байта.

```
#include <iostream>
using namespace std;
void dispBinary (unsigned char u);
int main() {
  unsigned char u;
  cout<<"input number between 0 and 255 :";
  cin >> u;
  cout <<"number in binary code:";
  dispBinary(u);
  cout << "addition to unity: ";
  dispBinary(~u);
  return 0;
}
```

```
void dispBinary (unsigned char u) {
  unsigned char t;
  for (t=128; t>0; t=t>>1)
    if (u&t)
      cout <<"1";
    else
      cout <<"0";
  cout<<"\n";
}
```

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

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

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

Начальное значение параметра цикла `t`, используемого как маска, число 128 соответствует двоичному 10000000. На каждом шаге итерации единица в записи числа `t` сдвигается на одну позицию вправо. Такие значения маски позволяют выделять двоичные разряды в числе слева направо. Если в разряде, определяемом маской `t`, стоит 1, поразрядное & даст в этом разряде ненулевой результат.

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

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

Пример 11. Реализовать эффективные операции умножения и целочисленного деления на 2, проверку на нечетность целого числа.

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

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

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

Сдвиг влево на 1 бит соотвествует умножению числа на 2, вправо — целочисленному делению. Значение младшего (самого правого) бита позволяет определить четность числа: если бит равен 1, то число является нечетным, если 0 — четным.

Пример 12. Реализовать функции для выделения бита с заданным номером и байта с заданным порядковым номером из беззнакового целого числа.

```
int get_bit(unsigned int x, unsigned int n){
  return (x>>(n-1))&1;
}

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

В функции `get_bit` бит с заданным номером сдвигается в самый правый разряд и затем выделяется с помощью операции `&` «побитовое и» с единицей.

В функции get_byte байт с заданным номером сдвигается вправо на место младшего байта и затем выделяется с помощью операции & «побитовое и» со значением 255, соответствующему в двоичном представлении восьми единичным разрядам (11111111).

Пример 13. Разработать функции для преобразования латинских букв в нижний или верхний регистр.

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

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

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

Пример 14. Даны два целых числа: сообщение и ключ шифрования. Закодировать сообщение и раскодировать его.

```
int code (int origin, int key) {
  return origin^key;
}

int main() {
  int key = 7777;
  int val = 23456;
```

```
  val = code(val,key); //кодирование
  cout<<val<<endl;
  val= code(val,key); //раскодирование
  cout<<val<<endl;
  return 0;
}
```

В данном примере продемонстрировано свойство операции «исключающее или» (^). Эта операция является обратимой, т.е. ее повторное применение восстанавливает исходное значение (x^y) ^ y = x. Поэтому операция нашла применение в криптографии как простейшая реализация идеального шифра (шифра Вернама).

1.11. Типы данных для вещественных значений

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

Все арифметические операции в языке C++ являются полиморфными, т. е. тип результата зависит от типа операндов.

> ➢ В отличие от языка Паскаль, в языке C++ операция деления (/), как и все остальные арифметические операции, является полиморфной.

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

Пример 15. Вычислить сумму $\sum_{i=1}^{n} \frac{1}{i}$.

```
#include <iostream>

using namespace std;

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

int main() {
  int n;
  cout<<"input number of terms"<<endl;
  cin>>n;
  cout.precision(20);
  cout<<sum_1(n)<<endl;
  cout<<sum_2(n)<<endl;
  return 0;
}

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

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

В этом примере вычисление конечной суммы осуществляется с помощью двух функций, отличающихся только организацией цикла суммирования.

Первая функция выполняет суммирование, начиная с первого слагаемого до последнего. Вторая функция выполняет суммирование в обратном порядке — от последнего слагаемого до первого. Задавая большие значения (например, 2000) количества суммируемых слагаемых `n`, можно увидеть, что результаты суммирования начинают отличаться из-за накопления ошибки округления (рис. 1).

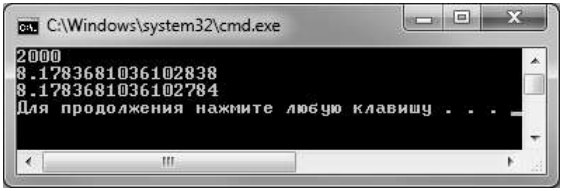


Рис. 1. Результат выполнения при $n = 2000$

В операторе объявления

```
double s=0;
```

продемонстрирована возможность использования инициализатора при объявлении.

Операторы цикла `for`

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

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

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

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

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

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

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

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

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

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

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

Пример 16. Используя разложение функции `sin(x)` в ряд $\sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{(2i+1)!}$, построить таблицу значений функции на отрезке [`a`, `b`] с шагом `h`.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
double m_sin(double x);
int main() {
  double a,b,h;
  cout<<"input a, b, h"<<endl;
  cin>>a>>b>>h;
```

```
  double bg=b+h/3;
  for(double x=a; x<bg; x+=h)
    cout<<setw(6)<<x<<setw(10)<<setprecision(3)<<m_sin(x)
    <<setw(10)<<setprecision(3)<<sin(x)<<endl;
  return 0;
}

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

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

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

Здесь мы воспользовались дополнительными манипуляторами для объекта `cout`. В отличие от стандартного манипулятора `endl` для их использования требуется подключать заголовочный файл iomanip. Манипуляторы включаются непосредственно в поток и действуют на ближайший следующий за ними элемент вывода.

Манипулятор `setw(6)` позволяет установить количество позиций для вывода значения переменной `x`, `setw(10)` — для вывода `m_sin(x)` и `sin(x)`.

Действие манипулятора `setprecision(n)` аналогично действию функции `precision(n)` объекта `cout`.

Количество манипуляторов больше чем функций объектов `cin`/`cout`. Манипуляторы позволяют устанавливать следующие флаги и

параметры: форматирования, пропуска пробельных символов при вводе, сброса буфера вывода и т.п.

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

В объявлении константы `eps` использован квалификатор `const`, который указывает, что объявляемый объект не может быть модифицирован:

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

Обратите внимание на закомментированный оператор цикла

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

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

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

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

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

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

1.12. Указатели

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

Объявление указателя:

```
<тип> * <имя>;
```

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

```
int *p, *q, *t;
int a=0;
p=&a;
t=p;
q=new int;
*q=5;
int b=*q;
```

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

```
*q=5;
```

Использование значения переменной `b` и значения `*q` равнозначно

```
cout<<b<<" "<<*q;
```

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

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

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

- ✓ использовать операцию "->"

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

Для обозначения «пустого» указателя, начиная со стандарта 1998 г., используется значение `0`. Ранее для этих целей использовалась константа `NULL`.

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

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

```
delete q;
```

Нужно понимать, что адресом переменной является целое число. Можно увидеть значение указателя, так же как значение перемнной любого базового типа.

Пример 17. Дан код программы. Дополнить его так, чтобы программа выводила информацию о расположении всех ее переменных в памяти.

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

  double a=2.5,b=0.9;

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

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

int main() {
  int k = 5, m= 7, n;
  p=my_max(k,m);
  my_swap(a,b);
  return 0;
}
```

В приведенном ниже варианте решения 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 my_swap(double &a, double &b){
  double r = a;
  double *t = &a;
  cout<<"    &a="<<t<<"  &b="<<&b<<"  &r="<<&r<<endl;
  a=b;
  b=r;
}

int my_max(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=my_max(k,m);
  cout<<endl<<"in swap ->"<<endl;
  my_swap(a,b);
  return 0;
}
```

Результаты выполнения при каждом запуске могут быть разными. Результаты одного из запусков представлены на рисунке 2.

Адреса выдаются в шестнадцатеричном формате. Проанализировав их значения можно увидеть, что глобальные и локальные переменные располагаются в разных областях памяти.

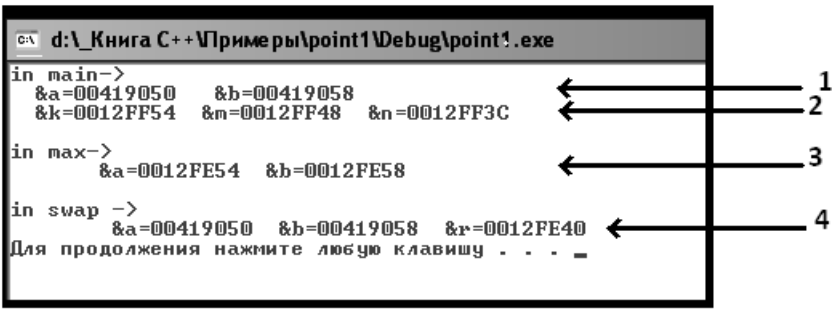


Рис. 2. Вывод адресов переменных

Программе перед ее выполнением выделяется фиксированная область памяти. Она делится на части: память под код программы, память для глобальных переменных, память стека вызовов подпрограмм.

Глобальные переменные находятся в статической области памяти программы и располагаются по возрастанию адресов (рис. 2 – 1).

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

Локальные переменные располагаются в автоматической памяти. Автоматическая память организована как стек вызовов подпрограмм. Стек начинается с адреса, имеющего максимальное значение. В процессе выполнения стек растет в сторону уменьшения адресов (рис. 2 – 2).

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

При передаче параметров по значению создаются копии фактических параметров, которые размещаются на стеке (рис. 2 – 3).

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

При передаче по ссылке функция получает адрес переменной (фактического параметра) и тоже размещает его на стеке (рис. 2 – 4).

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

Чтобы избежать ошибок использования указателей, связанных с передачей параметров и адресной арифметикой, важно понимать различие между константным указателем и указателем на константу. Различия в синтаксисе определяются положением ключевого слова `const` при объявлении указателя.

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

Вот несколько примеров, демонстрирующих корректное использование указателей на константу и константных указателей и типичные ошибки при работе с ними.

```
int a=100;
int b=222;

int *const P2=&a;   //Константный указатель
*P2=987;  //Менять значение разрешено,
//P2=&b;  //но изменять адрес не разрешается

const int *P1=&a;   //Указатель на константу
//*P1=110;     //Менять значение нельзя,
P1=&b;    //но менять адрес разрешено
const int *const P3=&a;  //Константный указатель на константу
//*P3=155;     //Изменять нельзя ни значение
//P3=&b;  //ни адрес, к которому такой
    //указатель привязан
```

Неправильное использование константных указателей и указателй на константу вызывает ошибку компиляции.

Модификатор `const` чаще всего используется при передаче параметров через указатель с целью защитить от изменений либо сам указатель, либо данные, на которые он ссылается.

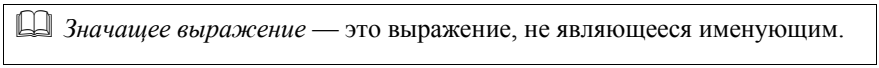
```
void f(int *a, const int *b, int *const c, const int *const d);
```

Заголовок функции `f` определяет, что в ее теле для указателя `a` может измениться и сам указатель и значение, на которое он ссылается. Для указателя `b` допустимо изменение только самого указателя, для `c` — значения, на которое ссылается указатель. Для `d` недопустимы ни те, ни другие изменения.

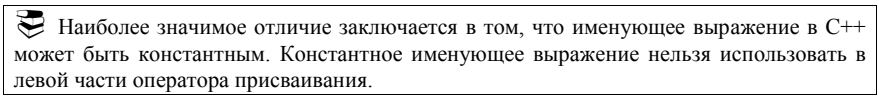
1.13. Выражения и операции

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

Именующими являются имя переменной, массив, ссылка на элемент массива по индексу, разыменованный указатель. Именующее выражение всегда связано с некоторой областью памяти, адрес которой известен.



Язык C++ заимствовал термины «именующее выражение» и «значащее выражение» из языка C, но трактует эти понятия шире.



Поэтому в операциях присваивания левым операндом должно быть модифицируемое именующее выражение. Операция взятия адреса `&` применяется к именующим выражениям, операции инкремента и декремента (`++`, `--`) применяются к модифицируемым именующим выражениям. Во

всех остальных операциях требуется использовать значащее выражение. В случае использования в них именующего выражения, оно будет преобразовано в значащее выражение.

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

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

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

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

Выполнение программы на C++ сводится к вычислению выражений. При этом некоторые выражения могут иметь один или несколько побочных эффектов. Иногда выражения включаются в программу только ради их побочных эффектов. Выражения состоят из операндов и операций.

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

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

Для бинарных арифметических и логических операций, для операций сдвига ассоциативность определена слева направо, а для побитовых логических — справа налево. Для всех унарных операций, в том числе и операции разыменования (*) ассоциативность определена справа налево.

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

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

`++a` — префиксная форма;

`a++` — постфиксная форма.

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

В качестве примера рассмотрим два случая:

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

В первом случае после выполнения двух операторов: `a=6` и `b=5`.

Во втором же случае: `a=6` и `b=6`.

Операция *присваивания* используется в выражениях присваивания. В выражении присваивания значение правого операнда присваивается левому. Левый операнд выражения присваивания должен быть именующим выражением, которое можно изменять. Результат выражения присваивания также будет именующим выражением — это будет левый операнд, после

того, как присваивание завершится. Это означает, что результат выражения присваивания может быть использован в другом выражении, в частности, в другом выражении присваивания:

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

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

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

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

Иногда удобно использовать операцию запятая и в выражении инициализации.

Операция запятая гарантирует *сериализацию* вычисления выражения. Это означает, что левый операнд будет вычислен раньше правого. Поэтому результат вычисления левого операнда может быть использован при вычислении правого. Результатом выражения с операцией запятая является значение правого операнда. У операции запятая самый низкий приоритет.

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

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

1.14. Операторы (управляющие инструкции)

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

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

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

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

Оператор «выражение» представляет собой любое выражение, за которым следует точка с запятой. Поскольку результат оператора выражения не используется, этот оператор включается в программу ради его побочных эффектов. Некоторые примеры приведены в таблице 3.

Таблица 3

Примеры побочных эффектов

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

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

Общий вид оператора цикла `for`:

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

Все три части, входящие в заголовок цикла (инициализация, условие и итерация), являются выражениями. Выражение итерации выполняется после каждого повторения оператора тела цикла. Условие проверяется перед каждым повторением цикла. Выражение инициализации выполняется перед первым выполнением оператора цикла. Важно, что все три выражения в заголовке могут отсутствовать. При этом отсутствующее выражение условия считается всегда истинным.

Например, бесконечный цикл можно записать следующим образом:

```
for (;;) оператор
```

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

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

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

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

```
if (условие)          switch (целочисленное выражение) {
     оператор1           case конст-выр1 : операторы
[else                    case конст-выр2 : операторы
     оператор2]        . . .
                         [default     : операторы]
                      }
```

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

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

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

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

Пример 18. Ввести последовательность символов, завершающуюся символом $. Посчитать, сколько раз среди вводимых символов встретились цифры 0, 1, 3, 7.

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

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

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

извлекает из входного потока один символ и возвращает извлеченный символ.

1.15. Ошибки и их обработка

Во время выполнения программы возможно возникновение ошибок, причинами которых могут быть некорректные данные, некорректная работа с памятью или файлами.

Такие ошибки называются ошибками времени выполнения. Они могут возникать не при каждом запуске программы, что затрудняет их поиск.

Поэтому рекомендуется предусматривать потенциальные ошибки времени выполнения и правильно их обрабатывать. Существуют несколько вариантов такой обработки.

Если ошибка может произойти внутри функции, то необходимо сформулировать охраняющее условие. При хороших данных функция должна вернуть результат или просто выполнить необходимые действия, а вслучае плохих — можно воспользоваться одним из приемов:

- аварийное завершение выполнения;
- выдача диагностики и завершение выполнения;
- возврат признака ошибки (например, `EOF`).

В первом случае используются функции `exit(code)` и `abort()`, выполняющие выход из всей программы. Функция `exit()` позволяет вернуть код возврата как признак ошибки.

Для реализации второго варианта обработки ошибок времени выполнения можно использовать функцию `assert()`.

```
int calc(int a, int e){
  assert(e,"division by zero");
  return a/e;
}
```

Функцию `assert()` удобно использовать для отладки программы. В окончательной версии отладочный код обычно отключается директивой `#define NDEBUG`

Для реализации третьего варианта используется функция с побочным эффектом. Так, например, заголовок функции, вычисляющей целое частное двух целых чисел,

```
int calc(int a, int e);
```

заменяем следующим заголовком:

```
bool calc (int a, int e, int &res);
```

При этом реализация функции выглядит следующим образом:

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

Как использовать такую функцию? Один из вариантов вызова может выглядеть следующим образом:

```
if (calc(a,e,r))
  cout<<r;
else
  cout<<”ERROR”;
```

Синтаксис C++ позволяет использовать вызов функции, игнорируя возвращаемое значение.

```
calc(a,e,r);
cout<<r;
```

В этом случае ошибка, обнаруженная в функции, но не обработанная при вызове, может привести к непредсказуемым последствиям.

Программисты часто игнорируют информацию об ошибках, потому что проверять все возможные ошибки после каждого вызова функции было бы слишком утомительно и неудобно.

Объектно-ориентированный подход использует другой способ обработки ошибок времени выполнения — *механизм обработки исключений*. Он является одной из важнейших возможностей C++: при возникновении критической ошибки функция создает *объект исключения*. Для разных ошибок можно создавать разные типы объектов, для каждого из которых можно предусмотреть *обработчик*.

Благодаря механизму обработки исключений при возникновении ошибки, можно прервать выполнение и передать управление в другую часть программы вместе с информацией об ошибке.

Оператор генерации исключения

```
throw <исключение>;
```

В этом операторе исключение может быть объектом произвольного типа, в том числе и встроенного. Но обычно для этих целей используются специальные классы.

Оператор `throw` создает объект исключения и может завершить выполнение функции, в которой возникла ошибка. При этом объект исключения возвращается как результат функции, даже если тип этого объекта не соответствует типу функции.

```
int calc (int a, int e) {
  if (e) {
    return a/e;
  }
  throw "Ошибка вычисления";
}
```

Теперь при вызове функции нужно учесть тот факт, что она может или вернуть результат вычислений, или выбросить исключение. Поэтому, чтобы предусмотреть обработку исключений, выполняемый код, в котором они могут возникнуть, помещается в блок `try`.

```
try {
// Программный код, который может генерировать исключения
}
```

Обработка исключений производится после блока `try`. Это позволяет основному коду не смешиваться с кодом обработки ошибок.

Блок, где программа должна среагировать на сгенерированное исключение, называется обработчиком исключения. Для каждого типа перехватываемого исключения должен быть свой обработчик. Обработчики исключений следуют сразу же за блоком `try` и обозначаются ключевым словом `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...
```

По своему синтаксису секции `catch` напоминают функции, вызываемые с одним аргументом. Идентификатор (`id1, id2, …, idN`) может использоваться внутри обработчика по аналогии с аргументом функции, но если он не нужен — его можно опустить. Обычно для обработки исключения достаточно знать имя типа.

```
int main (){
  int a,e;
  cin>>a>>e;
  try {
    cout << calc(a,e);
  }
  catch (char *s) {
    cout << s<<endl;
  // что делать в случае ошибки?
  }
  return 0;
}
```

Конечно, для большинства исключительных ситуаций обработчик должен сообщить подробную информацию о возникшей проблеме (вывести на экран, записать в файл) и корректно завершить выполнение программы (закрыть файлы, освободить динамческую память).

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

Но существуют задачи, в которых можно предложить пользователю возможность исправить ошибочную ситуацию (ввести данные заново) и продолжить выполнение программы.

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

Рассмотрим ситуацию, когда в `try`-блоке вызывается функция, которая сама не генерирует исключение, но при этом содержит вызов другой функции, которая в свою очередь либо генерирует исключение, либо содержит вызов третьей функции и т.д.

При генерации исключения выполнение будет передано от функции, сгенерировавшей исключение, в функцию, содержащую блок `try` и блоки `catch`. На рисунках 3 и 4 продемонстрировано различие в последовательности действий при нормальном завершении цепочки вызовов из трех функций и при генерации исключения в функции `f3()`.

И в том и другом случае происходит освобождение стека вызовов функций. При возникновении исключительных ситуаций, как и вслучае нормального возврата, все локальные объекты, созданные в процессе цепочки вызовов функций к моменту запуска исключения, уничтожаются. Автоматическое уничтожение локальных объектов часто называется «раскруткой стека».

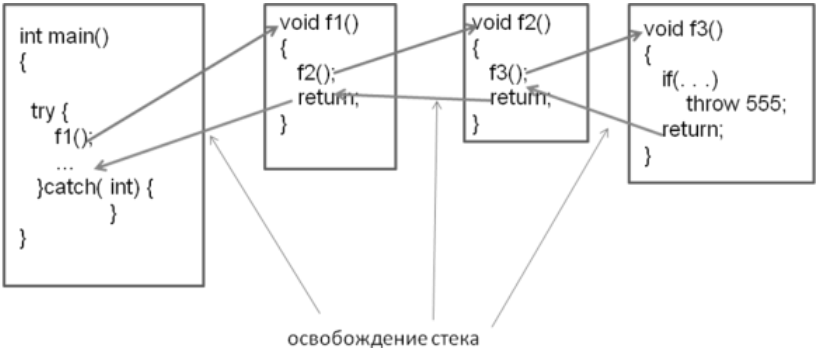


Рис.3. Нормальное завершение вызова функции f1

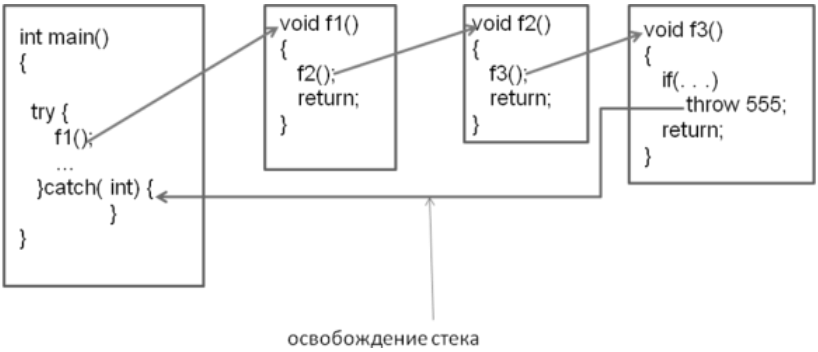


Рис.4. Завершение вызова функции f1 при возникновении исключительной ситуации

1.16. Рекурсия

Задача может быть решена рекурсивно, если ее можно разложить на совокупность подзадач того же типа, но имеющих меньшую размерность.

Никаких особенностей синтаксиса для рекурсивных функций в языках C и C++ нет. Как и в любом языке необходимо обращать внимание на наличие условия для выхода из рекурсии.

Пример 19. Реализовать рекурсивную функцию эффективного возведения вещественного числа в целую степень.

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

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

double sqr (double x){
  return x*x;
}

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));
}
```

На этом примере видно, что рекурсивная реализация функции близка к записи математической формулы.

Пример 20. Для вводимой последовательности ненулевых целых чисел, за которыми следует 0, выдать сначала все отрицательные, затем все положительные. При решении не использовать дополнительные структуры данных для хранения вводимых значений.

Чтобы вводимые и выводимые значения не смешивались, всю последовательность следует задавать сразу через пробел.

```
#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<<"Введите через пробел последовательность ненулевых ";
  cout<<"целых чисел, за которыми следует 0"<<endl;
  print();
  cout<<endl;
  return 0;
}
```

В функции `print()` вывод отрицательных чисел реализуется при рекурсивном спуске, а положительных — на рекурсивном подъеме.

Пример 21. Используя рекурсивную функцию, вычислить значение, задаваемое формулой. Формула вводится с клавиатуры посимвольно. Правила записи формулы определяются следующим образом:

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

Будем предполагать, что формула является синтаксически правильной, содержит только цифры и знаки, перечисленные в правилах, а также круглые скобки. При этом необходимо предусмотреть обработку возможной семантической ошибки — деления на ноль. Для обработки использовать механизм исключительных ситуаций.

```
#include <iostream>

using namespace std;

int formula() {
  char c, op;
  int x,y;
  cin>>c;
  if (c>='0' && c<='9') //цифра
    return c-'0';

  //начало формулы вида (x op y)
  x=formula();
  cin>>op;
```

```
  y=formula();
  cin>>c; // пропуск закрывающейся скобки
  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)
    try{
      cout<<formula()<<endl;
    }
    catch (int err) {
      if (err==-55)
        cout<<"Ошибка деления на ноль"<<endl;
      else
        cout<<"Ошибка в знаке"<<endl;
    }
  return 0;
}
```

На рисунке 5 представлен вариант запуска программы.

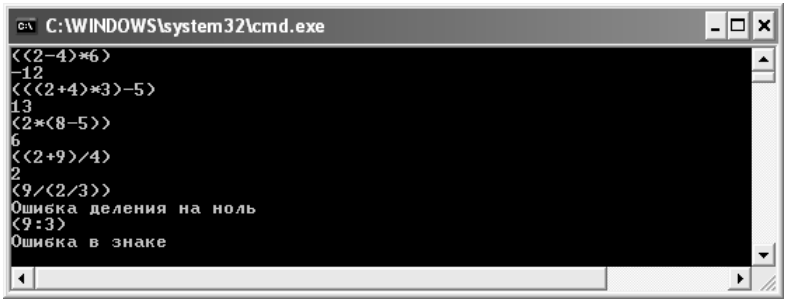


Рис. 5. Результат запуска программы

Для обработки ошибки деления на ноль использован механизм исключений.

```
case '/': if (y!=0)
            return x/y;
          else
            throw (-55);
```

В функции предусмотрена обработка всех символов, синтаксически допустимых в правильной формуле. Но так как компилятору не известны допустимые в формуле символы, он предупрежает о том, что не во всех случаях функция возвращает результат. Поэтому предусмотрена еще одна исключительная ситуация вместо оператора `return`.

```
throw(-1);
```

В обоих случаях в качестве типа исключительной ситуации используется целый тип. Чтобы 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;
  }
```

МОДУЛЬ 2. МАССИВЫ, СТРОКИ И ФУНКЦИИ

2.1. Одномерные массивы

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

```
<тип> <имя>[<размерность>];
```

Размер массива фиксирован на все время существования в памяти и не может быть изменен. Имя массива интерпретируется как константный указатель на адрес его размещения в памяти (первый элемент). Доступ к элементам массива производится с помощью операции индексации [ ].

Пример 22. Дан целочисленный массив, содержащий не более 10 элементов. Найти номер первого нулевого элемента в массиве.

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

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

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

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

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

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

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

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

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

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

Другой способ описания параметров-массивов в функции связан с использованием указателей:

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

 Изменить в примере 22 заголовок функции так, чтобы для передачи массива использовался указатель.

Пример 23. Написать программу, подсчитывающую во вводимой последовательности символов по отдельности каждую цифру, пробельные

литеры (пробелы, табуляции и новые строки) и все другие литеры, кроме «*». Признаком окончания последовательности является символ «*».

Вот один из вариантов решения:

```
#include <iostream>
using namespace std;
void main () {
  int i, nwhite, nother;
  char c;
  int ndigit [10];
  nwhite = nother = 0;
  for (i = 0; i < 10; ++i)
    ndigit [i] = 0;
  cin.get(c);
  while (c != '*') {
    if ((c >= '0') && (c <= '9'))
      ++ndigit [c-'0'];
    else if ((c ==' ') || (c == '\n') || (c == '\t'))
      ++nwhite;
    else
      ++nother;
    cin.get(c);
  }
  cout<<"digits = "<<endl;
  for (i = 0; i < 10; ++i)
      cout<<i<<'-'<<ndigit[i]<<' '<<endl;
  cout<<" white = "<<nwhite<<" other = "<<nother<<endl;
}
```

В рассмотренном примере имеется двенадцать категорий вводимых литер. Удобно десять счетчиков цифр хранить в массиве, а не в виде десяти отдельных переменных.

При определении значения индекса `ndigit [c-'0']` в массиве счетчиков цифр используется то обстоятельство, что тип `char` является целочисленным и не требует преобразования типов. Для двух оставшихся категорий литер используются отдельные счетчики.

2.2. Массивы в динамической памяти

Если массив создается путем объявления, то память под него будет отведена перед началом выполнения программы или функции и будет за-

креплена за массивом до завершения времени жизни массива. Оператор new позволяет выделять память под массив во время выполнения программы, причем указывая его размер не «по максимуму», а необходимый при конкретном выполнении программы.

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

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

```
delete[] a;
```

Пример 24. Вычислить среднее арифметическое элементов вещественного массива.

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

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

При передаче массива как параметра в функцию используется указатель:

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

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

2.3. Связь массивов и указателей

Неважно, как создан массив, имя массива — это адрес (указатель) первого по счету элемента массива `a[0]`. Над указателями определены: адресная арифметика, операции присваивания и сравнения.

Операции адресной арифметики: +, –, ++, –– для указателей увеличивают или уменьшают значение указателя на величину, кратную размеру адресуемого элемента данных. Такие операции используются при работе с массивами, поскольку элементы массива расположены в памяти последовательно. Если имеется два указателя на элементы одного массива, то разность этих указателей будет равна количеству элементов массива, расположенных между этими указателями.

Указатели можно сравнивать на равенство/неравенство. Остальные операции сравнения имеют смысл применительно к указателям на элементы одного и того же массива.

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

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

Пример 25. Проверить, является ли массив целых чисел симметричным относительно своей середины.

```
#include <iostream>
using namespace std;
bool simm( int *a, int n );
int main() {
  int *a, n;
  cout <<" The size of the array ";
  cin>>n;
  a=new int[n];
  for (int i=0; i<n; ++i) {
    cout<<i<<" element ";
    cin>>a[i];
  }
  cout<<simm(a,n)<<endl;
  delete []a;
  return 0;
}
bool simm( int *a, int n ) {
  int *b=a+n-1;
  while (a<=b)
    if (*a++!=*b--)
      return false;
  return true;
}
```

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

Пример 26. Описать функцию, вычисляющую сумму элементов массива, продемонстрировать ее использование для разных сегментов массива.

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

```
    <<sum_arr(array+n-4,array+n)<<endl;
  cout<<"The sum from 3 to 7 ="<<sum_arr(array+2,array+7) <<endl;
  return 0;
}
int sum_arr( const int *begin, const int *end) {
  const int *pt;
  int sum=0;
  for (pt=begin; pt!=end; ++pt)
    sum+=*pt;
  return sum;
}
```

Традиционный способ передачи параметров-массивов в функции заключается в передаче указателя на начало массива и размера массива. Существует еще один метод предоставления необходимой информации функции — указание диапазона элементов массива.

Диапазон задается полуоткрытым справа интервалом `[*begin;*end)`. Первый указатель определяет начало диапазона элементов массива, второй — его конец.

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

Просмотр всех элементов диапазона организован с помощью цикла

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

Условие `pt!=end` основано на том, что правая граница диапазона является указателем на элемент, следующий за последним элементом диапазона.

> 📖 Левая граница диапазона — это указатель на первый элемент диапазона, правая граница — это указатель на элемент, следующий за последним элементом диапазона.

Весь массив можно рассматривать как частный случай диапазона. В этом случае правую границу диапазона можно вычислить, добавив к адресу массива количество его элементов. Полученное значение адреса указывает за границу массива.

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

Для нахождения суммы с третьего элемента по седьмой используется следующий вызов.

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

 Реализация функции `sum_arr` предполагает корректное указание диапазона при ее вызове (`begin<=end`). Дополните функцию проверкой на корректность передаваемых параметров.

2.4. Массивы и рекурсия

Алгоритмы для массивов рекурсивны по своей сути. Это вытекает из возможности рекурсивного определения массива. Поэлементную обработку массива можно разделить на обработку одного элемента и поэлементую обработку остальных элементов. Так, например, сумма `n` элементов массива может быть сведена к задаче вычисления суммы первых `n-1` элементов и `n`-го элемента. Это продемонстрировано в следующих примерах.

Пример 27. Описать рекурсивные функции вычисления суммы элементов массива, используя разные списки параметров для передачи массива.

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

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

int summa(int*bg, int*end) {
  if ( bg == end)
    return 0;
    return summa(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<<summa(a1,6)<<endl; //все элементы массива
  cout<<summa(a2,9)<<endl; //все элементы массива
  cout<<summa(a1,3)<<endl; //первые три элемента массива
  cout<<summa(a1+3,3)<<endl; //последние три элемента массива

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

В первом случае использована передача массива через указатель на начало массива и количество элементов.

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

Соотвествующие вызовы:

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

Во втором случае сегмент массива передается двумя указателями. Первый указатель определяет начало сегмента, второй — конец сегмента (за последним элементом диапазона).

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

Соотвествующие вызовы:

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

Пример 28. Описать рекурсивные функции нахождения максимального элемента и определения количества нулевых элементов массива.

```
int max(int* a, int n) {
```

```
  if ( n==1 )
    return a[0];
  int m = max(a,n-1);
  return a[n-1]>m ? a[n-1] : m;
}

int kol0 (int *a, int n) {
  if ( n==0 )
    return 0;
  if (a[n-1]==0)
    return 1+kol0(a,n-1);
  return kol0 (a,n-1);
}
```

Особенностью реализации функции `max` является необходимость сохранять результат рекурсивного вызова функции в локальную переменную.

Попытка обойтись без использования локальной переменной

```
// return a[n-1]> max(a,n-1) ? a[n-1] : max(a,n-1);
```

приводит к удвоению числа рекурсивных вызовов с одними и теми же значениями параметров на каждом шаге.

В функции `kol0` также присутствуют два одинаковых рекурсивных вызова, но они находятся в разных ветках условного оператора. Поэтому нет необходимости использовать локальную переменную.

2.5. Статическое определение двумерных массивов

Специального типа многомерного массива (в том числе и двумерного) в C++ не существует. Но можно определить массив, каждый элемент которого, в свою очередь, является массивом. Для двумерного массива определение выглядит следующим образом:

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

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

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

Аналогичным образом можно объявлять любые многомерные массивы. Как и в случае одномерных массивов, возможна начальная инициализация:

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

При инициализации разрешается оставлять неопределенной только первую размерность:

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

Аналогичное правило распространяется и на описание многомерных массивов, являющихся параметрами функций. Для двумерного массива или массива с большим количеством размерностей первую размерность можно оставить неопределенной. Остальные размерности должны быть заданы:

```
void f1 (int a[][10],int n,int m);
```

Объявление параметра `a` функции `f1` эквивалентно объявлению указателя на массив, каждый элемент которого является массивом из 10 элементов целого типа:

```
void f1 (int (*a)[10],int n,int m);
```

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

Пример 29. Написать функцию, выводящую построчно матрицу целых чисел, если размер строки матрицы при объявлении равен 10.

```
void print (int p[][10], int n,int m) {
  int j;
  for (int i=0; i<n; ++i) {
    for (j=0; j<m; ++j)
      cout<<p[i][j]<<"  ";
    cout<<endl;
  }
}
```

Из-за того, что количество столбцов матрицы при описании параметров функции указано явно, для матриц с разным количеством столбцов придется описывать разные функции.

Важно понимать, что статические массивы любой размерности занимают в памяти непрерывный участок, размер которого вычисляется исходя из величины каждого измерения. Эти же величины используются при определении положения элемента массива, когда происходит обращение к нему с помощью индексного выражения. При этом имя многомерного массива по-прежнему является адресом его первого элемента.

Пример 30. Написать функцию, выводящую все элементы матрицы размера nx2, не используя доступ к элементам матрицы через индексное выражение.

```
#include <iostream>

using namespace std;

void print1(int b[][2],int n){
  int *c=&b[0][0];
  for (int i=0;i<n*2;++i)
    cout<<*c++<<' ';     //допустимо использование индекса c[i]
  cout<<endl;
}

int main() {
  int a[2][2]= {{1,2},{3,4}};
  int b[3][2]= {{1,2},{3,4},{5,6}};
  print1(a,2);
  print1(b,3);
  return 0;
}
```

В этом примере используется линейное расположение в памяти элементов матрицы по строкам. Количество столбцов учитывается при переходе от двойного индекса к одинарному.

Пример 31. Найти первое вхождение элемента с заданным значением в целочисленной матрице. Функция поиска должна возвратить индексы первого вхождения элемента, если такой элемент имеется в матрице.

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

```
bool findElem(int a[][10],int n,int m,int x,int &ki,int &kj){
  int i=0;
  int j=0;
  while (i<n && a[i][j]!=x) {
    j++;
    if (j==m) {
      j=0;
      i++;
    }
  }
  if (i<n) {
    ki=i;
    kj=j;
  }
  return i<n;
}

void input(int a[][10], 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 b[10][10];
  int ki,kj,n,m,x;
  cin>>n>>m;
  input(b,n,m);
  cin>>x;
  bool f=findElem(b,n,m,x,ki,kj);
  if (f)
    cout<<ki<<' '<<kj<<endl;
  else
    cout<<"elem not found"<<endl;
  return 0;
}
```

В основе алгоритма лежит решение, предложенное Д. Грисом. Особенностью этого алгоритма является использование одного цикла, в заголовке которого находится условие нахождениия элемента с заданным значением. Наличие условия `i<n` вызвано тем, что элемент с заданным значением может отсутствовать в матрице:

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

Для реализации перехода от строки к строке используется оператор:

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

Условие `i<n` после завершения цикла является признаком того, что элемент найден.

2.6. Двумерные массивы в динамической памяти

Указатель на массив указателей позволяет смоделировать в динамической памяти структуру данных аналогичную двумерному массиву — матрице. Для этого необходимо объявить переменную типа «указатель на указатель».

Например, если массив будет содержать целые значения, то описание выглядит следующим образом:

```
int **a;
```

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

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

После этого выделяется память для каждой строки:

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

На рисунке 6 приведена модель организации двумерного массива в динамической памяти.

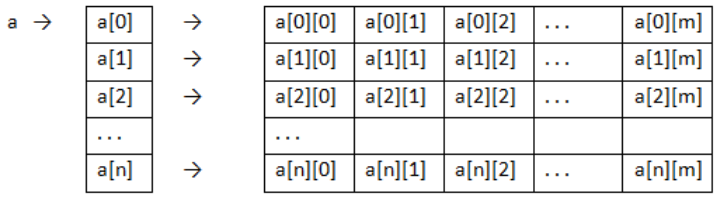


Рис. 6. Организация двумерного массива в динамической памяти

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

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

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

Пример 32. Создать и распечатать верхнетреугольную матрицу следующего вида:

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

В решении необходимо использовать экономное выделение памяти — только для элементов, расположенных на главной диагонали и выше.

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

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

```
    for (j=0; j<m; ++j)
      cout<<matr[i][j]<<" ";
    cout<<endl;
  }
  for (i=0; i<n; ++i)
    delete []matr[i];
  delete[] matr;
  return 0;
}
```

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

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

 Создать и распечатать нижнетреугольную матрицу, заполнив ее аналогично матрице из примера 32.

Пример 33. Написать функцию проверяющую, является ли квадратная матрица симметричной. Дополнительно определить функции для создания матрицы заданных размеров и заполнения ее с клавиатуры, для печати матрицы и для освобождения памяти, занимаемой матрицей.

```
#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;
}

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;
}
```

Организация двумерного массива в динамической памяти позволяет передавать его в качестве параметра функции как 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);
```

Описание параметра функции для двумерного массива как указатель на указатель позволяет разрабатывать универсальные функции работы с двумерными массивами. Однако фактическим параметром при вызове тоже должен быть указатель на указатель.

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

Этому описанию не соответствуют статические двумерные массивы. Чтобы передавать такие массиве как параметры в функцию необходимо создать вспомогательный массив указателей на строки матрицы. Например, чтобы распечатать двумерный статический массив `aa[2][3]`функцией `print` следует поступить так:

```
int aa[2][3];
int *p[2];
p[0]=&aa[0][0];
p[1]=&aa[1][0];
print (p,2,3);
```

В функции `create(int**& a,int n,int m)`, в отличие от остальных, параметр `a` (указатель на указатель) передается по ссылке. Это связано с тем, что его значение при выделении памяти в функции меняется. Несмотря на то, что функция `free(int** a,int n)` тоже работает с

памятью, при освобождении памяти значение параметра a не меняется. Поэтому передавать этот параметр по ссылке не требуется.

При освобождениие памяти, занимаемой двумерным массивом, требуется информация только о количестве строк. Это позволяет указывать в списке параметров только одну размерность.

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

Перегруженный вариант функции `create` демонстрирует возможность языка C++ возвращать адрес создаваемой динамически структуры данных.

```
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;
}
```

Это позволяет не использовать передачу по ссылке параметра для сложных структур данных. При этом вызов функции `create` внешне напоминает применение операции размещения в динамической памяти.

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

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

2.7. Сортировки массивов

Под сортировкой массива понимают процесс перестановки элементов массивов в определенном порядке. Цель сортировки — облегчить после-

дущую обработку массива, в частности, поиск элементов. С методами сортировок связаны многие фундаментальные приемы построения алгоритмов.

Основные требования к методам сортировок — экономия памяти и быстродействие. Первое требование может быть выполнено, если переупорядочивание будет выполняться без создания вспомогательного массива. Быстродействие зависит от количества выполняемых операций — сравнений и перестановок.

С этой точки зрения выделяют три простых метода сортировки, требующих порядка n^2 операций, и улучшенные методы, порядок операций которых стремится к $n \cdot \log_2 n$.

Пример 34. Написать функции, реализующие алгоритмы трех простых методов сортировки массивов (выбором, обменом, включением).

Сортировка выбором основана на идее многократного выбора максимального (минимального) элемента и перемещения его в начало (конец) массива.

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

Сортировка обменом (метод «пузырька») основана на систематическом обмене значениями пар стоящих рядом элементов, для которых нарушено условие упорядоченности. Процесс обмена продолжается до тех пор, пока таких пар не останется.

```
void bubbleSort (double * a, int n) {
  bool f = true;
  int i = n-1;
  int j;
```

```
  while (f) {
    f = false;
    for (j  = 0; j < i; ++j)
      if (a[ j ] > a[j+1]) {
        swap (a[ j ], a[j+1]);
        f = true;
      }
      i = j;
    }
}
```

Сортировка включением основана на включении одного элемента в упорядоченный набор с сохранением упорядоченности. Известно, что массив из одного элемента всегда отсортирован.

```
void insertionSort (double * a, int n) {
  double x; int j;
  for (int i = 1; i <n; ++i) {
    x = a[i];
    j = i-1;
    while (j > =0 && a[ j ] > x) {
      a[j+1] = a[j];
      j --;
    }
    a[j+1] = x;
  }
}
```

Реализуем перегрузку функции `insertionSort` для случая параметров, задающих сортируемый сегмент массива посредством указателей.

```
void insertionSort (double * a, double * b) {
  double x; double *d;
  for (double *c=a+1; c<b; ++c) {
    x =*c;
    d = c-1;
    while (d >=a && *d > x) {
      *(d+1) = *d;
      d--;
    }
    *(d+1) = x;
  }
}
```

Пример 35. Написать функцию, реализующую сортировку Хоара.

Сортировка, носящая имя А. Хоара часто называется быстрой сортировкой. Быстрая сортировка является улучшенным методом сортировки

обменом. Хоаром было отмечено, что для достижения большей эффективности необходимо производить обмены на больших расстояниях.

Алгоритм состоит из двух этапов.

- Выполнение перестановки элементов таким образом, чтобы массив разделился на две части относительно некоторого элемента, называемого опорным. Левая часть после перестановки будет содержать элементы меньшие или равные опорному, правая часть — большие или равные опорному.
- Рекурсивное применение этого алгоритма к каждой из частей, пока размер сортируемой части больше единицы.

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

Реализуем перегрузку функции `qSort` для случая параметров, задающих сортируемый сегмент массива посредством указателей.

```
void qSort(double*a, double *b) {
  double x = *(a+(b-a)/2);  // x - опорный
  double *p=a;
  double *q=b-1;
  do {
    while(*p < x) ++p;  // поиск элемента для переноса
    while(*q > x) --q;  // поиск элемента для переноса
    if (p <= q){        // обмен элементов местами:
      swap(*p,*q);
      ++p; --q;         // переход к следующим элементам:
    }
  } while(p < q);
```

```
  if (a < q)  qSort(a, q+1);
  if (p < b-1) qSort(p, b);
}
```

Из рекурсивной природы быстрой сортировки следует большое количество вызовов, поэтому для небольших массивов она будет давать худшие результаты по сравнению с простыми методами.

На практике рекомендуется учитывать размер сортируемого сегмента, и при уменьшении его ниже некоторой границы заменять вызов рекурсивной сортовки на один из простых методов сортировки. Например, сортировки включения, как показано в функции `qSortWithCut`.

```
void qSortWithCut(double*a, double *b) {
  if (b-a<CUTOFF) {
    insertionSort(a,b);
    return;
  }
  double x = *(a+(b-a)/2);  // x - опорный
  double *p=a;
  double *q=b-1;
  do {
    while(*p < x) ++p;  // поиск элемента для переноса
    while(*q > x) --q;  // поиск элемента для переноса
    if (p <= q){        // обмен элементов местами:
      swap(*p,*q);
      ++p; --q;         // переход к следующим элементам:
    }
  } while(p < q);
  if (a < q)  qSort(a, q+1);
  if (p < b-1) qSort(p, b);
}
```

Условие `(b-a<CUTOFF)` определяет случай использования простой сортировки. Для этого сравнивается размер сортируемой части массива с константой `CUTOFF`.

Пример 36. Применить индексную сортировку для упорядочивания строк матрицы по возрастанию сумм их элементов.

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

```
int main() {
  int matr[10][10];
  int n, m;
  int ind[10];
  cout<<"size matrix ";
  cin>>n>>m;
  for (int i=0; i<n; ++i) {
    cout<<"row "<<i<<" ";
    for(int j=0; j<m; ++j) {
      cin>>matr[i][j];
    }
  }
  ind_sort(matr,n,m,ind);
  for (int i=0; i<n; ++i) {
    cout<<endl;
    for(int j=0; j<m; ++j) {
      cout<<setw(4)<<matr[ind[i]][j];
    }
  }
  return 0;
}

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

Обмен значениями двух строк двумерного массива требует поэлементного обмена, что приводит к увеличению времени выполнения про-

граммы. Для повышения эффективности использован вспомогательный массив индексов, который отражает размещение строк в отсортированной матрице. Такая сортировка называется индексной.

Для индексной сортировки помимо сортируемого массива характерно использование массива ключей и массива индексов. Массив ключей содержит значения, по которым производится сортировка. Он может отсутствовать, если ключи хранятся в самом сортируемом массиве. Массив индексов на каждом шаге отражает размещение элементов в сортируемом массиве. Процесс сортировки заключается в сравнении ключей (доступ к ним осуществляется через индекс) и перестановке индексов.

Индексная сортировка реализована в виде функции

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

В качестве параметров передаются матрица, ее размеры и выходной параметр `int ind[ ]` для массива индексов. Ключами являются суммы элементов строк матрицы. Поэтому для хранения ключей организован массив `int sum[10]`.

Для доступа к элеметам отсортированной матрицы следует использовать в качестве номера строки соответствующий элемент индексного массива:

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

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

Пример 37. 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 создается, 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. 7).

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

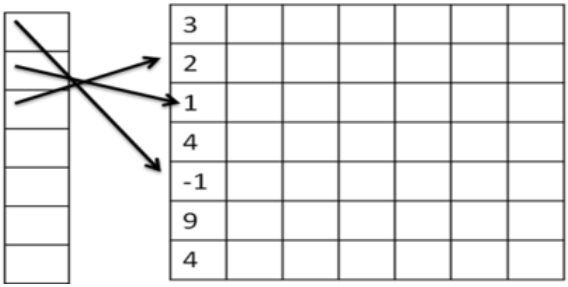


Рис.7. Перестановка указателей на строки матрицы.

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

2.8. Указатели на функции

В языке C++ аналогом процедурного типа языка Паскаль является тип — указатель на функцию, или функциональный тип.

Рассмотрим пример объявления:

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

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

Если в объявлении убрать круглые скобки:

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

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

Приоритет префиксного оператора `*` ниже, чем приоритет `()`, поэтому для объявления указателя на функцию скобки необходимы.

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 поочереди символы, то справа, то слева. 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 звездочка (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 прием «движения изнутри наружу чередуя направо и налево».

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, котрая возвращает указатель на массив из 10 указателей на функции, возвращающие значения типа int.

 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
```

Язык C++ предоставляет средство, позволяющее давать новые имена (синонимы) типам данных с помощью конструкции typedef. Например, объявление

```
typedef unsigned short UShort;
```

делает имя UShort синонимом unsigned short. С этого момента тип UShort можно применять точно так же как тип unsigned short.

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

Обявление

```
typedef double (*(*(*fp4)())[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.

Указателю на функцию, также как и любой другой переменной, перед дальнейшим использованием необходимо присвоить значение — адрес

функции. Адрес функции задается именем функции без списка аргументов и без круглых скобок.

Например, имеется функция `int func(int,double)`. Тогда ее адрес — `func`. Допустим и более очевидный синтаксис `&func()`.

Пример 38. Описать функцию `tabulat` построения таблицы значений для произвольной вещественной функции `f(x)`, где `x` — вещественная переменная. Таблица должна строиться на отрезке `[a,b]` с заданным шагом `h`. Использовать функцию `tabulat` для получения таблиц значений нескольких вещественных функций одной вещественной переменной.

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

const double eps=0.00001;
typedef double (*fType)(double x);

double f1(double x){
  return exp(x)*sin(x);
}

double f2(double x){
  return x*x+0.5*x-7;
}

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

void tabulat(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;
  tabulat(-2,2,0.5,f1);
  cout<<endl<<endl;
  cout<<"f(x)=x*x+0.5*x-7"<<endl;
  tabulat(-2,2,0.5,f2);
  cout<<endl<<endl;
  cout<<"f(x)=1/x"<<endl;
  tabulat(-2,2,0.5,f3);
  return 0;
}
```

Для вещественной функции одной вещественной переменной определен тип `fType`:

```
typedef double (*fType)(double x);
```

Этот тип использован при описании параметра `f` в функции `tabulat`:

```
void tabulat(double a,double b,double h,fType f)
```

Функции `f1`, `f2`, `f3`, соответствующие типу `fType`, используются в качестве фактических параметров при вызовах функции `tabulat`.

Результат выполнения представлен на рисунке 8.

Использование механизма исключений позволило выполнять табулирование функций с особенностями. Для этого соотвествующая функция в точках, где она не определена, должна выбрасывать исключения типа `int`.

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

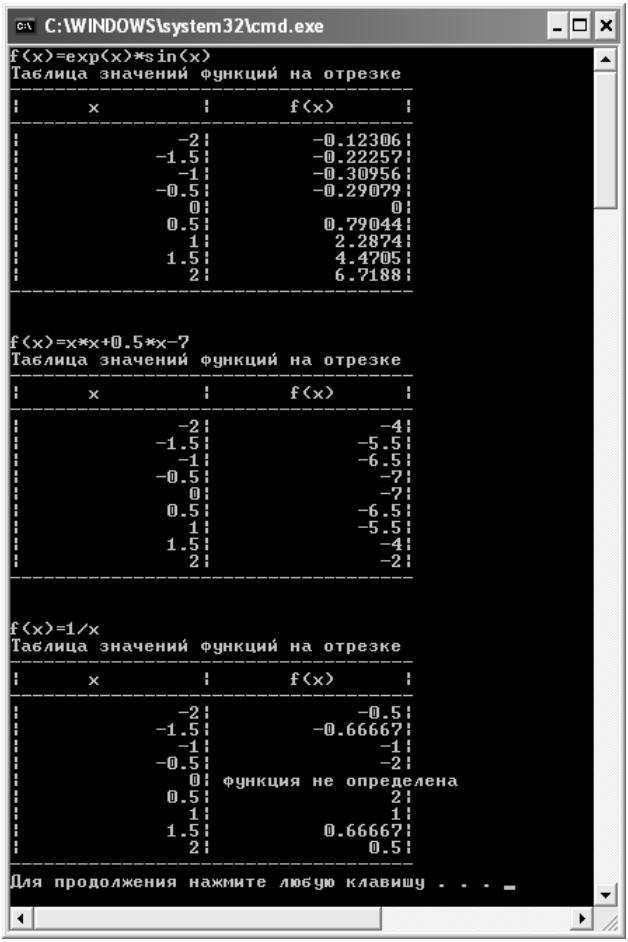


Рис. 8. Результат вызовов функции `tabulat`

Пример 39. 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 max(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 "<<max(a,12,isSimple)<<endl;
  cout<<"max pos "<<max(a,12,pos)<<endl;
  return 0;
}
```

Два введенных 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 `max` реализует алгоритм нахождения максимального элемента в массиве, удовлетворяющего заданному условию. Поскольку условие определяется как параметр-предикат, то при вызове функции можно использовать любые условия, офрмленные как соответствующие функции.

```
cout<<"max isSimple "<<max(a,12,isSimple)<<endl;
cout<<"max pos "<<max(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.

2.9. 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 нее 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 определен.

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 следующей программе:

```
#include <iostream>

using namespace std;

int main() {
  char s3[] = "String";
  char* s4 = "String";
  s3[1]='A';
  cout<<s3<<endl;
  //s4[1]='A'; – ошибка времени выполнения
  //cout<<s4<<endl;
  //cout<<*(++s3)<<endl; – ошибка компиляции
  cout<<*(++s4)<<endl;
  return 0;
}
```

Если в случае ошибки компиляции

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

необходимо исправить ее 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.

Распространенная ошибка при попытке описания строки:

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

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

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

Доступ к строке обычно осуществляется с помощью указателя `char*`. Поэтому, как правило, тип `char*` ассоциируется именно со стро-

кой. Так, если переменная `s` имеет тип `char*`, то при выполнении оператора

```
cout<<s;
```

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

Оператор >> позволяет ввести слово:

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

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

Рассмотрим распространенные ошибки при вводе строковых данных.

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

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

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

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

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

Для использования setw необходимо подключить заголовочный файл iomanip. Использование манипулятора влияет только на непосредственно следующую за 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");  //Еще один пример инициализации
  string s5(s3);  //И еще один пример инициализации
  s2 = "By";  //Присваивание
  s1 =s3 + " " + s4;  //Слияние строк – конкатенация
  s1 += " Good ";  //Присоединение новых символов к строке
  cout  << s1 + s2 + "!" << endl;
}
```

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

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

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

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

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

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

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

➢ Следует обратить внимание на то, что:

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

✓ операция присваивания для строк в стиле С++ присваивает строки, а в стиле С – только указатели.

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

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


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

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

2.10. Обработка строк в стиле языка C

Поскольку строка в стиле C завершается нулевым символом, ее обработка имеет определенную специфику. Рассмотрим ее 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'` нулем.

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

Поскольку в языке C 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 приведем 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 C, 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 C: `strlen`, `strcpy`, `strcmp`, `strcat`.

☠ Распространенная 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* max(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 << max("Hello", "By") << endl;
  cout << max(s1, s2) << endl;
  sMax = max(s1, s2);
  return 0;
}
```

Функция `max` использует функцию `strcmp` из стандартной библиотеки cstring:

```
strcmp(const char*string1,const char*string2)
```

Строки сравниваются в *лексикографическом* порядке. Результат сравнения определяется по правилам, приведенным 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 ее 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 прием 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, разделенных 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 четные и нечетные слова и записать результат в новую строку.

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

В данном алгоритме будут несколько раз встречаться следующие циклы:

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

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

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

```
void swap_even(const char *p, char *ps) {
  const char *p1;
  while (*p == ' ') p++; //пропустить начальные пробелы
  do {
    p1= p;//запомнить адрес нечетного слова
```

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

 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` соотвественно 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 измененные данные.

В силу особенностей представления строк в стиле С, 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 остается 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 ис-

ходной и результирующей строками не требует ни изменения содержимого, ни удаления строки с данными. Это сязано 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 рис. 9 продемонстрирован результат вызова функции `TrimLeftC` для входной строки «□□HELLO□□» с точки зрения размещения в памяти.

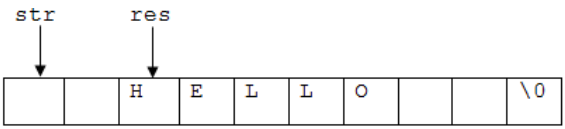


Рис. 9. Результат выполнения

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 путем сдвига влево. Этот вариант работает дольше, но решение полностью соотвествует постановке задачи.

```
void TrimLeftCInPlace(char* S, char C = ' '){
  char *p = S;
  while (*p != 0 && *p == C)
    p++;
  while (*S++ = *p++);
}
```

2.11. 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

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

//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`. Ее значение равно макси-

мально 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 произведенной замены, на тот случай, если `from` является подстрокой `to`. Поэтому значение переменной `lookHere` важно увеличить на длину заменяющей строки `to`.

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

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

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

Одно из самых ценных и удобных свойств строк C++ состоит в том, что они могут автоматически изменять размер по мере надобности, не требуя вмешательства со стороны программиста. Работа со строками становится не только более надежна, из нее практически полностью устраняются «нетворческие» операции — отслеживание границ памяти, в которой хранятся строковые данные.

Пример 49. Описать функцию, классифицирующую строку как идентификатор, целое число, число с плавающей точкой или неопределенную 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)
      return ident;//только допустимые символы
    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)
    return error; //число должно начинаться с цифры
  pos = s.find_first_not_of(digits, pos); //конец цепочки цифр
  if (pos == string::npos)
    return integer; //одни цифры - целое число
  else if (s[pos] == '.')
  {
    pos = s.find_first_not_of(digits, pos + 1);
    //конец цепочки цифр
    if (pos == string::npos) return floating;
  }
  //может быть экспонента
  if (s[pos] == 'e' || s[pos] == 'E') {
    if (pos == string::npos)
      return error; // символ E последний
    if (s[pos+1] == '+' || s[pos+1] == '-') ++pos;
    //пропускаем
```

```
    if (pos == s.length() - 1)
      return error;
      //знак -последний символ строки
    pos = s.find_first_not_of(digits, pos + 1);
    //конец цепочки цифр
    if (pos == string::npos)
      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 завершеатся неудачно, возвращается `string::npos`.

МОДУЛЬ 3. СТРУКТУРЫ, ФАЙЛЫ И СПИСКИ

3.1. 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 ее членом. Это имя используется для доступа с помощью операции точка (.).

Структуры можно рассматривать 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 50. 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 разменования `*`. Для упрощения записи используется постфиксная операция стрелка (`->`), которая позволяет обращаться к членам структуры через указатель. Таким образом, обозначение `holiday->monthName` равносильно приведенному 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 51. 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, передается в нее 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 ее пределы. При этом определение функций-членов за пределами структуры требует уточнения имени функции именем структуры с помощью операции разрешения области видимости.

```
int date::daysInMonth(){
…
}
```

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 определенным над ним набором операций.

3.2. Ввод/вывод и работа с файлами

В языки С и С++ функции ввода/вывода не встроены. Первоначально в языке С 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 С.

При запуске С-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` соотнесен с клавиатурой, а `stdout` и `stderr` — с экраном. Однако их можно

связать с файлами. Часто вывод в stderr отправляется на экран, даже если вывод stdout перенаправлен в другое место.

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

Заголовочный файл fstream определяет классы istream и ostream, позволяющие работать с файлами как с потоками.

И средства языка C, и средства языка C++ 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, самвол 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.

3.3. Работа с текстовыми файлами в стиле C++

Пример 52. Создать текстовый файл, содержащий ведомость о результатах сдачи студентами трех экзаменов. Вывести содержимое этого файла на экран.

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

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

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

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

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

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

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

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

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

Поскольку в задаче не требуется разбирать и анализировать содержимое текстового файла, самый простой способ — прочитать его посимвольно. При этом используется не операция >>, а функция посимвольного ввода `get(char&))`. Это связано с тем, что операция >> игнорирует пробельные (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 53. Написать программу, подсчитывающую общее количество символов в нескольких текстовых файлах. Список имен файлов, для которых нужно выполнить подсчет, задается в списке аргументов командной строки при запуске программы.

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

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

В программах, использующих файлы, часто применяют прием ввода имен обрабатываемых файлов через аргументы командной строки. Для рас-

сматриваемой программы, например, после ее компиляции и получения исполнимого файла count.exe на диске G в корневом каталоге, подсчитать общее количество символов в файлах text1.txt, text2.txt, info.dat можно следующим образом:

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

Чтобы обеспечить возможность работы в программе с аргументами командной строки, используется параметры в заголовке функции `main()`:

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

Параметр `argc` передает в программу количество аргументов командной строки, в число аргументов командной строки входит и само имя выполняемой программы. Параметр `argv[]` представляет собой массив указателей на символьные строки, содержащие аргументы командной строки. Обычно программы, использующие аргументы командной строки, начинаются с проверки правильности вызова и, в случае отсутствия необходимых аргументов, выдачи сообщения с подсказкой.

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

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

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

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

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

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 защищенный файл и пр., один из битов состояния потока устанавливается в 1. Нормальная работа с потоками возможна только тогда, когда все биты маски равны 0.

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

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

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

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

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

Как и в предыдущем примере, для подсчета всех символов в файлах используется посимвольный ввод методом `get()`. Этот метод возвращает

значение `true`, пока поток имеет «хорошее» состояние (все биты сброшены). При достижении конца файла меняется состояние потока, и метод возвращает значение `false`.

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 освобожены только тогда, когда завершится время жизни этого объекта. Поэтому данный прием экономит ресурсы.

☺ Всегда, когда алгоритм позволяет обрабатывать файлы последовательно, рекомендуется использовать один объект файлового типа, по очереди связывая его с файлами.

Пример 54. 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;
}
```

Приведенные в программе функции

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

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

Для выделения и печати каждого слова в текстовом файле описаны основная функция `void printWords(char* nameFile)` и две вспомогательные функции. При реализации вспомогательных функций использованы следующие предположения. Словом является любая последовательность непробельных символов. Между словами, в начале и конце строки

может быть любое количество пробелов. Слово не может быть расположено на нескольких строках.

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

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

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

Пример 55. Написать программу, позволяющую вводить строки и дописывать их в текстовый файл. Добавляемые строки переформатировать так, что бы 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 задается по умолчанию. Например, для класса `ifstream` по умолчанию используется режим, задаваемый константой `ios::in` (открыть для чтения). Для класса `ofstream` значение по умолчанию `ios::out|ios::trunc` (открыть для записи и усечь файл). Поразрядный оператор «или» (`|`) используется для объединения двух режимов.

Использованный в функции `appendTextFile()` режим `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 создается.

```
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, необходить сбросить `failbit`:

```
cin.clear();
```

Пример 56. 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()`.

3.4. Работа с бинарными файлами в стиле C++

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

Пример 57. 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 потокового объекта (или при открытии) необходимо указать режим `ios::binary`. В отличие от текстового режима, этот режим должен быть задан явно. При явном указании режима требуется определить все режимы открытия файла, соединив их поразрядной операцией | (или).

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

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

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

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

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

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

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

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

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

Для обоих методов существуют два варианта перегрузок с одним параметром и с двумя. В первой версии функции позиция задается абсолютным значением, во второй — через смещение от одной из позиций (начало, текущая, конец). В рассматриваемом примере использована первая версия:

```
fin.seekg(place);
```

Пример 58. 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)` создает бинарный файл вещественных чисел. Ввод данных с клавиатуры организован с помощью цикла `while`. 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`.

```
while (cin >> val){…}
```

Следует обратить 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 прием: 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 — ее 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();
```

3.5. Работа с текстовыми файлами в стиле языка С

Пример 59. 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 структуры необходимо подключить файл cstdio.

Параметр для установки режима открытия является строкой. Возможные режимы открытия приведены в таблице 5.

Таблица 5

Режимы открытия файлов

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

По умолчанию файл открывается в текстовом режиме. В случае открытия файла в бинарном режиме в строку определения режима необходимо добавить букву «b». При наличии любой ошибки чтения-записи `fopen` возвращает `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)` выдает 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 ее 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 60. 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 передается 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.

Некоторые часто используемые спецификации форматов приведены в таблице 6, более подробную информацию можно найти в справочниках и книгах по языку С. Например, Б. Керниган и Д. Ритчи «Язык программирования Си» [5].

Таблица 6

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 61. 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 ее в строку в стиле C, на которую указывает переменная `buf`. Чтение заканчивается, если прочитано 254 символа или введен 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 прием с удалением и переименованием файлов обычно не используется.

☺ При обработке файлов не рекомендуется смешивать использование потоковых классов и функций библиотеки cstdio.

3.6. 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 62. 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 _tmain(int argc, _TCHAR* 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 размер записи в файле. При этом указатель `beg` увеличивается, а `end` уменьшается. Выход из цикла произойдет, 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;
}
```

3.7. 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.


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 = 0;
  if (n > 0) {
    tail=head = new list;
    cout << "list item ->";
    cin >> head->inf;
    head->next = 0;
  }
  for (int i = 1; i<n; ++i) {
    p = new list;
    cout << "list item ->";
    cin >> p->inf;
    p->next = 0;
    tail->next = p;
    tail = p;
  }
  return head;
}
void print_sp(list* L) {
  list * p;
  p = L;
  while (p != NULL) {
    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 определенная 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` создает 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 передается по ссылке, так как его значение в функции меняется:

```
void erase(list*&L);
```

В функцию удаления списка передается указатель на голову списка по ссылке, чтобы обеспечить изменение фактического параметра после выполнения функции.

Спецификация формального параметра `list*&L` выглядит достаточно экзотично. Чтобы не прибегать к такой конструкции, можно описать тип указателя на список и изменить заголовки функций:

```
typedef list* sp_ptr;
```

Тогда объявления соответствующих функций будут выгдядеть так:

```
sp_ptr sp_create();
void print_sp(sp_ptr L);
void erase(sp_ptr &L);
```

Пример 64. 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=NULL;
  for (int i=0; i<n; ++i) {
    p = new list;
    cout << "list item ->";
    cin >>p->inf;
    p->next=L;
    L=p;
  }
  return L;
}
void print_sp(sp_ptr L) {
  sp_ptr p;
```

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

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

void erase(sp_ptr&L) {
  sp_ptr t;
  t=L;
  while(t!=NULL){
    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 NULL.

```
while (p!=NULL) {
q=p->next;
  while (q!= NULL && p->inf==q->inf)
  …
}
```

Пример 65. 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;
}
```

Вернемся 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;
```

3.8. Двусвязные списки

Линейные односвязные списки не поддерживают с должной эффективностью некоторые важные операции, например, переход на предыдущий элемент списка.

Самым простым решением является добавление еще 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 66. 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=0, pList L2=0);
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 = 0, L = 0;
  insert(F, L, 5);
  insert(F, L, 3, 0, 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 расположеные подряд
```

```
  // элементы одного списка, причем L1 < L2
  // или L1 должно быть NULL, если L2 – голова
  // или L2 должно быть NULL, если L1 – хвост
  assert(L1 == 0 || L2 == 0|| 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 != 0) {
    cout << p->inf << " ";
    p = p->next;
  }
  cout << "\n";
}

void reversePrint(pList L){
  pList p;
  p = L;
  while (p != 0) {
    cout << p->inf << " ";
    p = p->prev;
  }
  cout << "\n";
}

void delNode(pList &F, pList &L, pList p) {
  assert(F!=0 && p != 0);
  if (p == F) {
    F = F->next;
    if (F == 0)
      L = 0;
    else
      F->prev = 0;
```

```
  }
  else if (p == L){
    L = L->prev;
    L->next = 0;
  }
  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 != 0){
    F = t->next;
    delete t;
    t = F;
  }
  F = L = 0;
}
```

Описание узла списка теперь выглядит следующим образом:

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

При создании элемента двусвязного списка добавляется инициализация еще одной ссылки.

```
pList p = new list;
p->inf = a;
p->prev = L1;
p->next = L2;
```

Во всех функциях, где добавляются или удаляются элементы, список передается двумя указателями (на первый и последний элементы), причем оба передаются по ссылке.

```
void insert(pList &F, pList &L, int a, pList L1=0, pList L2=0);
void delNode(pList &F, pList &L, pList p);
void erase(pList &F, pList &L);
```

Особеностью реализации функции вставки является ее 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 ограничений. Для добавления в начало списка параметр L2 должен указывать на первый элемент, а L1 должно быть равен NULL . Для добавления в конец списка параметр L1 должен быть указателем на последний элемент, а L2 должен быть равен NULL. В остальных случаях параметры L1, L2 должны указывать на расположеные подряд элементы одного списка, причем L1<L2.

Ниже приведены примеры вызовов этой функции:

```
insert(F, L, 5);          //Добавление элемента в пустой список
insert(F, L, 3, 0, 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, ее 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 перадется указатель на удаляемый элемент.

3.9. Бинарные деревья

Деревья можно определить двумя способами: рекурсивным и нерекурсивным. Рекурсивное определение позволяет компактно описывать ал-

горитмы для работы с деревьями: 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 путем 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.

Пример 67. 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!=NULL) {
    printTREE(t->rt,n+1);
    for (int i=0; i<n; ++i)
      cout<<"  ";
    cout<<t->inf<<endl;
    printTREE(t->lt,n+1);
  }
}
t_ptr create(int n) {
  t_ptr p;
  int d;
  if (n>0) {
    p= new elem;
    cout<<"Another element?";
    cin>> p->inf;
    d= n/2;
    p->lt=create(d);
    p->rt=create(n-1-d);
    return p;
  }
  else
```

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

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

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 создает один узел и выполняет рекурсивные вызовы для создания левого и правого поддерева, содержащих соответственно `n/2` и `n-1-d/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 поддеревь-

ев 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 нее 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 10 приведены результаты двух вариантов вывода дерева: при инфиксном обходе слева направо и в виде дерева, повернутого на 90 градусов против часовой стрелки.

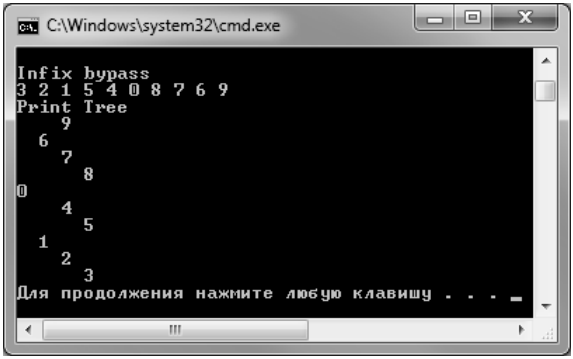


Рис. 10. Вывод сбалансированного дерева

Пример 68. Написать функции для вычисления суммы четных 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 != NULL) {
    printTREE(t->rt, n + 1);
    for (int i = 0; i<n; ++i)
      cout << "  ";
    cout << t->inf << endl;
    printTREE(t->lt, n + 1);
  }
}
t_ptr create(int n) {
  t_ptr p;
  int d;
  if (n>0) {
    p = new elem;
    cout << "Another element?";
    cin >> p->inf;
    d = n / 2;
    p->lt = create(d);
    p->rt = create(n - 1 - d);
    return p;
  }
  else
    return NULL;
}

void erase(t_ptr t) {
  if (t != NULL) {
    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 69. 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 = 0;
  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 0;
  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 != 0)
  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 == 0) && (t->rt == 0)) {// 1
    delete t;
    t = 0;
  }
  else if (t->lt == 0) {// 2
    delNode = t;
    t = t->rt;
    delete delNode;
  }
  else if (t->rt == 0) {// 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 == 0) {// 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 = 0;
    t->rt = 0;
  }
  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 0;
  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 расмотрен 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 осо-

бенностей и аналогично удалению последнего элемента односвязного списка.

```
if ((t->lt == 0) && (t->rt == 0)) { // 1
  delete t;
  t = 0;
}
```

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 == 0) {// 2
  delNode = t;
  t = t->rt;
  delete delNode;
}
else if (t->rt == 0) {// 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 == 0) // 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`.

МОДУЛЬ 4. ПОДРОБНЕЕ О ФУНКЦИЯХ

4.1. Указатели и массивы указателей на функции

Указатель на функцию может использоваться не только для передачи параметров, но и как переменная, через которую можно осуществить вызов функции.

Пример 70. Продемонстрировать использование указателя для вызова функции.

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

void func() {
  cout<<"func() called... "<<endl;
}

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

Пример 71. Дан массив целых чисел из `N` элементов. Создать функцию `Accumulate(f,s,A,n)`, применяющую функцию `f(A,n)` к массиву `A`. Функция `f(A, n)` передается в качестве параметра и возвращает целое значение. Результат применения функции `f(A,n)` к массиву `A` записать в выходной параметр `s` функции `Accumulate(f,s,A,n)`. С помощью функции `Accumulate(f,s,A,n)` найти минимальный элемент в массиве, сумму и произведение элементов массива.

Заголовочный файл funcs.h

```
#ifndef FUNCS_H
#define FUNCS_H

void input (int a[], int& n, int maxn);
int min(int a[], int n);
```

```
int sum(int a[], int n);
int mult(int a[], int n);
typedef int (*pf) (int[], int);
void Accumulate(pf f, int& s, int a [], int n);
#endif
```

В заголовочном файле используется определение типа указателя на функцию

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

Параметр такого типа передается в функцию `Accumulate`.

Описания объявленных функций — файл funcs.cpp

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

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

int min(int a[], int n){
  int min = a[0];
  for (int i = 1; i < n; ++i)
    if (a[i] < min)
      min = a[i];
  return min;
}
int sum(int a[], int n){
  int s = 0;
  for (int i = 0; i < n; ++i)
    s += a[i];
  return s;
}
int mult(int a[], int n){
  int p = 1;
  for (int i = 0; i < n; ++i)
    p *= a[i];
  return p;
}
```

```
void Accumulate(pf f, int& s, int a [], int n){
  s = f(a,n) ;
  return;
}
```

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

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

int main(){
  const int nmax = 100;
  int a[nmax], n;
  input(a, n, nmax);
  int s;
  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()`.

Пример 72. Продемонстрировать использование массива указателей на функции, реализованные в примере 71.

```
#include "funcs.h"
#include <iostream>
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};
```

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 прием особенно удобен при частом добавлении или удалении функций из таблицы, а также при динамическом создании или изменении таблицы.

4.2. Шаблоны функций

Перегруженные функции обычно используются для выполнения похожих по синитаксису, но разных по семантике операций. Если же для каждого типа данных должны выполняться идентичные операции, то более

компактным и удобным решением является использование шаблонов функций.

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

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

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

Пример 73. 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» [9].

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

Например, при вызове

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

реализация функции для типа `int` будет выглядеть следующим образом:

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

При вызове

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

реализация функции для типа `double` будет выглядеть так:

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

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

При многофайловой организации проекта может создаваться слишком много одинаковых конкретизаций шаблонов (шаблонных функций) для

одинаковых списков параметров. Для предотвращения такой ситуации можно использовать явное объявление конкретизации. Тогда шаблонная функция будет создана заранее ровно один раз для одного списка параметров, указанного в объявлении.

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

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

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

Пример 74. 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 max(const T a, const T b) {
  return a>b ? a : b;
}

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

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

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

```
  else if (a.month<b.month)
    return b;
  else if (a.day>b.day)
    return a;
  else
    return b;
}

int main() {
  cout << max(3, 2) << endl;
  cout << max(3.5, 20.4) << endl;
  cout << max("Hello", "By") << endl;
  date a = { 27, 03, 2015 }, b = { 31, 01, 2008 }, c;
  c = max(a, b);
  cout << c.day << '.' << c.month << '.' << c.year << endl;
  return 0;
}
```

Шаблон функции необходим, когда описываемый алгоритм может быть применен к данным различных типов. Поэтому шаблон функции `T max(const T a, const T b)` можно использовать для создания шаблонных функций для типов `int` и `double`.

```
cout << max(3, 2) << endl;
cout << max(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* max(const char* a, const char* b) {
  if (strcmp(a, b)>0)
    return a;
  else
    return b;
}

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

```
date max(const date &a, const date &b){//перегруженная функция
  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:

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

4.3. Приведение типов данных

Приведение типов данных в стиле языка С доступно и в языке С++ и до сих пор применяется в силу лаконичности записи. Приведение в стиле С 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 С++ описывает синтаксис операций явного приведения типов (таблица 7), способных полностью заменить операции приведения в стиле С. 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 С++, то компилятор сообщит об ошибке. Приведение в стиле С не обладает та-

кой возможностью. Если в программе встречается приведение типа, то следует восприниматеь его как предупреждение, что происходит что-то необычное. При этом операторы в стиле C++ несут больше информации.

Таблица 7

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 надежного использования значение должно быть приведено обратно к исходному типу. Тип, к которому выполняется приведение, обычно задействуется для манипуляций с битами или других неочевидных целей. Это самый опасный из всех видов приведения.
`dynamic_cast`	Понижающее приведение, безопасное по отношению к типам.

ЛИТЕРАТУРА

1. *Абрамян М. Э.* 1000 задач по программированию. Ч. I–III. — Ростов н/Д: УПЛ РГУ, 2004. — 128 с.
2. *Абрамян М. Э., Захаренко Е. О., Михалкович С. С., Столяр А. М.* Программирование и вычислительная физика. Вып. IV: Указатели, ссылки и массивы в C++. — Ростов н/Д: УПЛ РГУ, 2005. — 41 с.
3. *Дейтел Х. М., Дейтел П. Дж.* Как программировать на C++. — М.: ЗАО «Издательство БИНОМ», 2000. — 1024 с.
4. *Кениг Эндрю, Му Барбара.* Эффективное программирование на C++. Практическое программирование на примерах. — М.: Издательский дом «Вильямс», 2002. — 384 с.
5. *Керниган Б., Ритчи Д.* Язык программирования Си = The C programming language. — 2-е изд. — М.: Вильямс, 2007. — 304 с.
6. *Лишнер Рэй.* C++. Справочник. — СПб.: Питер, 2005. — 907 с.
7. *Прата С.* Язык программирования C: Лекции и упражнения = C Primer Plus. — М.: Вильямс, 2006. — 960 с.
8. *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.
9. *Седжвик Р.* Фундаментальные алгоритмы на C. Анализ/Структуры данных/Сортировка/Поиск/Алгоритмы на графах. — СПб.: ООО «ДиаСофтЮП», 2003. — 1136 с.
10. *Страуструп Бьерн.* Язык программирования C++. Специальное издание. — М.: ООО «Бином-Пресс», 2004. — 1104 с.
11. *Эккель Брюс.* Философия C++. Введение в стандартный C++. — СПб.: Питер, 2004. — 572 с.

12. *Эккель Брюс, Эллисон Чак.* Философия C++. Практическое программирование. — СПб.: Питер, 2004. — 608 с.
13. Стандарт C++11 (ISO/IEC 14882:2011) http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

Учебное издание

Демяненко Яна Михайловна
Чердынцева Марина Игорьевна

МЕТОДЫ ПРОЦЕДУРНОГО ПРОГРАММИРОВАНИЯ В C++

Учебное пособие

Компьютерная верстка *Я. М. Демяненко, М. И. Чердынцева*

Подписано в печать 12.12.2014 г. Заказ № 4044.
Тираж 100 экз. Формат 60×84 $^1/_{16}$. Печ. лист 12,09. Уч.изд.л. 6,1.

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) 247-80-51.