Д. В. Алейников, Е. П. Холодова

ЯЗЫК ПРОГРАММИРОВАНИЯ C++

Структурное программирование

Учебно-методическое пособие

Рекомендовано учебно-методическим объединением
по образованию в области управления для студентов
учреждений высшего образования специальности
1-26 03 01 Управление информационными ресурсами
в качестве учебно-методического пособия

Минск
2014

УДК 004.432 : 004.42(075)
ББК 32.973.22я7
А45

Д. В. Алейников, Е. П. Холодова

Рецензенты:
Шемаров А. И., кандидат технических наук, доцент, заведующий кафедрой управления информационными ресурсами Академии управления при Президенте Республики Беларусь;
Белодед Н. И., кандидат технических наук, доцент кафедры управления информационными ресурсами Академии управления при Президенте Республики Беларусь

Алейников, Д. В., Холодова, Е. П.

А45 Язык программирования С++ : структурное программирование : учебно-методическое пособие : рекомендовано учебно-методическим объединением по образованию в области управления для студентов учреждений высшего образования специальности 1-26 03 01 Управление информационными ресурсами в качестве учебно-методического пособия / Д. В. Алейников, Е. П. Холодова. – Минск : Национальная библиотека Беларуси, 2014. – 114 с.

ISBN 978-985-7039-50-0

В пособии излагаются основы языка программирования С++: типы данных, переменные, базовые конструкции структурного программирования, массивы, строки, указатели и функции. Теоретические сведения излагаются в доступной форме и дополняются простыми примерами.

Учебно-методическое пособие разработано для студентов экономических специальностей, но будет полезно всем желающим получить базовые навыки в написании простейших программ на языке С++.

УДК 004.432 : 004.42(075)
ББК 32.973.22я7

ISBN 978-985-7039-50-0

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Среди множества языков программирования C++ занимает особое место. Предлагаемое учебно-методическое пособие предназначено для студентов экономических специальностей, изучающих язык C++ «с нуля». Его задача – дать краткое и четкое изложение основ языка C++ с целью приобретения студентами навыков программирования. Пособие не претендует на полноту изложения материала, для этого следует использовать справочники и дополнительную литературу. В нем даются краткие теоретические сведения, необходимые для освоения рассматриваемых тем, а для лучшего их усвоения в пособии приведено большое количество наглядных примеров программ с подробными комментариями.

Пособие составлено с учетом опыта преподавания курса «Алгоритмизация и программирование», читаемого в Институте бизнеса и менеджмента технологий БГУ на факультете бизнеса для студентов специальности «Управление информационными ресурсами».

Авторы выражают благодарность рецензентам: заведующему кафедрой управления информационными ресурсами Академии управления при Президенте Республики Беларусь, кандидату технических наук, доценту Шемарову Александру Ивановичу; кандидату технических наук, доценту кафедры управления информационными ресурсами Академии управления при Президенте Республики Беларусь Белодеду Николаю Ивановичу – за ценные замечания и предложения, способствовавшие улучшению пособия, а также заведующему кафедрой менеджмента технологий ИБМТ БГУ, доценту, кандидату технических наук Силковичу Юрию Николаевичу за предоставленную возможность чтения дисциплины «Алгоритмизация и программирование».

ВВЕДЕНИЕ

Если и существует один компьютерный язык, который определяет суть современного программирования, то, безусловно, это C++. Язык C++ предназначен для разработки высокопроизводительного программного обеспечения. C++ послужил фундаментом для разработки языков будущего. Например, как Java, так и C# – прямые потомки языка C++.

Самым трудным в изучении языка программирования, безусловно, является то, что ни один его элемент не существует изолированно от других. Компоненты языка работают вместе, можно сказать, в дружном «коллективе». Такая тесная взаимосвязь усложняет рассмотрение одного аспекта C++ без изучения других. Зачастую обсуждение одного средства предусматривает предварительное знакомство с другим.

Язык C++ был создан Бьерном Страуструпом (Bjarne Stroustrup) в 1979 году в компании Bell Laboratories (г. Муррей-Хилл, шт. Нью-Джерси). Сначала новый язык получил имя "C с классами" (C with Classes), но в 1983 году он стал называться C++.

Страуструп построил C++ на фундаменте языка C, включающем все его средства, атрибуты и основные достоинства. Большинство новшеств, которыми Страуструп обогатил язык C, было предназначено для поддержки объектно-ориентированного программирования (ООП). По сути, C++ стал объектно-ориентированной версией языка C. Взяв язык C за основу, Страуструп подготовил плавный переход к ООП.

Несмотря на то, что C++ изначально был нацелен на поддержку очень больших программ, этим, конечно же, его использование не ограничивалось. И в самом деле, объектно-ориентированные средства C++ можно эффективно применять практически к любой задаче программирования. Неудивительно, что C++ используется для создания компиляторов, редакторов, компьютерных игр и программ сетевого обслуживания. Программное обеспечение многих высокоэффективных систем построено с использованием C++. Кроме того, C++ –это язык, который чаще всего выбирается для Windows-программирования.

В языке C++ воплощены лучшие идеи структурного программирования, которое зародилось в середине 1960-х гг. Структурная методология разработки программного обеспечения была изобретена голландским ученым Эдсгером Дейкстрой (Edsger Dijkstra). Он впервые сосредоточил внимание на эффективных способах внутренней организации программы. В 1969 г. в Риме прошла научная конференция НАТО по программированию. Дейкстра выступил на ней со знаменитым докладом «Структурное программирование». Он показал, что при программной реализации любого алгоритма можно обойтись без оператора перехода. Достаточно придерживаться последовательного выполнения частей программы. Сама программа создается методом «сверху-вниз». Сначала в реализуемом алгоритме выделяются крупные модули. Их функционирование детализируется до более и более мелких элементов, но все они выполняются последовательно.

Структурное программирование – технология создания программ в виде иерархически связанных модулей, представляющих собой линейную последовательность операторов цикла, присваивания и условных операторов, а также вызовов вложенных модулей. Концепция структурного программирования обусловила возможность автономного написания и отладки крупных модулей и стала необходимой предпосылкой для перехода к концепции объектно-ориентированного программирования.

1. БАЗОВЫЕ СВЕДЕНИЯ О ЯЗЫКЕ C++

Как и все языки программирования, язык C++ является способом передачи компьютеру списка логически точных инструкций, которые предписывают компьютеру, что нужно делать. Этот список инструкций и является текстом программы, называемый *исходным кодом (source code)*. Также компьютеру требуется информация – это *данные (data)* для программы. Преимущество любого языка программирования заключается в том, что он следует правилам, которые не допускают неопределенности. Инструкции языка программирования имеют строгие синтаксические правила.

Так как компьютер понимает только свой родной язык – *машинный код (machine code)*, то для работы программы требуется ее преобразование в машинный код. Приложение, которое преобразует исходный код программы (то есть инструкции языка C++) в машинный код, называется *компилятором (compiler)*. *Объектный модуль* (*object code*) – результат обработки компилятором исходного модуля. Объектный модуль не может быть выполнен. Это незавершенный вариант машинной программы. К объектному модулю в общем случае должны быть подсоединены модули стандартной библиотеки, и он должен быть настроен по месту выполнения. Исполняемый (абсолютный) модуль создает *компоновщик* (*linker*), объединяя в один общий модуль объектные модули, реализующие отдельные части алгоритма. На этом этапе к машинной программе подсоединяются необходимые функции стандартной библиотеки.

Стандартная библиотека (*library*) – набор программных модулей, выполняющих наиболее часто встречающиеся в программировании задачи: ввод, вывод данных, вычисление математических функций, сортировки, работа с памятью и т. д.

До компиляции над программой обычно выполняются некоторые предварительные действия, например подключение к компилируемой программе текстов других исходных модулей и соответствующих разделов системной библиотеки. Эта работа выполняется так называемым *препроцессором*, обычно являющимся составной частью компилятора.

Чтобы написать программу на языке С++, необходимо каким-то образом ввести инструкции программы. Для этого существует несколько способов. Но чаще всего для создания программ используют специальную интегрированную среду разработки (*IDE – Integrated Development Environment*). Среда разработки содержит встроенный текстовый редактор, графический интерфейс и другие инструменты, которые позволяют редактировать, компилировать и отлаживать код С++. Наиболее известными являются: Microsoft Visual Studio и С++ Builder. На рис. 1.1 представлен вид окна Microsoft Visual Studio с редактируемым кодом программы.

Все программы данного пособия были разработаны и выполнены в среде разработки *Code::Blocks* с использованием компилятора *GCC*.

В языке программирования существует своя терминология. В данном разделе приводятся основные понятия, которыми оперирует язык С++.

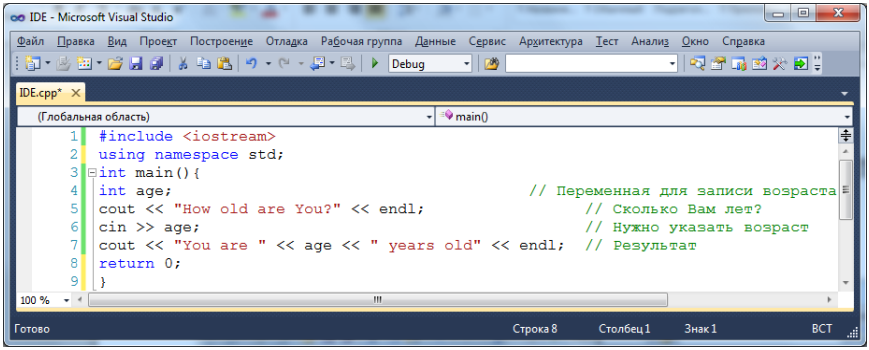


Рис. 1.1. Вид окна Microsoft Visual Studio с кодом программы

1.1. Алфавит языка

Алфавит языка программирования, или его символы – это основные неделимые знаки, с помощью которых пишутся все тексты программ. Алфавит С++ включает:

- прописные и строчные латинские буквы и знак подчеркивания;

- арабские цифры от 0 до 9;
- специальные знаки[1]:

" { } , | [ ] ( ) + - / % * . \
' : ? < = > ! & # ~ ; ^

- пробельные символы: пробел, символы табуляции, символы перехода на новую строку.

Из символов алфавита формируются лексемы языка. ***Лексема,*** или элементарная конструкция, – минимальная единица языка, имеющая самостоятельный смысл. Лексемы C++ – это:

- идентификаторы;
- ключевые (зарезервированные) слова;
- знаки операций;
- константы;
- разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами, такими, как разделители или знаки операций.

1.2. Идентификаторы

Идентификатор в C++ представляет имя программного объекта. Идентификаторы могут состоять из одного или нескольких символов, цифр и символов подчеркивания. Первым символом идентификатора должна быть буква или символ подчеркивания, но не цифра. Символ подчеркивания используется для улучшения читабельности идентификатора, состоящего из нескольких слов, например *line_count*.

🕭 ***Строчное и прописное написание букв в C++ различается.***

Например, *sbmt, Sbmt, sBMT, sbMT* – это разные идентификаторы. Определяя идентификаторы, нельзя допускать совпадений с ключевыми словами языка и именами стандартных функций.

1.3. Ключевые слова

Ключевые слова – это идентификаторы, зарезервированные стандартом C++, которые имеют специальное значение. *Все*

[1] Назначение этих символов будет рассмотрено далее в других разделах пособия.

ключевые слова записываются строчными буквами и должны быть использованы в коде программы именно в соответствии со своим назначением. В большинстве книг по C++ приводится полный перечень ключевых слов.

✍ В данном пособии все ключевые слова будут выделены **полужирным шрифтом**.

1.4. Комментарии

Подобно большинству других языков программирования, C++ позволяет вводить в исходный код программы комментарии.

Комментарии представляют собой специально обозначаемые фрагменты программы (любая совокупность знаков), содержание которых компилятор игнорирует.

С помощью комментариев описываются или разъясняются действия, выполняемые в программе, и эти разъяснения предназначаются для тех, кто будет читать исходный код. В реальных приложениях комментарии используются для разъяснения особенностей работы отдельных частей программы, конкретных действий программных средств, детального описания всех (или некоторых) ее строк.

В C++ поддерживается два типа комментариев: однострочные и многострочные. *Однострочный комментарий* начинается с пары символов // и продолжается только до конца строки. *Многострочный комментарий* должен начинаться символами /* и заканчиваться */, может занимать несколько строк. Например,

```
// Это однострочный комментарий
/* Этот
          комментарий уже
                              многострочный  */
```

1.5. Переменные

Основная цель любой программы состоит в обработке данных. ***Переменная*** – это именованная область памяти, используемая для хранения информации. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение. Память компьютера можно рассматривать как ряд последовательно пронумерованных ячеек, каждая из которых занимает

1 байт. Номера ячеек называют адресами памяти. Имя переменной (например, *Line_Count*) можно представить себе как надпись на ячейке памяти, по которой, не зная настоящего адреса памяти, можно ее найти. В зависимости от своего размера, переменная *Line_Count* может занимать одну или несколько ячеек памяти[2].

Данные различного типа хранятся и обрабатываются по-разному. В любом языке каждая переменная должна иметь определенный тип. Язык C++ строг в отношении переменных – их тип надо указать до использования переменной. Эта операция называется ***объявлением переменной*** и имеет синтаксис[3]:

тип_данных *имя_переменной*

Тип данных определяет:

- *внутреннее представление* данных в памяти компьютера;
- *множество допустимых значений*, которое может принимать переменная этого типа;
- *операции и функции*, которые можно применять к величинам этого типа. В табл. 1 представлен список базовых, или основных, типов данных C++.

Таблица 1. Основные типы данных

Тип данных	Размер в байтах	Название	Диапазон
bool	1	Логический тип	Значения *true* или *false*
char	1	Символьный тип	От 0 до 255
int	4	Целые числа	От -2147483648 до 2147483647
float	4	Действительные числа	От 1.8E-38 до 1.8E+38
double	8	Действительные числа двойной точности	От 2.2E-308 до 1.8E+308
wchar_t	2	Символьный двухбайтовый тип	От 0 до 65535
void	0	Отсутствие значения	

[2] Более подробно размещение переменных в памяти компьютера будет рассмотрено в разделе «Указатели».

[3] Слова, выделенные в описании синтаксиса *курсивом*, представляют инструкции, которые программист вставляет на их место.

Компьютер соотносит с именами переменных определенные адреса в памяти и, начиная с этих адресов, выделит участки памяти (в байтах) в соответствии с тем, какого типа объявлены переменные.

При определении переменной в языке C++ необходимо предоставить компилятору информацию о ее типе, например **int**, **char** или другого типа. Благодаря этой информации компилятору будет известно, сколько места нужно зарезервировать для нее и какого рода значение будут хранить в этой переменной. Если для переменной указанного типа требуется четыре байта, то для нее будет выделено четыре ячейки, то есть именно по типу переменной (например, **int**) компилятор судит о том, какой объем памяти (сколько ячеек) нужно зарезервировать для этой переменной. Например,

```
int example;
```

В примере тип переменной *example*, заданный при объявлении, – **int** (от англ. *integer* – целое число). Это означает, что переменная имеет вид *«целое число со знаком»* и что под каждое значение числа, которое будет записано на участке *example*, отведено 32 бита (4 байта).

Максимальным значением обычной целочисленной переменной является число 2 147 483 647, минимальным – -2 147 483 648, то есть общий диапазон – около 4 млрд чисел. Таким образом, имена переменных – это названия тех участков в памяти компьютера (каждый участок имеет свой адрес), где будут находиться данные (это могут быть не только числа), с которыми программа будет работать при реализации алгоритма. Имена переменным надо давать осмысленно – так, чтобы они отражали характер содержания переменной.

С помощью одной инструкции можно объявить сразу несколько переменных, например:

```
int d1, d3, d3;
```

В языке C++ выражение, после которого стоит точка с запятой, считается ***оператором***, то есть законченным действием. Перечень описываемых переменных одного типа (тип указывается в начале перечня) обязательно должен заканчиваться точкой с

запятой – сигналом для компилятора, что описание переменных данного типа завершено. В противном случае компилятор станет при компиляции искать ближайшую точку с запятой и объединять все, что находится до нее, в один оператор (пытаться объединить разнородные данные) и в конце концов выдаст ошибку компиляции.

Следующая строка:

```
float fahr, cels;
```

означает описание переменных с именами *fahr* и *cels*, но тип этих переменных уже другой. Эти переменные – не целые числа, а так называемые *«числа с плавающей точкой».* Участки памяти, обозначаемые этими переменными, могут хранить любые вещественные числа, а не только целые. Под этот тип данных компилятор отводит 4 байта (32 бита).

Таким образом, перед составлением программы, которая будет оперировать данными (числовыми и нечисловыми), *эти данные следует описать,* им должны быть присвоены типы и имена. Присвоение переменным типов и имен фактически означает, что компилятор определит им место в памяти, куда данные будут помещаться и откуда будут извлекаться при выполнении операций над ними. Следовательно, когда мы пишем

```
c = a + b;
```

это означает, что одна часть данных будет извлечена с участка памяти с именем *a*, а другая часть данных – с участка памяти с именем *b*, произойдет их суммирование, и результат будет «положен» (записан) в участок памяти с именем *c*. Знак (=) означает «присвоить». Присваивать некоторой переменной можно не только значение с какого-либо участка памяти, то есть значение другой переменной, но и простые числа, например, `a=10`. В этом случае, компилятор просто «положит» в переменную *a* число 10.

Вместе с идентификаторами типов могут использоваться так называемые модификаторы типа. *Модификаторы типа* – это специальные ключевые слова, которые указываются перед идентификатором типа и позволяют изменять базовый тип. В C++ используются модификаторы: **signed** (значения со знаком), **unsigned** (значения без знака), **short** (укороченный тип), **long** (расширенный тип).

Все четыре модификатора могут использоваться для типа **int.** Модификаторы **signed** и **unsigned**, кроме этого, используются с типом **char.** Модификатор **long** используют с типом **double**. Что касается диапазона значений для данных разных типов, то в разных компиляторах диапазоны значений данных различны. В языке C++ вводятся стандарты только для минимально необходимого диапазона, который должен поддерживать компилятор. Данные, имя типа которых включает модификатор **signed**, являются данными со знаком, а данные, имя типа которых содержит модификатор **unsigned** – данными без знака. При отсутствии модификатора по умолчанию принимается модификатор **signed.**

В C++ предусмотрен сокращенный способ объявления **unsigned**-, **short**- и **long**-значений целочисленного типа. Это значит, что при объявлении **int**-значений достаточно использовать слова **unsigned**, **short** и **long**, не указывая тип **int**, то есть тип **int** подразумевается. Например, следующие две инструкции объявляют целочисленные переменные без знака:

```
unsigned x; unsigned int y;
```

Тип **char** предназначен для хранения символов. Чтобы задать символ, необходимо заключить его в одинарные кавычки. Например,

```
char ch;
ch='x';
```

Тип **char** может быть модифицирован с помощью модификаторов **signed** и **unsigned**. Переменные типа **char** можно использовать не только для хранения символов, но и для хранения числовых значений. Переменные типа **char** могут содержать «небольшие» целые числа в диапазоне от -128 до 127. Объявление типа **char** подразумевает значение со знаком. Следовательно, использование модификатора **signed** для **char**-объявления также избыточно.

Тип **wchar_t** предназначен для хранения символов, входящих в состав больших символьных наборов. Например, в китайском языке определено очень большое количество символов, для которых представление, обеспечиваемое типом **char** недостаточно.

Чаще всего в профессиональных программах используется тип **double**, так как большинство математических функций используют **double**-значения.

Тип **bool** предназначен для хранения булевых (то есть ИСТИНА/ЛОЖЬ) значений. В C++ определены две булевы константы: **true** и **false**, являющиеся единственными значениями, которые могут иметь переменные типа **bool**.

Важно понимать, как значения ИСТИНА/ЛОЖЬ определяются в C++. Один из фундаментальных принципов C++ состоит в том, что любое ненулевое значение интерпретируется как ИСТИНА, а нуль – как ЛОЖЬ. Этот принцип полностью согласуется с типом данных **bool**, поскольку любое ненулевое значение, используемое в булевом выражении, автоматически преобразуется в значение **true**, а нуль – в значение **false**. Обратное утверждение также справедливо: при использовании в небулевом выражении значение **true** преобразуется в число 1, а значение **false** – в число 0. Конвертируемость нулевых и ненулевых значений в их булевы эквиваленты особенно важна при использовании инструкций управления.

Тип **void** используется для задания выражений, которые не возвращают значений. Тип **void** не ассоциируется с каким-либо набором данных. Его задача заключается в обеспечении синтаксических конструкций в специфических случаях, например при описании функций, не имеющих параметров или не возвращающих значений.

1.6. Константы и литералы

В C++ под константами и литералами, как правило, подразумевают одно и то же: это такие значения, предназначенные для восприятия пользователем, которые не могут быть изменены в ходе выполнения программы. Приведенное определение относится большей частью к ***литералу***.

Константы – именованные ячейки памяти, значения которых фиксируются на начальном этапе выполнения программы и не могут быть изменены программой. В этом смысле константы – это те же переменные, но только с фиксированным, определен-

ным единожды, значением. Константы могут иметь любой базовый тип данных.

Объявляются константы при помощи спецификатора **const**:

const тип_константы *имя_константы=значение*;

Например, так можно объявить константу с именем *max*:

```
const int max=9;
```

1.7. Преобразование типов

Если переменные одного типа смешаны с переменными другого типа, происходит *преобразование типов*. При выполнении инструкции присваивания действует простое правило преобразования типов: значение, расположенное с правой стороны от оператора присваивания, преобразуется в значение типа переменной, расположенной слева от оператора присваивания.

Также в C++ предусмотрена возможность установить заданный тип для выражения. Для этого используется *операция приведения типов* (cast), общий формат которой имеет вид:

static_cast<тип>(*выражение*)

Здесь элемент *тип* означает тип данных, к которому необходимо привести выражение. Например,

```
int i;     // Объявляется переменная i типа int
i=1;       // Переменной i присваивается число 1
float f;   // Объявляется переменная f типа float
f=static_cast<float>(i+i); /* Выражение i+i приводится
к типу float и присваивается переменной f  */
```

2. СТРУКТУРА ПРОГРАММЫ НА С++

В общем случае программа на С++ имеет следующую структуру:

1. Блок заголовков программы.
2. Блок с прототипами (объявлениями функций)[4].
3. Главная функция программы (main).
4. Блок с описаниями функций (прототип которых указан во 2-м блоке).

Обязательными для любой программы являются первый и третий блоки. Рассмотрим пример простой консольной программы, которую на первом этапе можно использовать как шаблон структуры простейших программ.

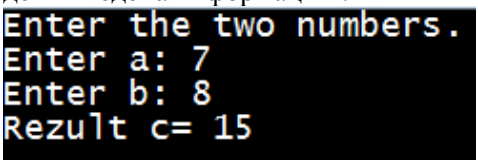
Пример. Программа[5], которая запрашивает у пользователя два числа и выдает на экран сумму этих чисел.

```
1  /* Это
2        простая
3              С++-программа */
4  #include <iostream>
5  // С++-программа  начинается  с  функции main()
6  int main()
7  {std::cout<<"Enter the two numbers."<<std::endl;
8  int  a;
9  int  b;
10 std::cout << "Enter a: ";
11 std::cin >> a;
12 std::cout << "Enter b: ";
13 std::cin >> b;
14 int c;
15 c=a+b;
16 std::cout << "Rezult c= " << c << std::endl;
17 return   0;
18 }
```

[4] О функциях подробно будет изложено в разделе «Функции».

[5] Для удобства анализа программы слева указан номер строки программного кода.

В результате выполнения программы в консольном окне будет выведена информация[6]:



Приведенный выше пример программы содержит ключевые средства, характерные для всех C++-программ. Рассмотрим подробно каждую ее строку. Строки 1–3 представляют *многострочный комментарий*.

Рассмотрим следующую строку: `#include <iostream>`. Знаком # (*sharp*) в C++ начинаются *директивы препроцессора*. Важнейшей директивой является `#include`, которая используется для добавления содержимого одного из библиотечных файлов языка C++ в исходный код программы. Как уже упоминалось ранее, библиотечные файлы подключаются для того, чтобы в программе можно было использовать их функции. В файле `<iostream>` определены функции, осуществляющие ввод/вывод данных.

Очередная 5-я строка в программе – это *однострочный комментарий*.

Одним из ключевых элементов C++ является блок кода. *Блок кода* – это логически связанная группа программных инструкций (то есть одна или несколько инструкций), размещенных между фигурными (открывающей и закрывающей) скобками ({}), которые обрабатываются как единое целое.

Основной код программы начинается в строке 6 с вызова функции **main**, содержимое которой ограничено с двух сторон фигурными скобками. Любая программа на C++ обязательно содержит функцию **main**, и может включать одну или несколько других функций. *Функция* – это блок кода программы, который выполняет одно или несколько действий. Обычно функцию вызывает другая функция или оператор, но функция **main** – особая: она вызывается автоматически при запуске программы.

[6] Числа 7 и 8 для переменных a и b введены пользователем.

Функция **main**, подобно всем остальным функциям, должна быть объявлена с указанием типа возвращаемого значения. Ключевое слово **int** (сокращение от слова *integer*), стоящее перед именем **main**, означает, что функция возвращает целое число[7].

В строке 7 находится инструкция вывода данных на консоль. При ее выполнении на экране компьютера отобразится сообщение "`Enter the two numbers`". В этой инструкции используется символ **<<**, который является оператором перенаправления потока. Он обеспечивает вывод выражения, стоящего с правой стороны, на устройство, указанное с левой. Команда **`std::cout`** состоит из двух частей: **`std`** и **`cout`**, разделенных знаком **`::`**. Часть **`cout`** (идентификатор составлен из частей слов *console output*) обеспечивает вывод информации на консоль, который в большинстве случаев означает экран компьютера. Идентификатор **`std`** указывает на то, что идентификатор **cout** принадлежит *пространству имен* **std**. Пространство имен создает декларативный район, в который помещаются различные элементы программы. Элементы, объявленные в одном пространстве имен, отделены от элементов, объявленных в другом пространстве. В пространстве **std** объявлена вся библиотека стандартного C++.

Итак, рассматриваемая инструкция обеспечивает вывод заданного сообщения на экран. Конструкция **`std::endl`** обеспечивает перевод текстового курсора в новую строку экрана. Обратите внимание на то, что эта инструкция *завершается точкой с запятой*. Все выполняемые C++-инструкции завершаются точкой с запятой. Отдельные элементы любой инструкции можно располагать на отдельных строках. Сообщение "`Enter the two numbers.`" представляет собой текстовую строку. В C++ под строкой понимается последовательность символов, заключенная в двойные кавычки. Строка в C++ – это один из часто используемых элементов языка.

[7] Подробно о функциях и возвращаемых ими значениях будет описано в разделе «Функции».

Строки 8–9 – это инструкции объявления переменных. Сначала указывается тип, а затем имя одной или нескольких переменных этого типа.

Инструкции в строках 11 и 13 осуществляют организацию ввода значений с клавиатуры. Каждая инструкция состоит из идентификатора устройства ввода **std::cin** (**cin** – *console input*), оператора ввода >> и имени переменной, которой в качестве значения присваивается число, которое вводит пользователь с клавиатуры.

В строке 14 программы объявляется новая переменная `c`. Далее в строке 15 следует оператор присваивания. Данная инструкция присваивает переменной `c` сумму переменных `a` и `b`.

Далее в строке 16 следует инструкция вывода значения переменной `c` на экран. Она содержит несколько операторов вывода <<, после которых последовательно указываются данные, выводимые на экран. Выводимая строка формируется двумя фрагментами: текстовой строкой и значением переменной.

Из примера видно, что эти операции могут образовывать цепочки, что приводит к последовательному выполнению всех операций ввода-вывода слева направо. Операндами при выводе могут служить *текстовые строки* (последовательности символов в кавычках).

Строкой 17 завершается функция **main**. При ее выполнении функция **main** возвращает вызывающему процессу (в роли которого обычно выступает операционная система) значение 0. Для большинства операционных систем нулевое значение, которое возвращает эта функция, свидетельствует о нормальном завершении программы. Слово **return** относится к числу ключевых слов C++ и используется для возврата значения из функции.

Можно включить в программу инструкцию `using namespace std;`, которая является командой компилятору использовать пространство имен **std**. Тогда **cin** и **cout** в коде программы можно использовать без идентификатора **std.**

3. ВЫЧИСЛЕНИЯ В С++

3.1. Понятие операции и выражения в С++

Выражения используются для вычисления значений (определенного типа) и состоят из *операндов, операций* и *скобок*. Каждый операнд может быть, в свою очередь, выражением или одним из его частных случаев – константой или переменной. Операнды задают данные для вычислений. *Выражение* – это последовательность операндов, разделителей (круглых скобок) и знаков операций, задающая вычисление. *Знак операции* – это один или более символов, определяющих действие над операндами, то есть операции задают действия, которые необходимо выполнить. Внутри знака операции пробелы не допускаются. Операции делятся на унарные, бинарные и тернарные – по количеству участвующих в них операндов, и выполняются в соответствии с приоритетами. Большинство операций выполняются слева направо. Исключение составляют унарные операции, операции присваивания и условная операция (?:), которые выполняются справа налево. Если аргумент касается двух операций одинакового приоритета, то очередность выполнения операций определяется по связыванию. Для изменения порядка выполнения операций используют круглые скобки. Таким образом, выражение x=y=z означает x=(y=z), а x+y+z означает (x+y)+z, согласно соответствующим правилам связывания.

3.2. Арифметические операции

Арифметические операции служат для описания арифметических действий:

- `-` замена знака
- `++` инкремент, добавление к аргументу единицы
- `--` декремент, вычитание из аргумента единицы
- `+` сложение
- `-` вычитание
- `*` умножение
- `/` деление
- `%` вычисление остатка от деления первого аргумента на второй

Операторы замены знака, инкремента и декремента – одноаргументные (*унарные*), остальные двухаргументные (*бинарные*).

Операторы инкремента и декремента могут стоять как перед операндом (*префиксная форма*), так и после него (*постфиксная форма*). В случае преинкремента `++arg` результатом операции является `arg+1`, в случае постинкремента `arg++` результатом операции является `arg`. Аналогичные правила относятся к декременту (предекремент `--arg` и постдекремент `arg--`).

Деление целых аргументов приводит к отбрасыванию дробной части результата. Например, результат 10/3 равен 3. В C++ оператор «%» можно применять к операндам целочисленного типа. Например, результат 10%3 равен 1. В случае вычисления остатка от деления знак результата совпадает со знаком делимого.

Пример. Демонстрация использования арифметических операций.

```
// Демонстрация использования арифметических операций
#include<iostream>
using namespace std;
int main()
{  int number; /* Объявление переменной number типа int
*/
int variable;/* Объявление переменной variable типа int
*/
cout<<"Enter number ";//Вывод сообщения
cin>>number; /* Ввод числа с клавиатуры в переменную
number */
cout<<"number "<<" variable" << endl; /* Вывод строки */
// Замена знака
variable=-number; //Присваивание переменной variable
//отрицательного значения number
/*Вывод переменных number и variable, разделенных
пробелами */
cout<< number <<"      "<< variable << endl;
//Инкремент
variable = ++number; /* После выполнения number=11,
variable=11 */
cout<< number <<"      "<< variable << endl;
variable = number++; /*После выполнения number=12,
variable=11 */
cout<< number <<"      "<< variable << endl;
```

```
//Декремент
variable = --number; / *После выполнения number=11,
variable=11 */
cout<< number << "       " << variable << endl;
variable = number--; /* После выполнения number=10,
variable=11 */
cout<< number << "       " << variable << endl;
// Сложение
variable=number+7; /* После выполнения number=10,
variable=17 */
cout<< number << "       " << variable << endl;
// Вычитание
variable=number-5;/* После выполнения number=10,
variable=5 */
cout<< number << "       " << variable << endl;
// Умножение
variable=number*4; / *После выполнения number=10,
variable=40 */
cout<< number << "       " << variable << endl;
// Деление
variable=number/3; /* После выполнения number=10,
variable=3 */
cout<<number<< "       " <<variable << endl;
/* Вычисление остатка от деления первого аргумента на
второй */
variable=number%3; /* После выполнения number=10,
variable=1 */
cout<< number << "       " << variable << endl;
return 0;
}
```

Результат выполнения программы:

```
Enter number 10
number  variable
10      -10
11      11
12      11
11      11
10      11
10      17
10      5
10      40
10      3
10      1
```

3.3. Операция присваивания

Операция присваивания может быть простой и составной. Простая – двухаргументная операция вида:

переменная = выражение

В этой операции тип выражения всегда преобразуется в тип переменной. Выполнение операции присваивания приводит к присваиванию переменной значения выражения, стоящего с правой стороны оператора присваивания. Результатом операции является значение правой стороны оператора, что позволяет составлять цепочки присваиваний, например `x=y=z=100;` (результатом выполнения "=" является значение выражение, стоящего с правой стороны, то есть все переменные получат значение, равное 100).

Операторы присваивания, в которых переменная из левой части повторяется в правой записывается в более компактной форме. Например, выражение `i=i+2` можно записать `i+=2`. Знак += обозначает *операцию с присваиванием*, а выражение `i+=2` называется *составным оператором присваивания*.

Составное присваивание имеет вид:

переменная оп выражение

где `оп` – одна из следующих операций:

+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=

3.4. Операции сравнения

Операции сравнения служат для описания следующих действий:

>	больше
<	меньше
>=	больше или равно
<=	меньше или равно
==	проверка на равенство
!=	неравенство

Результат сравнения имеет тип **int** и принимает значение 0 при невыполнении условия сравнения (ложно) и 1 при выполнении условия (истинно). Если какой-либо аргумент сравнения имеет тип **char**, то он подвергается преобразованию в тип **int**.

3.5. Логические операции

Логические операции служат для описания следующих действий:

!	инверсия
&&	конъюнкция
||	дизъюнкция

Первый из перечисленных операторов является одноаргументным, а два других – двухаргументными. Во всех случаях результат операции будет иметь значение 0 (условие ложно) или 1 (условие истинно).

Результатом инверсии будет **true**, если аргумент имел значение **false** и наоборот. Результатом конъюнкции будет **true**, если аргументы имеют значения, отличные от нуля, или **false** – в противном случае. Результатом дизъюнкции является **true**, если по крайней мере один аргумент отличен от нуля, или **false** – в противном случае.

3.6. Операция размера

Иногда полезно знать размер (в байтах) одного из типов данных. Поскольку размеры встроенных C++-типов данных в различных вычислительных средах могут быть различными, знать заранее размер переменной во всех ситуациях не представляется возможным. Для решения этой задачи в C++ включен оператор **sizeof**.

В общем случае одноаргументная операция размера имеет формат:

sizeof *имя_переменной* или **sizeof**(**имя_типа**)

Результатом операции является значение, которое определяет размер (в байтах) аргумента.

Пример. Определение размера переменной.

```
int ar;
var =  sizeof ar;   // Определение  размера переменной  ar
```

Если переменная типа **int** представлена в 4-х байтах, то переменной *var* будет присвоено значение 4.

Пример. Определение размера разных типов.

```
#include <iostream>
using namespace std;
int main()
{ cout << "char = "                << sizeof(char) << endl;
  cout << "wchar_t = "             << sizeof(wchar_t) << endl;
  cout << "unsigned char = "       << sizeof(unsigned char) << endl;
  cout << "signed char = "         << sizeof(signed char) << endl;
  cout << "int = "                 << sizeof(int) << endl;
  cout << "unsigned int = "        << sizeof(unsigned int) << endl;
  cout << "signed int = "          << sizeof(signed int) << endl;
  cout << "unsigned short int = " << sizeof(unsigned short int) << endl;
  cout << "signed short int = "    << sizeof(signed short int) << endl;
  cout << "long int = "            << sizeof(long int) << endl;
  cout << "signed long int = "     << sizeof(signed long int) << endl;
  cout << "unsigned long int = "   << sizeof(unsigned long int) << endl;
  cout << "float = "               << sizeof(float) << endl;
  cout << "double = "              << sizeof(double) << endl;
  cout << "logn double = "         << sizeof(long double) << endl;
  return 0;}
```

Результат выполнения программы:

```
unsigned int = 4
signed int = 4
unsigned short int = 2
signed short int = 2
long int = 4
signed long int = 4
unsigned long int = 4
float = 4
double = 8
logn double = 12
```

4. УПРАВЛЯЮЩИЕ ИНСТРУКЦИИ

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур, называемых *следованием, ветвлением* и *циклом*. Этот результат установлен Боймом и Якопини еще в 1966 г. путем доказательства того, что любую программу можно преобразовать в эквивалентную, состоящую только из этих структур и их комбинаций.

Следование, ветвление и цикл называют *базовыми конструкциями структурного* программирования (рис. 4.1). *Следованием* называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или составных). *Ветвление* задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия. *Цикл* задает многократное выполнение операторов.

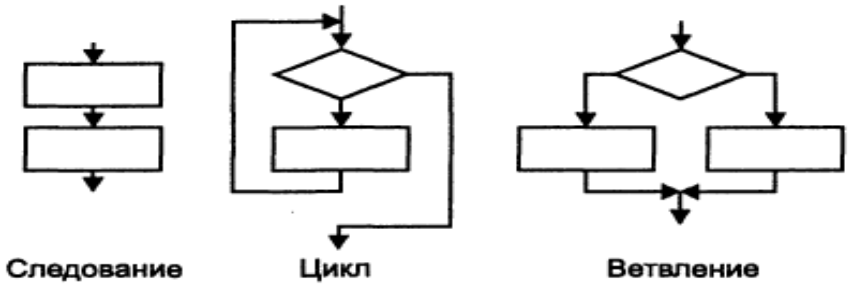


Рис. 4.1. Базовые конструкции структурного программирования

Особенностью базовых конструкций является то, что любая из них имеет только один вход и один выход, поэтому конструкции могут вкладываться друг в друга произвольным образом, например, цикл может содержать следование из двух ветвлений, каждое из которых включает вложенные циклы. Целью использования базовых конструкций является получение программы простой структуры. В большинстве языков высокого уровня существует несколько реализаций базовых конструкций.

В следующих разделах будут рассмотрены операторы языка, реализующие базовые конструкции структурного программирования.

4.1. Условные операторы

Под *условными* обычно подразумевают операторные конструкции, с помощью которых в программе реализуются точки ветвления. В C++ два условных оператора: **if** и **switch**.

4.1.1. Оператор if

Оператор **if** используется для разветвления процесса вычислений на несколько направлений, позволяет выполнять разные блоки операторов в зависимости от того, выполняется ли некое условие.

Первая форма записи оператора if

Вариант а

if (*выражение***)** *инструкция1***;**
else *инструкция2***;**

Вариант б

if (*выражение***)**
{*последовательность инструкций 1***}**
else
{*последовательность инструкций 2***}**

Условие указывается в круглых скобках после ключевого слова **if**. Если его значение истинно, то выполняется *инструкция1* (для варианта а) или *последовательность инструкций 1* (для варианта б), в противном случае выполняется *инструкция 2* или *последовательность инструкций 2* (для варианта б).

После выполнения условного оператора управление передается оператору, следующему после него. На рис. 4.2 показана структурная схема, иллюстрирующая работу условного оператора.

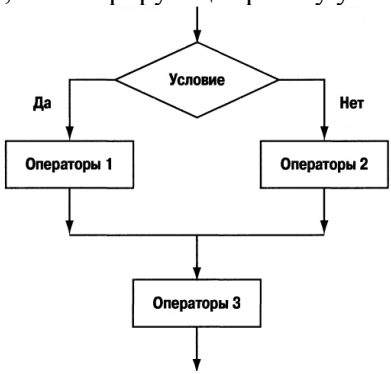
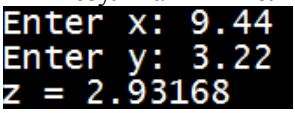


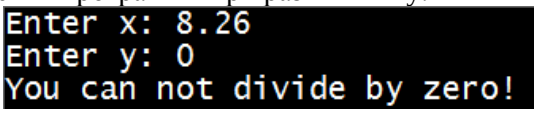
Рис. 4.2. Структурная схема, иллюстрирующая работу условного оператора

Пример. В данной программе вычисляется частное чисел *x* и *y*. Если делитель равен нулю, то выдается соответствующее сообщение, иначе вычисляется частное.

```
#include<iostream>
using namespace std;
int main()
{
float x,y,z; /* Объявление трех переменных типа float */
cout<<"Enter x: ";/* Вывод текстового сообщения */
cin>> x;      /* Ввод с клавиатуры значения переменной x */
cout<<"Enter y: ";//Вывод текстового сообщения
cin>> y;    /* Ввод с клавиатуры значения переменной y */
/* Если y=0, то вывод сообщения о невозможности деления на
ноль */
if (y==0)  cout<<"You can not divide by zero!"<<endl;
else//Иначе, т.е. если y не равен нулю
{z=x/y;     // Вычисление значения переменной z
cout<<"z = "<< z <<endl;} /* Вывод значения z на экран */
return 0; }
```

Результаты выполнения программы при разных x и y:





Если требуется проверить несколько условий, их объединяют знаками логических операций. Например,

```
if (a<b && (a>d || a==0))
b++;
else
{b *= a;
a = 0;}
```

Выражение в примере будет истинно в том случае, если выполнится одновременно условие $a<b$ и одно из условий в скобках. Если опустить внутренние скобки, будет выполнено сначала логическое И, а потом – ИЛИ.

Вторая форма оператора if

Нередко на практике используют комбинацию из нескольких вложенных условных операторов:

if (*выражение*) **{**операторы 1**}**

else if (*выражение*) **{**операторы 2**}**

…
else
{операторы N**}**

Если условный оператор содержит другие условные операторы, то слово **else** соотносится с ближайшим ключевым словом **if**, еще не связанным ни с каким ключевым словом **else**.

На рис. 4.3 представлена структурная схема работы блока из условных операторов.

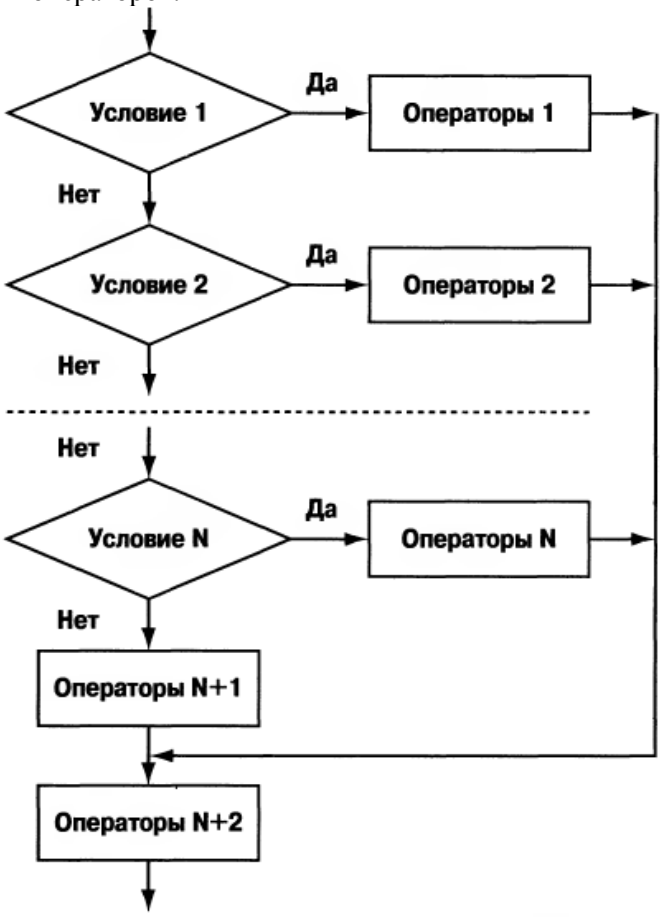


Рис. 4.3. Блок-схема использования нескольких вложенных условных операторов

Пример. Программа, в которой вычисляется значение *Y* в зависимости от переменной *x*:

$$Y = \begin{cases} x + \dfrac{2}{x}, если\ x > 0 \\ \dfrac{4}{x}, если\ 0 \le x \le 1 \\ 2 - x, если\ x < 0 \end{cases}$$

```
1   #include <iostream>
2   using namespace std;
3   int main()
4   {float y,x;
5   cout << "Enter x: ";
6   cin >> x;  //Ввод с клавиатуры переменной x
7   if (x < 0) //Проверка: если x<0
8   { y = 2-x; }// Этот оператор будет выполнен, если x<0
9   //Следующий оператор выполняется если x больше либо равен 0
10  else if (x <= 1)// Проверка: x<=1
11  { y = 4/x;} // Этот оператор будет выполнен, если x находится
12                // в диапазоне (0,1)
13  else
14  {y = x+2/x;}// Этот оператор будет выполнен, если x>1
15  cout << "y= " << y << endl;
16  return 0; }
```

Результаты выполнения программы:

```
Enter x: -10.55
y= 12.55
```

```
Enter x: 0.2
y= 20
```

```
Enter x: 6.5
y= 6.80769
```

Пример. Реализация предыдущей задачи с использованием вложенных операторов **if**.

```
#include<iostream>
int main()
{ float y,x;
std::cout << "Enter x, please:" << std::endl;
std::cin >> x;
if (x>= 0)
{    if (x<=1)
{ y = 4/x; }
     else
{ y=x+2/x;}
}
else
{ y = 2-x; }
std::cout << "y= " << y << std::endl;
return 0;  }
```

4.1.2. Оператор switch

Оператор **switch** (переключатель) обеспечивает многонаправленное ветвление. Его используют вместо нескольких вложенных условных операторов **if** в тех случаях, когда проверяется больше одного условия.

Синтаксис вызова оператора **switch**() следующий:

switch (*выражение***)** {
case *константа1*:
операторы
break;
case *константа2*:
операторы
break;
• • •
default:
операторы
}

В круглых скобках после ключевого слова **switch** указывается *выражение*, значение которого проверяется. Результатом *выражения* может быть целое число или символ. Значение, возвращаемое выражением, сравнивается со значениями, указанными после ключевых слов **case**. Если имеет место совпадение, выполняется

соответствующий блок операторов. Операторы выполняются до конца оператора **switch** или пока не встретится инструкция **break** (в общем случае инструкция **break** используется для выхода из оператора цикла и перехода к следующему оператору). Если значение *выражения* не совпадает ни с одним из константных выражений, то происходит переход к операторам, помеченным меткой **default**. Таких меток внутри переключателя должно быть не более одной. Причем в случае ее отсутствия предполагается, что никаких действий по умолчанию (при несовпадении *выражения* с константными выражениями) не предусмотрено и ни один из внутренних операторов не выполняется.

Сами по себе метки **case** и **default** не изменяют последовательности выполнения операторов. Если не предусмотрены выходы из **case** (**break**), то в нем последовательно выполняются все операторы, начиная с метки, которой передано управление. На рисунке 4.5 представлена схема работы оператора выбора **switch**. Представленная схема приведена в предположении, что в каждом **case**-блоке использована инструкция **break**, а в конце **switch**-оператора использована инструкция **default**. Эта инструкция не является обязательной, также как и инструкции **break**.

*Итак, для применения **switch** – инструкции надо знать следующее:*

- *выражение* можно тестировать только с использованием условия равенства, в то время как условное **if** – выражение может быть любого типа;
- никакие две константы не могут иметь одинаковых значений;
- последовательность инструкций, связанная с каждой **case** – ветвью, не является блоком. Однако полная **switch** – инструкция определяет блок.

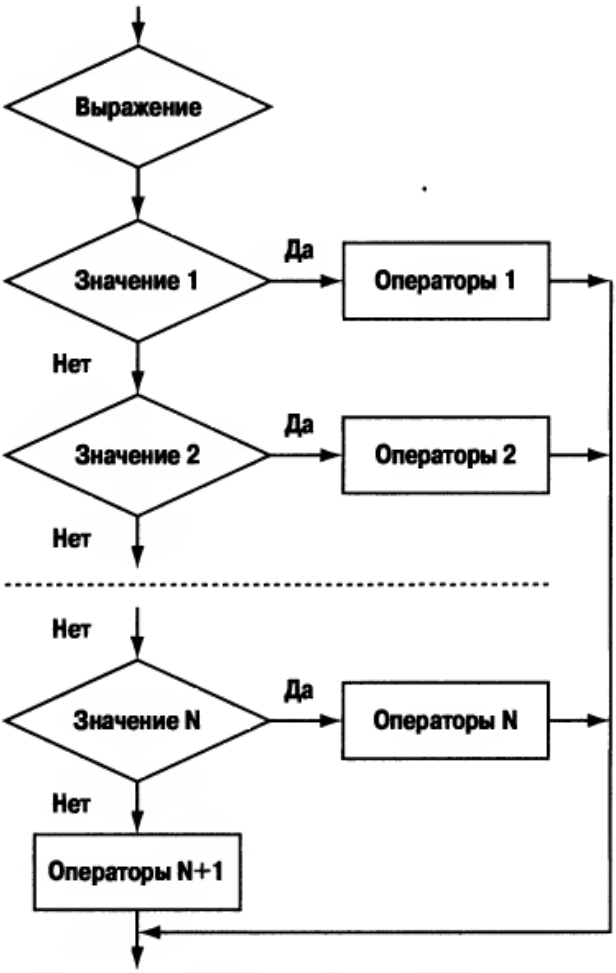


Рис. 4.5. Схема работы оператора выбора switch

Пример. В программе демонстрируется работа оператора выбора **switch**: с клавиатуры вводится символ и в зависимости от введенного символа выводится соответствующее сообщение.

```
#include<iostream>
int main()
{char x; std::cout << "ENTER SIMVOL: "; std::cin >> x;
switch (x)
{case 'A':
std::cout << "A have been chosen! " << std::endl;
break;
case 'B':
std::cout << "B have been chosen! " << std::endl;
break;
case 'C':
std::cout << "C have been chosen! " << std::endl;
break;
default:
std::cout << "You should choose between A, B, and C!";
}
return 0; }
```

Результаты выполнения программы:

```
ENTER SIMVOL: A
A have been chosen!
```

```
ENTER SIMVOL: C
C have been chosen!
```

```
ENTER SIMVOL: B
B have been chosen!
```

```
ENTER SIMVOL: b
You should choose between A, B, and C!
```

4.2. Операторы организации циклов

Операторы цикла позволяют многократно выполнять серии однотипных действий. Действия выполняются до тех пор, пока остается справедливым (или пока не будет выполнено) некоторое условие. Существуют три формы оператора цикла: **for**, **while** и **do while**.

4.2.1. Оператор for

Синтаксис оператора **for** имеет вид:

for (инициализация переменных; выражение; изменение переменных**)**

{ операторы **}**

В круглых скобках после ключевого слова **for** указывается программный код из трех блоков (при этом каждый из блоков может быть пустым). Блоки разделяются точкой с запятой. *Первый блок* является блоком инициализации. В нем обычно присваиваются на-

чальные значения для переменной (или переменных) цикла. *Второй блок* – условие выполнения операторов цикла. Пока справедливо условие, операторы цикла будут выполняться. *Третий блок* – это блок изменения индексных переменных. Если среднее выражение отсутствует, то по умолчанию оно принимается равным 1 (истинно), и в этом случае цикл выполняется бесконечно. Схема работы оператора цикла **for** представлена на рис. 4.6.

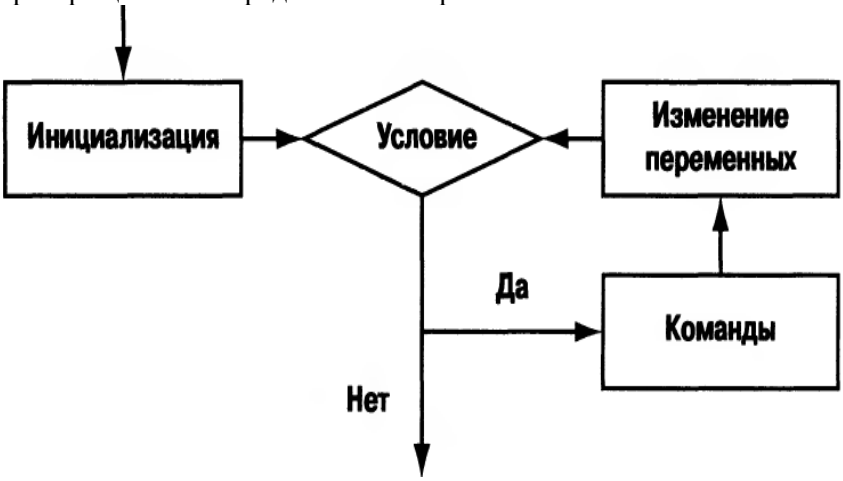


Рис. 4.6. Схема работы оператора цикла for

Пример. Вычислить сумму *n* целых чисел.

```
#include <iostream>
using namespace std;
int main()
{  // Объявление переменных,
int n, i, s=0;  // причем s инициализируется значением 0
cout  <<  "Enter number n=" ;  /* Вывод текстового сообщения */
cin >> n; // Ввод значения в переменную n
for (i=1; i<=n; i++) // Цикл будет выполняться n раз
          {s+=i; // Увеличение переменной s на текущее значение i
    cout << "i= " << i ; // Вывод текущего значения переменной i
    cout << "  s= " << s << endl; //Вывод текущего значения переменной s
          }
// Вывод итогового значения  s
cout << "Sum of natural numbers is: " << s<< endl;
return 0;}
```

Результат выполнения программы будет иметь вид:

```
Enter number n=5
i= 1   s= 1
i= 2   s= 3
i= 3   s= 6
i= 4   s= 10
i= 5   s= 15
Sum of natural numbers is: 15
```

Пояснение к программе. Основу программы составляет оператор цикла, который содержит в первом блоке команду инициализации индексной переменной i=l начальным единичным значением. Второй блок – проверяемое условие i<=n. Это означает, что оператор цикла выполняется до тех пор, пока индексная переменная i не превышает значения переменной n (значение переменной предварительно вводится с клавиатуры). В третьем блоке указана инструкция i++, в силу чего значение индексной переменной на каждом шаге увеличивается на единицу. Наконец, в основном блоке оператора цикла (в фигурных скобках) использована команда s+=i, которой на каждом шаге целочисленная переменная s (начальное нулевое значение этой переменной установлено при ее объявлении) увеличивается на значение индексной переменной i и выводится на экран. Процесс будет продолжаться до тех пор, пока значение индексной переменной не превысит значения переменной n. Таким образом, после выполнения оператора цикла значение переменной s определяется суммой натуральных чисел от 1 до n.

Управляющая переменная цикла **for** может изменяться как с положительным, так и с отрицательным приращением, причем величина этого приращения может быть любой.

Пример. Вывести в столбец числа от 50 до -50 с шагом 10.

```
1  #include<iostream>
2  int main()
3  {
4  for (int i=50;i>=-50;i-=10)
5  std::cout <<i<< std::endl;
6  return 0;}
```

4.2.2. Оператор while

Синтаксис оператора **while** имеет следующий вид:

while (*условие***)**

{ операторы **}**

Сначала проверяется *условие*, указанное в круглых скобках после ключевого слова **while**. Если *условие* справедливо, поочередно выполняются *операторы*, указанные в фигурных скобках. Если оператор один, фигурные скобки можно не указывать. *Операторы* выполняются до тех пор, пока *условие* истинно. Значение выражения условия вычисляется перед каждым выполнением операторов цикла, который, таким образом, будет выполняться ноль или более раз (не выполнится ни разу в случае ложности условия при входе в цикл). Схема выполнения оператора цикла **while** показана на рис. 4.7.

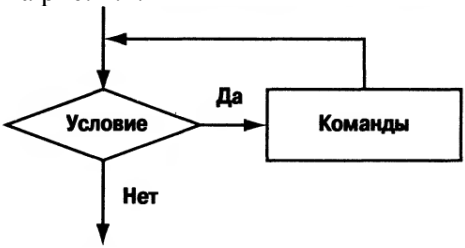


Рис. 4.7. Схема выполнения оператора цикла while

Пример. Программа для вычисления суммы целых чисел с помощью оператора цикла **while**.

```
1  #include <iostream>
2  using namespace std;
3  int main () {
4  int n, i=1,s=0;
5  cout << "Enter number n=";
6  cin >> n;
7  // Операторы цикла будут выполняться пока i<=n
8  while (i<=n){
9  s+=i;
10 i++;
11 cout<< "Sum of natural numbers is: "<< s;
12 return 0;}
```

Результат выполнения программы будет иметь вид:

```
Enter number n=5
Sum of natural numbers is: 15
```

4.2.3. Оператор do while

Синтаксис оператора **do while**:

do {
операторы
}
while (*условие*)

В операторе цикла **do while** выполняемые операторы (заключенные в фигурные скобки) указываются после ключевого слова **do**. Далее проверяется *условие*, указанное в круглых скобках. Если условие выполняется, снова выполняются команды после ключевого слова **do** и т. д. Операторы выполняются до тех пор, пока *условие* истинно. Значение выражения условия вычисляется и анализируется после каждого выполнения операторов цикла, которые будут выполнены один или более раз. Схема выполнения оператора цикла **do while** показана на рис. 4.8.

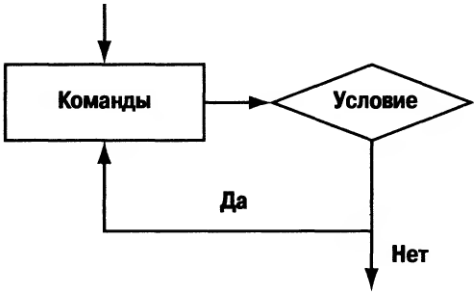


Рис. 4.8. Схема выполнения оператора цикла do while

Принципиальная разница между операторами **while** и **do while** состоит в том, что в первом случае сначала проверяется условие, а затем (если верно условие) выполняются операторы. Во втором случае, сначала, по крайней мере один раз выполняются операторы, а затем проверяется условие. По сравнению с оператором цикла **for**, операторы **while** и **do while** требуют от программиста большей ответственности в первую очередь в плане детальной

проработки механизма изменения значения проверяемого условия в процессе выполнения команд основного блока оператора. Программный код должен быть составлен корректно, чтобы не получить бесконечный цикл.

Пример. Программа для вычисления суммы целых чисел с помощью оператора цикла **do while**.

```
#include <iostream>
using namespace std;
int main () {
int n,i=1,s=0;
cout << "Enter number n=";
cin >> n;
do {
s+=i;
i++; }
while(i<=n);
cout<< "Summa of natural numbers is: "<< s;
return 0;}
```

Результат выполнения программы будет иметь вид:

```
Enter number n=5
Sum of natural numbers is: 15
```

Пояснение к программам. На первый взгляд может показаться, что разницы между двумя программами с циклами **while** и **do while** практически никакой, но одно принципиальное отличие все же есть. Если пользователь введет, например, отрицательное число (значение переменной n), то первая программа в качестве значения суммы укажет 0, в то время как во втором случае будет выведена 1. Причина в том, что в цикле **while** при ложном проверяемом условии команды оператора цикла не выполняются и в качестве значения суммы возвращается начальное нулевое значение переменной s. Во втором случае сначала выполняется один цикл и уже после этого проверяется условие. За этот один выполненный цикл значение переменной s увеличивается на 1, и в результате программой для суммы натуральных чисел возвращается единичное значение.

5. УКАЗАТЕЛИ

5.1. Понятие переменных-указателей

Одной из наиболее мощных возможностей языка C++ является непосредственный доступ к памяти при помощи указателей. Этим язык C++ превосходит некоторые другие языки. В то же время, использование указателей – одна из наиболее подверженных ошибкам областей программирования. Правильное применение требует отличного понимания того, как компилятор управляет распределением памяти. *Переменная (variable)* – это именованная область памяти, в которой могут храниться различные значения. С точки зрения программиста *переменная* это место в памяти компьютера, где можно размещать хранимое значение, а затем извлекать его оттуда.

Указатель (*pointer*) – это переменная, значением которой является адрес другой переменной, то есть адрес области в оперативной памяти. Концептуально каждый указатель состоит из двух частей: *области памяти* и *знания*, как следует интерпретировать содержимое этой области. Как хранятся переменные? Что такое адрес области в памяти (memory address)? Ответ на эти вопросы требует рассмотрения устройства компьютерной памяти. *Область памяти* – это адрес, часто представленный в шестнадцатеричном виде. В 32-разрядном процессоре адрес будет 32-битным числом, например 0x0001EA40. Сам по себе указатель содержит только этот адрес. Чтобы обратиться к данным, на которые этот указатель указывает, надо пойти по этому адресу и как-то интерпретировать содержимое памяти в этой области. Сам по себе этот участок памяти – просто набор битов. Чтобы он обрел смысл, его надо как-то истолковать. Информация о том, как интерпретировать содержимое области памяти, предоставляется основным типом указателя. Если указатель указывает на целое число, то на самом деле это значит, что компилятор интерпретирует область памяти, задаваемую указателем, как целое число.

Оперативная память компьютера – это хранилище значений переменных. Она разделена на последовательно пронумерованные ячейки, каждая из которых занимает 1 байт или 8 бит. Номера ячеек называют *адресами памяти*. Переменные имеют не только

адреса, но и имена. Имя переменной можно представить себе как надпись на ячейке, по которой ее можно найти, даже не зная настоящего номера (адреса в памяти).

Каждая переменная в зависимости от ее типа размещается в одной или нескольких последовательно расположенных отдельных ячейках памяти, адрес первой из них является адресом переменной в памяти. На рисунке 5.1 схематически представлена эта идея. Согласно рисунку, переменная *Line_Count* начинается с ячейки с адресом 203. В зависимости от своего размера, переменная *Line_Count* может занимать одну или несколько ячеек памяти.

Имя переменной				*Line_Count*			
Память				00000000	00000101		
Адреса ячеек памяти	200	201	202	203	204	205	206

Рис. 5.1. Расположение переменных в памяти компьютера

Когда разработчик объявляет переменную, например типа **int**, он сообщает компилятору о необходимости зарезервировать для нее в памяти 4 байта. Компилятор соотносит с именами переменных определенные адреса в памяти и, начиная с этих адресов, выделяет участки памяти (в байтах) в соответствии с тем, какого типа объявлены переменные. Заботу о том, по какому именно адресу будет размещена переменная, берет на себя операционная система.

Типичная компьютерная система содержит массив последовательно пронумерованных (адресуемых) ячеек памяти, с которыми можно работать по отдельности либо целыми непрерывными группами. Указатель представляет собой группу ячеек памяти, которые содержат адрес какой-либо переменной.

Переменные-указатели (или переменные типа указатель) должны быть объявлены в программе. Формат объявления переменной-указателя:

тип ***имя_переменной*;**

При объявлении указателя задают тип переменной, на которую он должен указывать. Это позволяет уведомить компилятор о том, как именно воспринимать область памяти, на которую он указывает. Сам указатель содержит лишь адрес. Когда создается указатель, компилятор выделяет для него в памяти объем, достаточный для хранения адреса, размер которого соответствует используемым аппаратным средствам и операционной системе. Размер указателя может и не совпадать с размером целого числа, кроме того, поскольку размер указателя зависит от конкретной системы, бессмысленно делать предположения о его размере.

Пример. Объявление указателей.

```
/*Символ «звездочка» (*) свидетельствует о том, что
   это указатель */
int *ptr1;      /*Объявление указателя ptr1 на int –
   значение */
double *ptr2;  /*Объявление указателя ptr2 на double
   – значение */
```

Пояснение. Переменная *ptr1* объявляется указателем на тип **int**. В результате будет получена переменная *ptr1*, предназначенная для хранения адреса значения типа **int**. Переменная *ptr2* объявляется указателем на тип **double**. В результате будет получена переменная *ptr2*, предназначенная для хранения адреса значения типа **double**. Когда переменная объявляется указателем на какой-либо тип, это значит, что она будет хранить адрес переменной определенного типа.

Поскольку указатели – это не более чем обычные переменные, им можно присваивать любые имена, допустимые для других переменных. Но есть негласное правило соглашения об именновании – имена всех указателей пишут с маленькой буквы p (от *pointer* – указатель), например, *pAge, pNumber*.

После объявления указателю обязательно нужно присвоить какое-либо значение. Если предназначенный для хранения в указателе адрес заранее не известен, ему следует присвоить значение NULL. Указатели, значения которых равны NULL, называют

пустыми (null pointer). Неинициализированные указатели называются ***дикими*** (wild pointer). Они очень опасны.

Указатели принято инициализировать при объявлении либо нулем (то есть указатель не содержит адреса и не указывает ни на какой объект), либо адресом другой переменной.

Пример. Объявление и инициализация указателей.

```
#include <iostream>
using namespace std;
int main()
{
int *ptx = 0; // указатель ptx инициализируется при объявлении нулем
int *ptz;    // указатель ptz не инициализируется при объявлении

/* Примечание. Для доступа к значению, хранящемуся в указателе,
 используется только его имя без знака звездочка  */
cout << "Value ptx =" << ptx << endl; //Вывод значения указателя ptx
cout << "Value ptz =" << ptz << endl; //Вывод значения указателя ptz
int x;
int *pty = &x; // указатель pty инициализируется при объявлении
               // адресом переменной x
ptz = &x; //объявленному в строке 6 указателю ptz присваивается
          //адрес переменной x
cout << "Value pty =" << pty << endl;//Вывод значения указателя pty
cout << "Value ptz =" << ptz << endl;//Вывод нового значения указателя ptz
return 0;}
```

Результат выполнения программы:

```
Value ptx =0
Value ptz =0x22ff28
Value pty =0x22ff00
Value ptz =0x22ff00
```

Примечание. Полученный результат может выглядеть иначе, поскольку каждый компьютер сохраняет переменные по разным адресам, в зависимости от того, что еще находится в памяти и сколько ее доступно.

5.2. Операторы, используемые при работе с указателями

С указателями используются следующие операторы:

& – оператор раскрытия указателя, который служит для получения адреса переменной. Например, если *Dec* – переменная, то операция *&Dec* позволяет получить адрес этой переменной;

***** – оператор косвенного доступа, также называется оператором взятия значения, разыменовывания или ссылкой.

Оператор косвенного доступа (*) используется двумя способами: при объявлении и при косвенном доступе. *При объявлении* он свидетельствует о том, что это указатель, а не обычная переменная, например:

```
int *pAge = 0; /* Объявить указатель и инициализировать нулем */
```

При косвенном доступе звездочка означает, что речь идет о значении, которое находится в памяти по адресу, хранящемуся в переменной-указателе, а не о самом адресе. Иначе говоря, при извлечении значения из указателя будет возвращено то значение, которое хранится по адресу, содержащемуся в указателе, например:

```
*pAge = 5; /* разместить значение 5 по адресу, находящемуся в pAge */
```

Если *pAge* – указатель, то операция **pAge* – получение значения, которое расположено по адресу памяти, который содержится в указателе *pAge*.

Примечание. Тот же символ (*) используется как оператор умножения. Что именно имел в виду программист, поставив звездочку, компилятор определяет, исходя из контекста.

Указатель обеспечивает лишь косвенный доступ к значению переменной, адрес которой он хранит. Чтобы присвоить переменной *yourAge* значение переменной *ShowOld* с помощью указателя *pAge*, применяется следующий программный код:

```
int SchowOld =40;   // объявить переменную SchowOld и присвоить ей  значение
int *pAge = &SchowOld; /*  объявить указатель pAge и присвоить ему адрес
переменной SchowOld */
int yourAge; // объявить переменную yourAge
yourAge= *pAge; /*  переменной yourAge присвоить значение, которое
находится по адресу, хранящемуся в указателе pAge, т.е. значение
переменной SchowOld, равное 40 */
```

Оператор косвенного доступа (*) перед переменной *pAge* означает, что «само значение находится по адресу...». Иными словами, «возьмите значение, хранимое по адресу, который находится в переменной *pAge*, и присвойте его переменной *yourAge*».

Пример. Присвоение указателям адресов переменных, косвенное обращение к переменным через указатели.

```
#include <iostream>
using namespace std;
int main()
{
int DEC=125; // Объявляется и инициализируется переменая DEC
int *Ptr=0; // Объявляется и инициализируется указатель Ptr
Ptr=&DEC; //Указателю присваивается адрес переменной DEC
DEC=DEC+25; // Переменная DEC увеличивается на число 25
//Далее выводится значение переменной DEC
cout << "Value DEC  = " << DEC << endl;
/*Далее выводится значение, расположенное по адресу, содержащемуся
   в переменной-указателе Ptr */
cout << "Value *Ptr = " << *Ptr << endl;
//Далее выводится адрес переменной DEC
cout << "Adress DEC = " << &DEC << endl;
//Далее выводится значение переменной-указателя Ptr
cout << "Value Ptr  = " << Ptr << endl;
//Далее выводится адрес переменной-указателя Ptr
cout << "Adress Ptr = " << &Ptr << endl;
return 0;}
```

Консольное окно с результатами выполнения программы:

```
Value DEC  = 150
Value *Ptr = 150
Adress DEC = 0x22ff0c
Value Ptr  = 0x22ff0c
Adress Ptr = 0x22ff08
```

Очень важно понимать разницу между указателем, хранимым в нем адресом и значением, расположенным по адресу, который содержится в этом указателе. Обычно понимание этого и составляет основную сложность при изучении указателей.

Поскольку указатели сами являются переменными, их можно употреблять без разыменовывания.

Пример. Присваивание указателю значения другого указателя.

```
#include <iostream>
using namespace std;
int main()
{double a,b; // Объявление переменных a и b
double *pa = &a; // Объявление указателя pa и инициализация адресом a
double *pb = &b; // Объявление указателя pb и инициализация адресом b
cout << "Value pa  = " << pa << endl; //Вывод значения pa, т.е. адреса a
cout << "Value pb  = " << pb << endl; //Вывод значения pb, т.е. адреса b
*pa = 5.55; //Через указатель pa присваивается число 5.55 переменной a
*pb=*pa + 6.57;//Через указатель pb присваивается значение переменной b
cout << "Value *pa  = " << *pa << endl;//Выводится значение, расположенное по
      //адресу, который хранится в указателе pa, т.е. значение переменной a
cout << "Value *pb  = " << *pb << endl;//Выводится значение, расположенное по
      //адресу, который хранится в указателе pb, т.е. значение переменной b
pb=pa; // Указателю pb присваивается значение указателя pa
cout << "Value pa  = " << pa << endl;// Вывод значения pa
cout << "Value pb  = " << pb << endl;// Вывод значения pb
return 0;}
```

Результат выполнения программы.

```
Value pa  = 0x22fef0
Value pb  = 0x22fee8
Value *pa  = 5.55
Value *pb  = 12.12
Value pa  = 0x22fef0
Value pb  = 0x22fef0
```

Пояснение. Указатель *pb* указывает на такую же ячейку, что и *pa*, но значения переменных *a* и *b* не изменились. Заметьте кардинальное отличие от операции **pb* = **pa*, когда меняется значение переменной *b*, но значение самого указателя остается неизменным. Проверить это можно, заменив строку *pb* = *pa* строкой **pb* = **pa*.

Следует заметить, что любой указатель может указывать только на объекты одного конкретного типа данных, заданного при его объявлении. То есть указатель типа **double** не может указывать на тип **int**. Исключением является только указатель на тип **void**, в котором может содержаться произвольный адрес без указателя на тип данных. Однако по указателям этого типа нельзя ссылаться и получать значения.

5.3. Арифметические действия с указателями

С указателями можно использовать только четыре арифметических оператора: `++`, `--` , `+` и `-`. Чтобы лучше понять, что происходит при выполнении арифметических действий с указателями, начнем с примера. Пусть *p1* – указатель на **int**-переменную с текущим значением 2000 (то есть *p1* содержит адрес 2000). После выполнения (в 32-разрядной среде) выражения *p1*++ содержимое переменной-указателя *p1* станет равным 2004, а не 2001. Дело в том, что при каждом инкрементировании указатель *p1* будет указывать на *следующее* **int**-значение. Для операции декрементирования справедливо обратное утверждение, то есть при каждом декрементировании значение *p1* будет уменьшаться на 4. Например, после выполнения инструкции *p1*--; указатель *p1* будет иметь значение 1996, если до этого оно было равно 2000. Вышесказанное продемонстрировано на рис. 5.2.

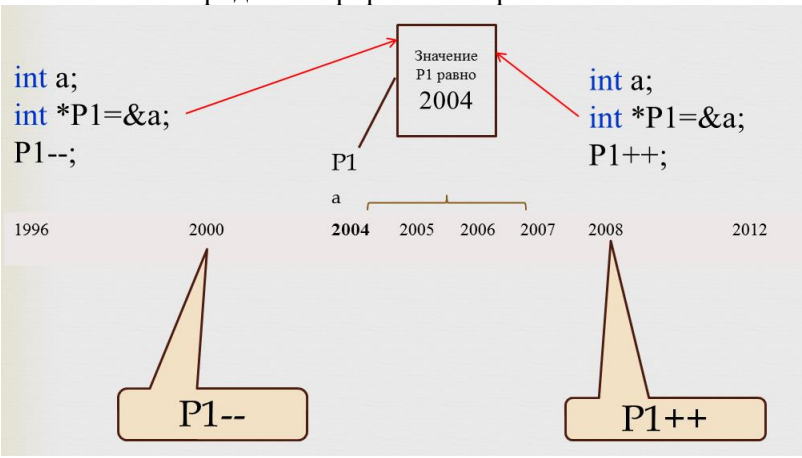


Рис. 5.2. Инкрементирование и декрементирование указателей

Итак, каждый раз, когда указатель инкрементируется, он будет указывать на область памяти, содержащую следующий элемент базового типа этого указателя. А при каждом декрементировании он будет указывать на область памяти, содержащую предыдущий элемент базового типа этого указателя.

Для указателей на символьные значения результат операций инкрементирования и декрементирования будет таким же, как при

«нормальной» арифметике, поскольку символы занимают только один байт. Но при использовании любого другого типа указателя при инкрементировании или декрементировании значение переменной-указателя будет увеличиваться или уменьшаться на величину, равную размеру его базового типа.

Выражение *p1=p1+9* заставляет *p1* ссылаться на девятый элемент базового типа указателя p1 относительно элемента, на который *p1* ссылался до выполнения этой инструкции.

Несмотря на то, что складывать указатели нельзя, один указатель можно вычесть из другого (если они оба имеют один и тот же базовый тип). Разность покажет количество элементов базового типа, которые разделяют эти два указателя.

Помимо сложения указателя с целочисленным значением и вычитания его из указателя, а также вычисления разности двух указателей, над указателями никакие другие арифметические операции не выполняются.

5.4. Стек и динамически распределяемая память

Программы взаимодействуют с пятью следующими областями памяти: с областью глобальных переменных, с динамически распределяемой памятью, с регистрами, с сегментами программного кода, со стеком.

Локальные переменные и параметры функций размещаются в стеке. Объектный код программ размещается в сегментах. Глобальные переменные – в области глобальных переменных. Регистры используются для внутренних целей процессора. Остальная часть памяти составляет так называемую динамически распределяемую память (свободное хранилище, *heap*).

Особенностью *локальных* переменных является то, что после выхода из функции, в которой они были объявлены, память, выделенная для их хранения, освобождается, а значения переменных уничтожаются. *Глобальные* переменные позволяют частично решить эту проблему ценой неограниченного доступа к ним из любой точки программы, что значительно усложняет восприятие текста программы. Использование динамической памяти полностью решает обе проблемы. Динамически распределяемую память можно представить как огромный массив последовательно пронумерованных ячеек, предназначенных для хранения данных.

🔔 **Ячейкам динамической памяти нельзя присвоить имя** (в отличие от стека). Но можно, зарезервировав определенное количество ячеек, запомнить адрес первой из них в указателе. Когда функция возвращает значение, стек очищается автоматически. Когда область видимости локальных переменных заканчивается, они также удаляются из стека. Но динамическая память не очищается до завершения самой программы. Поэтому ответственность за освобождение всей памяти, зарезервированной под все использованные данные, ложится на программиста.

Важное преимущество динамической памяти: выделенная в ней область не может использоваться в других целях до тех пор, пока не будет освобождена явно. Поэтому, если во время работы функции в динамической памяти выделяется область, ее можно использовать даже по завершении работы функции. Недостаток динамической памяти: выделенные в ней участки остаются зарезервированными до тех пор, пока они не будут освобождены явно. Если этого не сделать, области памяти так и останутся занятыми, что через какое-то время способно исчерпать ресурсы системы.

Еще одним преимуществом динамического выделения памяти по сравнению с глобальными переменными является то, что доступ к данным можно получить только из тех функций, которые обладают доступом к указателю, хранящему нужный адрес. Это позволяет жестко контролировать манипулирование данными, а также избегать нежелательного или случайного их изменения.

5.5. Указатели и объекты в динамической памяти

Для работы с динамической памятью (heap) используются указатели. Чтобы определить переменную в динамической памяти необходимо:

1. Определить указатель на соответствующий тип.
2. Запросить память для переменной этого типа при помощи оператора **new.**

Ключевое слово **new** используется для выделения участка в динамически распределяемой области памяти, после которого указывают тип размещаемого в памяти объекта. Так, компилятор определяет размер необходимой для размещения объекта в области памяти. В качестве результата оператор **new** возвращает адрес

выделенного фрагмента памяти, который должен быть присвоен указателю.

Пример. Объявить переменную типа **double** в динамической памяти.

1-й способ.

```
double *pa = 0;
pa = new double;
```

2-й способ.

```
double *pa = new double;
```

Операция **new** возвращает адрес памяти из свободного хранилища, выделенный для размещения переменной типа **double**, и этот адрес сохраняется в указателе *pa*. Затем этот указатель, как было показано выше, применяется для работы с данной переменной.

Пример. Переменной, расположенной по адресу, содержащемуся в *pa*, присвоить число 7.65

```
*pa = 7.65;
```

Эту строку можно прочитать так: "Присвоить число 7.65 значению указателя pa" или "Разместить число 7.65 в той области динамически распределяемой памяти, на которую указывает pa".

Если оператор **new** не сможет выделить место в динамически распределяемой памяти (в конце концов память – ограниченный ресурс), то он передаст исключение. *Исключения* – это объекты, позволяющие отреагировать на ошибки.

Переменную можно сразу инициализировать операцией **new** при ее создании.

Пример. Объявить переменную типа **double** в динамической памяти и инициализировать ее числом 7.65.

1-й способ.

```
double *pa = 0;
pa = new double(7.65);
```

2-й способ.

```
double *pa=new double(7.65);
```

Когда отпадает необходимость в переменной, определенной динамически, следует:

1. Освободить память, на которую указывает указатель, при помощи оператора **delete**:

delete *имя_указателя*;

2. Присвоить указателю нулевое значение или, если планируется его использовать далее, присвоить адрес новой переменной.

Поскольку распределенная с помощью оператора **new** память автоматически не освобождается, в случае потери адреса ее не удастся ни удалить, ни использовать. Такой участок памяти становится абсолютно недоступным, а подобная ситуация называется *утечкой памяти* (memory leak).

Пример. Удалить переменную, определенную динамически.

1-й способ.
```
delete pa;
pa = 0;
```

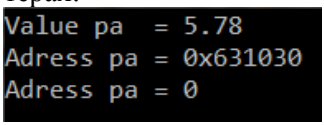
2-й способ.
```
delete pa;
pa = new double;
```

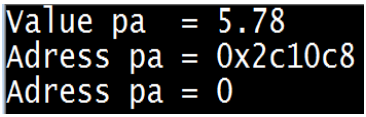
При удалении указателя с помощью оператора **delete** происходит реальное освобождение участка памяти, адрес которого содержится в указателе. Иными словами, отдается команда: *«Вернуть в динамически распределяемую память участок, на который указывает этот указатель»*. Но сам указатель остается (ведь это обычная переменная), и ему может быть передан на хранение другой адрес. В первом случае *pa* присваивается нулевое значение, *pa* не указывает ни на какую ячейку. Во втором случае оператор **delete** освобождает ячейку памяти, на которую указывает *pa*, а оператор **new** определяет новую ячейку памяти для переменной типа **double**.

Пример. Демонстрация использования динамической памяти.

```
#include <iostream>
using namespace std;
int main()
{
double *pa = 0; // Объявление указателя и инициализация его нулем
pa = new double; /* Выделение участка в динамически распределяемой
      области памяти для указателя pa */
*pa = 5.78; /* Переменной, расположенной по адресу, содержащемуся в pa,
             присвоить число 5.78  */
cout << "Value pa  = " << *pa << endl; // Вывести значение переменной,
                                       // на которую указывает pa
cout << "Adress pa = " << pa << endl; // Вывести значение указателя
delete pa; //Освободить память, на которую указывает указатель pa
pa = 0; // Присвоить указателю нулевое значение
cout << "Adress pa = " << pa << endl; // Вывести значение указателя
return 0;}
```

Результаты выполнения программы на двух разных компьютерах:





6. МАССИВЫ

Массив – это последовательность переменных одного типа, использующая одно имя. Обращение к элементам массива осуществляется с помощью индексов. *Индекс* – ссылка на конкретное значение в массиве, то есть индекс определяет позицию элемента внутри массива. В C++ все массивы используют *ноль* в качестве индекса своего первого элемента! Массивы удобны для хранения большого количества взаимосвязанных значений. Для современного программирования массивы являются фундаментальными объектами.

6.1. Одномерные массивы

Объявление одномерного массива имеет следующий вид:

тип_массива *имя_массива* [*размер*]

Размер массива – целое число, хотя может быть и произвольным константным выражением, величина которого должна быть определена перед началом выполнения программы.

Пример. Объявить одномерный массив целых чисел с именем *Vek* для хранения трех элементов.

```
int Vek [3]; /* Объявление массива Vek из трех элементов (Vek[0], Vek[l] и Vek[2]) типа int */
```

Пример. Объявить массив *my_array* с размерностью, заданной константой и присвоить значения элементам массива.

```
1  #include<iostream>
2  int main()
3  {
4  const int n = 3; //Объявляется константа n
5  int my_array [n]; //Объявляется массив my_array размерностью n
6  my_array[0] = 12; // Присваивается значение 1-му элементу
7  my_array[1] = 1245;
8  my_array[2] = 18094;
9  return 0; }
```

Объявление вида *int my_array[n];* без предварительной инициализации переменной *n* и объявления ее константой вызовет ошибку компиляции. Запрещено также объявление массива без

указания размерности, за исключением объявления с инициализацией. Это происходит от того, что память под все элементы массива в стеке выделяется на этапе компиляции. Исключение составляет объявление массива в динамической памяти. Далее память, используемая для хранения массива, должна оставаться выделенной на все время существования массива.

🔔 ***В C++ не выполняется никакой проверки «нарушения границ» массивов***, то есть ничего не может помешать программисту обратиться к массиву за его пределами: компилятор запустит такой код на выполнение, но результаты могут быть непредсказуемые.

6.2. Двумерные массивы

C++ допускает возможность создания многомерных массивов. Простейшей формой такого массива является двумерный массив. Двумерный массив можно трактовать как двумерную матрицу, состоящую из строк и столбцов. Объявление двумерного массива имеет следующий вид:

тип_массива *имя_массива*[*число строк*][*число столбцов*]

Левый индекс нумерует строки, а правый столбцы. В отличие о других языков программирования, которые для разделения размерностей массива используют запятые, в C++ каждый размер берется в квадратные скобки.

Пример. Объявить массив *my_array* из 3-х строк и 4-х столбцов.

1-й способ.

```
/* Размерность двумерного
массива определяется целыми
положительными числами */
int my_array [3][4];
```

2-й способ.

```
/*Размерность двумерного
массива определяется целыми
положительными константами */
const int n = 3;
const int n = 4;
int my_array [n][m];
```

Доступ к элементам массива осуществляется по двум индексам, например, элементами массива *my_array* будут:

my_array[0][0]	my_array[0][1]	my_array[0][2]	my_array[0][3]
my_array[1][0]	my_array[1][1]	my_array[1][2]	my_array[1][3]
my_array[2][0]	my_array[2][1]	my_array[2][2]	my_array[2][3]

6.3. Инициализация одномерных массивов

В C++ предусмотрено несколько способов инициализации массивов.

1-й способ – *инициализация при объявлении,* который предусматривает следующие варианты:

а) *инициализация при объявлении с указанием размерности*, например:

```
int my_array[3] = {12, 1245, 18094};

// или

const int n = 3;
int my_array [n] = {12, 1245, 18094};
```

Примечание. Фрагмент кода, который приводится ниже не является правильным, так как инициализировать можно только при объявлении!

```
int our_array [3];

our_array [3]= {12, 1245, 18094}; /* Ошибка компиляции! */
```

б) *Объявление с инициализацией без указания размерности*, например:

```
/* компилятор сам определяет размерность
массива в процессе компиляции  */
int my_array []={12, 1245, 18094};
```

в) Объявление с инициализацией не всех элементов, например:

```
/* в данном случае компилятор инициализирует
 нулями два оставшихся элемента массива*/
int my_array [5]={12, 1245, 18094};
```

2-й способ – *инициализация в цикле*. Для этого используются следующие конструкции:

тип *имя_массива*[*размер*]**;**
for (i=0; i<размер; i++) **cin** >> *имя_массива*[i];

или

тип *имя_массива*[*размер*]**;**
for (i=0; i<размер; i++) *имя_массива*[i] = *значение*;

Первая позволяет вводить элементы массива с клавиатуры, вторая – присваивать значения, полученные, например, в процессе вычисления некоторого выражения.

Пример. Инициализировать массив *my_array* из 5 элементов и заполнить значениями, введя их с клавиатуры.

```
1  #include<iostream>
2  int main()
3  {
4  const int n=5;
5  int my_array [n];
6  for (int i=0; i<n; i++)
7  { std::cout<< "Enter " <<i<<" element =";
8    std::cin >> my_array[i]; }
9  return 0; }
```

Результат выполнения программы:

```
Enter 0 element =7
Enter 1 element =9
Enter 2 element =6
Enter 3 element =4
Enter 4 element =3
```

Пример. Инициализировать массив *my_array* из 5 элементов и заполнить значениями, равными удвоенному значению индекса каждого элемента.

```
#include<iostream>
int main()
{
const int n=3;
int my_array [n]; // Объявление массива
for (int i=0; i<n; i++)  // Оператор цикла
{ my_array[i]= i*2; // Присваивание значения i-му элементу массива
std::cout<< "my_array["<<i<<"] =" << my_array[i] << std::endl; // Вывод i-го элемента
}
return 0; }
```

Результат выполнения программы:

```
my_array[0] =0
my_array[1] =2
my_array[2] =4
```

Работая с массивами, следует иметь в виду серьезное ограничение: в C++ недопустимо копировать один массив в другой с помощью операции присваивания. Например, следующий фрагмент неправильный:

```
int our_array [3];
int my_array [3]={6,8,3};
our_array = my_array; // Ошибка компиляции!!
```

Для передачи содержимого одного массива в другой необходимо скопировать каждый элемент индивидуально.

Пример. Копирование элементов массива *my_array* в массив *our_array*.

```
#include<iostream>
int main()
{
const int n = 5;
int my_array [n]={12, 1245, 18094, 245, 123};
int our_array[n];
for (int i=0; i<n; i++)
{our_array[i] = my_array[i];
// Вывод i-го элемента our_array
std::cout<< "our_array["<<i<<"] =" << our_array[i] << std::endl;}
return 0; }
```

Результат выполнения программы:

```
our_array[0] =12
our_array[1] =1245
our_array[2] =18094
our_array[3] =245
our_array[4] =123
```

6.4. Инициализация двумерных массивов

Многомерные массивы инициализируются так же, как и одномерные.

1-й способ *– инициализация при объявлении:*

а) *объявление с указанием размерности*, например:

```
int my_array [4][3] = { {12, 1245, 18094}, {2, 518, 45},
          {45, 67, 45}, {34, 562, 98} };
/* если опустить внутренние фигурные скобки, то результат не изменится.
В любом случае, сначала будут заполняться строки, а затем столбцы */

// или

const int n = 4;
const int m=3;
int my_array [n][m] = { {12, 1245, 18094}, {2, 518, 45},
          {45, 67, 45}, {34, 562, 98} };
```

б) *объявление с инициализацией без указания первой размерности*, например:

```
int tu_array [ ][3] = { {12, 1245, 18094}, {2, 518, 56},
 {45, 67, 45}, {34, 562, 98} };
 /* Можно не указывать размерность основного массива.
При этом размерности вложенных массивов должны быть указаны. */
```

Например, код приведенный ниже неправильный.

```
int my_array [ ][ ] = { {12, 1245, 18094}, {2,
518, 56},
{45, 67, 45}, {34, 562, 98} }; /*Ошибка
компиляции!!*/
```

в) *объявление с инициализацией не всех элементов*, например:

```
int my_array [4][3] = { {12, 1245, 18094}, {2, 518},
 {45, 67, 45}, {34, 98} };
 /* Можно пропустить инициализацию значений в любой строке,
 при этом значение автоматически будет инициализировано нулем */
```

2-й способ – *инициализация в цикле.* Рассмотрим возможные варианты на примере.

Пример. Объявить массив размерностью 3×4 и ввести его значения с клавиатуры.

```
#include<iostream>
#include<iomanip>
int main()
{
const int row = 3;
const int column = 4;
int data [row][column];
for (int i = 0; i < row; i++)
{
for (int j = 0; j < column; j++)
{ std::cout<< "Enter data["<<i<<"]["  << j<< "]=";
    std::cin >> data [i][j]; }
}
for (int i = 0; i < row; i++)
{
for (int j = 0; j < column; j++)
{
std::cout << std::setw(6) << data [i][j]; }
std::cout << std::endl;
}
return 0; }
```

Результат выполнения программы:

```
Enter data[0][0]=34
Enter data[0][1]=678
Enter data[0][2]=23
Enter data[0][3]=89
Enter data[1][0]=234
Enter data[1][1]=128
Enter data[1][2]=78
Enter data[1][3]=2
Enter data[2][0]=457
Enter data[2][1]=467
Enter data[2][2]=295
Enter data[2][3]=300
    34   678    23    89
   234   128    78     2
   457   467   295   300
```

Обратите внимание на использование вложенных циклов **for** при инициализации многомерных массивов. Массив инициализируется построчно.

6.5. Массивы и указатели

В языке C++ существует тесная связь между массивами и указателями. Любую операцию, выполняемую при помощи индексации массива можно выполнить с применением указателей, причем код с применением указателей обычно работает быстрее, хотя и выглядит более запутанно.

Ели использовать обращение к элементам массива с помощью указателей, то, если *arr* – одномерный массив, обратиться к элементу (*arr[i]*) необходимо с помощью выражения *(*arr*+*i*).

Пример. Задать массив из семи элементов и вывести на экран его 6-й элемент двумя способами.

```
1  #include  <iostream>
2 int main() {
3  int arr[7]={5,10,15,20,21,63,58};
4  std::cout << arr[5] << std::endl;//Вывод 6-го элемента
5  // Его индекс будет равен 5, т.к. индексация элементов начинается с 0
6  std::cout << *(arr+5) << std::endl; //Вывод 6-го элемента
7  return 0;  }
```

В результате на экране будет два раза выдано число – 63.

Такое обращение к элементам массива основано на том факте, что:

- имя массива является указателем на начало массива;
- сами элементы массива располагаются в памяти по порядку.

Следовательно, прибавление к указателю целого числа приводит к смещению адреса, на который указатель указывает, на число позиций слагаемого (рис. 6.1).

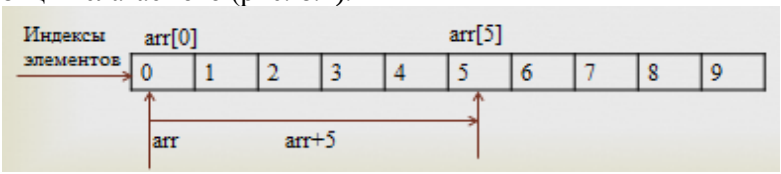


Рис. 6.1. Обращение к элементам массива при помощи указателя

Имея это в виду, легко понять, что если *arr*, например, двумерный массив, то выражение **(*(arr+i)+j)* является элементом *arr[i][j]*, а выражение ***arr* – элементом *arr[0][0]*.

Пример. Задать массив 2*2 и вывести на экран его левый верхний и правый нижний элемент при помощи указателя.

```
1  #include  <iostream>
2  int main() {
3  int a[2][2]={5,10,15,20};
4  std::cout << **a << std::endl;//Вывод элемента a[0][0]
5  std::cout << *(*(a+1)+1 ) << std::endl;//Вывод элемента a[1][1]
6  return 0;   }
```

В результате на экране будут выведены числа 5 и 20.

Принимая во внимание изложенное выше, можно заключить, что если некоторое данное указывает элемент массива с индексом *i*, то добавление к нему целого данного величиной *j* приведет к образованию данного, указывающего на элемент таблицы с индексом *i+j*. Это подтверждает тесную связь операций с указателями и массивами на языке C++.

6.6. Массивы, расположенные в динамической памяти

Массивы являются эффективным подходом к хранению данных, но они имеют серьезный недостаток: необходимо знать при написании программы, насколько большой массив необходим для решения задачи. Следующий код работать не будет!!

```
cin >> n; // Ввод переменной  n с клавиатуры
int my_array [n]; // Ошибка компиляции!!
//Размер массива должен быть константой!
```

Компилятор требует, чтобы размерность массива была константой. Однако, в большинстве случаев неизвестно требуемое количество памяти до запуска программы. Например, надо сохранить массив, который определит пользователь. В этой ситуации возможно:

– определить размерность массива достаточно большим числом, но это приводит к перерасходу памяти;

– при помощи универсальной операции получить память у операционной системы. Данная операция возвращает указатель на начало выделенного блока. В данном случае говорят, что *массив определен в динамической памяти*.

Для определения одномерного массива в динамической памяти используется синтаксис:

тип **имя_массива*=0;
имя_массива = **new тип** [*размер массива*];
или
тип **имя_массива* = **new тип** [*размер массива*];

Например, объявить массив *parr* из 10 элементов в динамической памяти можно следующим образом:

```
//1-й способ                    //2-й способ
double *parr = 0;
parr = new                      double *parr = new
double[10];                     double[10];
```

При этом в динамической памяти выделяется блок, состоящий из десяти элементов типа **double**, а указатель *parr* содержит адрес нулевого элемента массива. Обращение к элементам массива происходит либо посредством разыменовывания указателя, то есть **(parr+i)* является *i*-м элементом массива, либо посредством индексации массива: *parr[i]* также представляет собой значение *i*-го элемента массива.

Отметим два главных отличия массива, расположенного в динамической памяти, и массива, расположенного в стеке:

1) массив, расположенный в стеке может быть инициализирован при объявлении, а массив, определенный в динамической памяти, может быть инициализирован только поэлементно;

2) преимуществом создания массива в динамической памяти является возможность определить его размерность в процессе выполнения программы, а затем уже создать его, например:

```
cin >> n;
int *parr = new int [n];
```

Напомним, что размерность массива, определенного в стеке, объявляется целой константой, инициализированной перед выполнением программы. Как и в случае с обычными переменными,

определенными в динамической памяти, после завершения работы с массивом, его надо удалить из памяти:

delete [ ] **имя_массива*;

Например,

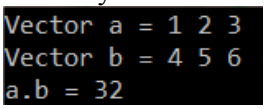
```
delete [ ] parr;
```

🔔 Обратите внимание на квадратные скобки после оператора **delete**. Если их опустить, то произойдет удаление только нулевого элемента массива!!! Память, занимаемая остальными элементами, освобождена не будет. Квадратные скобки сообщают компилятору, что удаляется массив. Компилятор вычисляет размер всех элементов массива и освобождает занимаемую им область динамической памяти. Напомним, что после удаления массива указатель по-прежнему указывает на область памяти, где раньше находился массив. Поэтому указатель надо обнулить, если нет необходимости его использования в дальнейшем, или присвоить ему другой адрес.

Пример. Вычислить скалярное произведение двух векторов. Каждый вектор реализуется как одномерный массив из трех элементов.

```
#include<iostream>
using namespace std;
int main(){
int i;//Индексная переменная
double *a= new double[3];//Первый массив
double *b= new double[3];//Второй массив
double res=0;//Переменная для результата
cout<<"Vector a = ";
for(i=0;i<3;i++) cin>>*(a+i);//Ввод элементов массива a
cout<<"Vector b = ";
for(i=0;i<3;i++) cin>>*(b+i);//Ввод элементов массива b
//Вычисление скалярного произведения
for(i=0;i<3;i++) res+=a[i]*b[i];
//Вывод результата
cout<<"a.b = "<<res<<endl;
delete [] a;
a=0;
delete [] b;
b=0;
return 0;}
```

Результат выполнения программы:



Двумерные массивы можно рассматривать как частный случай одномерного массива указателей. Рассмотрим объявления двух массивов *arr_1* и *arr_2*:

```
const int n = 3; const int m = 4;
double arr_1 [n][m];
double *arr_2[n];
```

Как *arr_1[i][j]* или **(*(arr_1+i)+j)*, так и *arr_2[i][j]* или **(*(arr_2+i)+j)*– вполне законные обращения к одному элементу данных типа **double**. При этом *arr_1* – полноправный двумерный массив; для него выделяется память в размере $n \times m$ ячеек по восемь байт, и для поиска элемента используется перевод координат прямоугольного массива в линейный адрес по формуле $i \times m + j$. А для *arr_2* выделяется *n* ячеек по четыре байта. Если предположить, что каждый элемент *arr_2* указывает на массив из *m* элементов, то получим те же самые $n \times m$ ячеек типа **double** плюс *n* ячеек на указатели. Важное преимущество массива указателей состоит в том, что строки массива могут иметь различную длину, то есть каждый элемент *arr_2* не обязан указывать на массив из десяти элементов. Одни могут указывать на два элемента, другие – на один, а третьи – вообще ни на что. Как было отмечено выше, имя массива является указателем на нулевой элемент. В случае массива указателей элементом является сам указатель, а двумерного массива – строка, то есть массив.

Таким образом, имя двумерного массива или массива указателей имеет тип указателя на указатель, то есть двойного указателя. Следовательно, обращения к элементам двумерного массива осуществляется посредством двойного разыменовывания указателя **(*(arr_1+i)+j)*. Это эквивалентно операторам:

```
arr_1[i][j];
*(arr_1[i]+j);
(*(arr_1+i))[j];
```

Однако, если мы попытаемся присвоить указателю на указатель имя двумерного массива, то компилятор выдаст ошибку:

```
double ** pp =  arr_1; // Ошибка компиляции!!
double ** pp =  arr_2; // Правильно!
```

Пример. Массив указателей определяется в стеке, а массивы, на которые эти указатели указывают в динамической памяти инициализируются после ввода массива.

```
#include<iostream>
#include<iomanip>
int main()
{const int row = 3;int column = 4;
int *data [row]; //Объявление одномерного массива указателей размерности row
for (int i=0; i<row; i++)
{ *(data+i)=new int[column]; //Инициализация i-го элемента массива data (i-го указателя)
     //адресом нулевого элемента динамического массива размерности column
for (int j=0; j<column; j++)
{std::cout << "Enter " << i << " " << j<< " elements of data: ";
std::cin >> *(*(data+i)+j); // Ввод элементов массива
}
}
for (int i=0; i<row; i++)
{
for (int j=0; j<column; j++)
{std::cout << std::setw(3) << *(*(data+i)+j);} //Вывод строки массива
 std::cout << std::endl;
}
for (int i=0; i<row; i++)
{
delete [ ]*(data+i); // Удаление массивов из динамической памяти
}
return 0;
}
```

Результат выполнения программы:

```
Enter 0 0 elements of data: 56
Enter 0 1 elements of data: 45
Enter 0 2 elements of data: 25
Enter 0 3 elements of data: 236
Enter 1 0 elements of data: 554
Enter 1 1 elements of data: 856
Enter 1 2 elements of data: 458
Enter 1 3 elements of data: 45
Enter 2 0 elements of data: 555
Enter 2 1 elements of data: 412
Enter 2 2 elements of data: 23
Enter 2 3 elements of data: 35
  56   45   25  236
 554  856  458   45
 555  412   23   35
```

Важно!!	**В динамической памяти двумерный массив можно определить только как одномерный массив указателей!**

```
int n = 5;
double ** my_array = new double* [n];
```

Пример. Массив указателей, и сами массивы определены в динамической памяти.

```
#include<iostream>
#include<iomanip>
int main()
{int row = 3;int column = 4;
int **data = new int*[row]; /*Объявление массива
    указателей в динамической памяти размерностью row */
for (int i=0; i<row; i++)
{
*(data+i)=new int[column]; /* Каждый элемент массива
  инициализируется адресом нулевого элемента массива
  размерности column */
for (int j=0; j<column; j++)
{   std::cout << "Enter " << i << " " << j << " elements of data:";
    std::cin >> *(*(data+i)+j); }
}
for (int i=0; i<row; i++)
{
for (int j=0; j<column; j++)
{   std::cout << std::setw(3) << *(*(data+i)+j);}
std::cout << std::endl; }
for (int i=0; i<row; i++) { delete [ ]*(data+i); }
delete [ ] data; data=0;
return 0; }
```

Результат выполнения программы:

```
Enter 0 0 elements of data:5
Enter 0 1 elements of data:9
Enter 0 2 elements of data:4
Enter 0 3 elements of data:2
Enter 1 0 elements of data:4
Enter 1 1 elements of data:5
Enter 1 2 elements of data:3
Enter 1 3 elements of data:3
Enter 2 0 elements of data:3
Enter 2 1 elements of data:5
Enter 2 2 elements of data:9
Enter 2 3 elements of data:7
  5  9  4  2
  4  5  3  3
  3  5  9  7
```

6.7. Массивы символов

В C++ не существует встроенного типа для текстовых данных. Один из способов реализации текстовых строк в C++ – в виде *массивов символов,* которые объявляются и реализуются точно так же, как и массивы других встроенных типов. Однако имеется важная особенность символьных массивов, позволяющая использовать их для работы с текстом, а именно, если последний символ массива является нулевым ‘\0’, то такой массив называется строковым литералом. Данный массив символов эквивалентен текстовой строке, заключенной в двойные кавычки.

```
const int n = 4;
char array_1 [n] = {'C', 'I', 'T', 'Y',}; /*Не
строка!!!*/
const int m=5;
char array_2[m]={'C','I','T','Y','\0'}; // Строка!!!
```

Последний оператор эквивалентен оператору:

```
char array_2 [m] = "CITY";
```

Кавычки не являются частью строки, а только ограничивают ее. Строковый литерал является массивом символов, размерность которого превышает на единицу количество символов, записанное между кавычками (последнюю размерность занимает нулевой символ). Таким образом, массив типа **char** можно инициализировать набором символов, взятых в двойные кавычки, кроме того, при выводе на экран строкового литерала достаточно указать имя массива, содержащего данный литерал:

Пример. Массив указателей, и сами массивы определены в динамической памяти.

```
#include<iostream>
#include<iomanip>
int main( )
{
const int n = 4;
char array_1[n]={'C','I','T','Y'};
const int m = 5;
char array_2[m]="city";
for (char *p=array_2;p-array_2<m;p++)// выводим поэлементно array_2
{std::cout << std::setw(3) << *p;   }
std::cout << std::endl;
std::cout<<array_2<<std::endl;// выводим строку, содержащуюся в array_2
for(char *p=array_2;p-array_2<n;p++) // копируем array_1 в array_2
{*p=*(array_1+(p-array_2));}
for (char *p=array_2;p-array_2<m;p++)  // выводим поэлементно array_2
{std::cout << std::setw(3) << *p;}
std::cout << std::endl;
std::cout<<array_2<<std::endl;// выводим строку, содержащуюся в array_2
return 0;}
```

Результат выполнения программы:

```
  c  i  t  y
city
  C  I  T  Y
CITY
```

Заметим, что аналогичный вывод на экран массива *array_1* возможен только поэлементно!

6.8. Массив строк

Двумерные массивы типа **char** определяются точно так же, как и двумерные массивы переменных других типов, например **int** или **double**. Однако двумерный массив типа **char** можно рассматривать как массив строк:

```
#include<iostream>
#include<iomanip>
int main( )
{ const int days = 7;
const int max = 10;
char week [days][max] =
{"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday"
};
for (int i = 0; i < days; i++)
{ for (int j = 0; j < max; j++)
  {  std::cout << std::setw(3) << *(*(week+i)+j); }
 std::cout << std::endl;
}
std::cout << std::endl;
for (int i = 0; i < days; i++)
{std::cout << week[i] <<std::endl;}
return 0;}
```

Результат выполнения программы:

```
  S  u  n  d  a  y
  M  o  n  d  a  y
  T  u  e  s  d  a  y
  W  e  d  n  e  s  d  a  y
  T  h  u  r  s  d  a  y
  F  r  i  d  a  y
  S  a  t  u  r  d  a  y

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Обратите внимание, что размерность массива на единицу больше числа букв в слове “Wednesday”, то есть девять букв и символ ‘\0’. Число столбцов не должно быть меньше длины максимального литерала включая ‘\0’.

Существенный недостаток такого представления текста как набора элементов в массиве – это невозможность манипуляции текстом как единым целым; вместо этого применяются операции с указателями.

7. СТРУКТУРЫ И ОБЪЕДИНЕНИЯ

Особый тип данных образуют *структуры* и *объединения*. В отличие от массивов они в общем случае состоят из компонентов различных типов. Компоненты структуры и объединения называются *полями*. Поля структуры размещаются в оперативной памяти одно за другим в той последовательности, в которой они перечислены в описании. Поля объединений разделяют одну область памяти, и, следовательно, накладываются друг на друга. Разница между структурами и объединениями ограничивается способом размещения их полей.

7.1. Определение структуры

Под ***структурами*** подразумевают группу переменных, объединенных общим именем. Удобство структуры состоит в том, что она позволяет группировать разнородные данные, что бывает полезно при работе с базами данных и записями. Общий синтаксис объявления структуры следующий*:*

struct *имя* {
  тип1 *поле1*;
  тип2 *поле2*;
  …
  типN *полеN*;
  } *список_переменных*;

Объявление структуры начинается с ключевого слова **struct**, после которого следует имя структуры и, в фигурных скобках, перечисляются поля структуры (типы и имена переменных, входящих в структуру). После фигурных скобок, заканчивающих непосредственно описание структуры, можно указать (а можно и не указывать) список переменных структуры. Само по себе описание структуры эту структуру не создает. *Описание структуры* – это всего лишь некий шаблон, который описывает данные, входящие в состав структуры и по которому впоследствии создаются переменные. Определение не создает никаких переменных, то есть не происходит ни выделения физической памяти, ни объявления переменной, в то время как определение обычной переменной предполагает выделение памяти под эту переменную.

Пример. Создать структуру для работы графическим объектом – точкой, которая описывается своими координатами *x, y* и *z*. Пусть координаты имеют тип **double**.

```
struct point
{
double x;
double y;
double z;
};
```

Данное определение структуры необходимо для того, чтобы создавать на его основе переменные типа *point*.

7.2. Определение структурной переменной

С точки зрения использования в программе имеет смысл говорить лишь о переменных структуры. Фактически *переменная структуры* – это объект, который имеет имя (имя переменной структуры) и поля, тип и названия которых определяются описанием структуры. Чтобы в программе создать переменную структуры, необходимо указать имя структуры, в соответствии с описанием которой создается переменная, и имя этой переменной. Другими словами, переменная структуры в программе создается точно так же, как и переменная любого базового типа, только вместо названия типа переменной указывается название структуры.

Пример. Создать две переменных, имеющих тип *point*.

```
point A,B;
```

Определение переменных *A* и *B* означает, что под эти переменные выделяется память. Под структурную переменную всегда отводится столько памяти, сколько достаточно для хранения всех ее полей. В нашем примере под каждую переменную необходимо выделить по восемь байт для трех полей типа **double**. Таким образом, структура занимает в памяти 24 байта.

7.3. Доступ к полям структуры

Когда структурная переменная определена, все данные записываются в поля структуры, доступ к которым осуществляется через так называемый точечный синтаксис в формате:

имя_структурной_переменной . имя_поля

Операция точки в соответствии с общепринятой терминологией называется *операцией доступа к полю структуры*.

Пример. Присвоить полям переменной *A*, имеющей тип *point*, значения 1.11, 2.22, 3.33.

```
A.x=1.11;
A.y=2.22;
A.z=3.33;
```

С полями структурной переменной можно обращаться точно так же, как и с обычными переменными. Например, вывод значения поля *x* переменной *A*:

```
std::cout << A.x << std::endl;
```

Структуры можно вкладывать друг в друга. Глубина вложения может быть произвольной. Представим вектор на плоскости, как пару точек.

Пример. Объявить структуру *my_vector*, полями которой являются структуры типа *point*. Объявить переменную *Vector* типа *my_vector*.

```
struct my_vector // Объявление структуры my_vector
{
point A1; // 1-е поле структуры my_vector типа point
point A2; // 2-е поле структуры my_vector типа point
};
my_vector Vector; // Объявление переменной Vector типа my_vector
```

Тогда, чтобы задать координаты начальной и конечной точки вектора, необходимо использовать операторы:

```
1  Vector.A1.x=1.11;
2  Vector.A1.y=2.22;
3  Vector.A1.z=3.33;
4  Vector.A2.x=2.34;
5  Vector.A2.y=3.22;
6  Vector.A2.z=1.14;
```

7.4. Инициализация структурных переменных

Структуры также можно инициализировать при объявлении.

Пример. Объявить и инициализировать переменные *A* и *B* типа *point*.

```
point A={1.11, 2.22, 3.33};
point B={2.34, 3.22, 1.14};
```

Пример. Объявить и инициализировать переменную *Vector* типа *my_vector*.

```
my_vector Vector ={{1.11, 2.22, 3.33},{2.34, 3.22, 1.14}};

//или

point A={1.11, 2.22, 3.33};
point B={2.34, 3.22, 1.14};
my_vector Vector ={A,B};
```

Также можно присвоить значение одной структурной переменной другой структурной переменной, например:

```
point A={1.11, 2.22, 3.33};
point B={2.34, 3.22, 1.14};
my_vector Vector1;
Vector1.A1=A; /* Полю A1 структурной переменной  Vector1
присваивается значение структурной переменной A,  т.е.
значение каждого поля переменной A присваивается
соответствующему полю A1 переменной Vector1   */
Vector1.A2=B; /* Полю A2 структурной переменной  Vector1
присваивается значение структурной переменной B */
my_vector Vector2; // Объявляется переменная Vector2 типа my_vector
Vector2 = Vector1; // Переменной Vector2 присваивается значение Vector1
```

7.5. Объединения

Под ***объединениями*** подразумевают область памяти, в которой одновременно хранится несколько различных переменных. Одно и то же объединение в одних операциях может участвовать как переменная типа **int**, а в других – как переменная типа **float** или **double**. Для всех членов такой переменной реализуется единая область памяти, которую они используют совместно. Размерность объединения определяется размерностью самого большого по памяти его элемента. Общий синтаксис объявления объедине-

ния подразумевает использование ключевого слова **union** и имеет следующий вид:

union *имя_объединения* {
тип_переменной1 *имя_переменной1*;
тип_переменной2 *имя_переменной2*;
тип_переменнойN *имя_переменнойN*;
} *список переменных*;

Как и в случае структуры, само по себе объявление объединения не приводит к созданию новых переменных. При объявлении переменной объединения в качестве ее типа указывают имя объединения. Например,

```
union uni{ //Объявление объединения с именем uni
int i;     // 1-е поле объединения uni
char ch;   // 2-е поле объединения uni
double d; // 3-е поле объединения uni
};
```

Как и в случае со структурой, описание объединения не распределяет память, а предоставляет шаблон для будущего объявления переменных. Переменные описанного типа *uni* будут занимать в памяти по 8 байтов (таков размер самого большого поля *d*). Переменная *perem* типа *uni* описывается следующим образом:

```
uni perem;
```

Данное объединение содержит три элемента: *ch, d, i*. Как и для структуры, программа может присвоить значение любому элементу, однако в отличие от структуры значение может быть присвоено только одному элементу в каждый момент времени. Предположим, программа присваивает значение элементу *ch*:

```
perem.ch = 'A';
```

Если программа присваивает значение элементу *d*, то значение, присвоенное элементу *ch*, теряется.

В отличие от структуры элементы объединения инициализировать нельзя. Ссылка на элементы объединения выполняется с помощью того же синтаксиса, что и ссылка на поля структуры: член объединения указывается через точку после имени экземпляра объединения.

Пример. Данный пример демонстрирует использование объединений.

```
#include <iostream>
using namespace std;
union tag
{   int intval;
    float floatval;
    char *charval;
};
int main()
{   tag  u;
    cout << " u.charval=" << u.charval << endl;
    cout << " u.floatval=" << u.floatval << endl;
    cout << " u.intval=" << u.intval << endl;
    u.charval="UIR"; cout << endl;
    cout << " u.charval=" << u.charval << endl;
    u.floatval=2.6; cout << endl;
    cout << " u.floatval=" << u.floatval << endl;
    return 0;}
```

Результат выполнения программы:

```
u.charval=â-¶[╔├1 δΘë├ìC☺ï¶à¿↔@
u.floatval=5.88701e-039
u.intval=4201110

u.charval=UIR

u.floatval=2.6
```

Обратите внимание, что после объявления поля объединения не пустые, а содержат произвольные значения.

8. ФУНКЦИИ

В С++ сложная задача может быть разделена на более простые и обозримые с помощью функций, после чего программу можно рассматривать в более укрупненном виде – на уровне взаимодействия функций. Использование функций ведет к упрощению ее структуры. Разделение программы на функции позволяет избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы.

Функция – это именованный программный код, выполняющий какое-либо законченное действие, который может многократно вызываться в программе. Функции реализуются отдельными программными блоками, могут иметь параметры и могут возвращать значения в качестве результата. Функция начинает выполнение в момент вызова. Любая функция должна быть *объявлена* и *определена*.

8.1. Объявление, определение и вызов функции

Объявление функции (прототип) задает ее имя, тип возвращаемого ею значения, тип и количество параметров. *Определение* функции содержит, кроме объявления, тело функции, представляющее собой последовательность операторов и описаний в фигурных скобках. Все С++ - функции имеют общий формат:

тип_возвращаемого_значения *имя_функции* **(тип** *имя_параметра1***, тип** *имя_параметра2***,…, тип** *имя_параметраN***)**
{*операторы функции***}**

В С++ все функции должны быть объявлены до их использования. Поэтому прототип располагается до функции **main**(). *В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать*. Чтобы вызвать функцию, в общем случае, достаточно указать ее имя с парой круглых скобок. После вызова управление программой переходит к функции. Выполнение функции продолжается до обнаружения закрывающей фигурной скобки. Когда функция завершается, управление передается инициатору ее вызова.

Пример. Создание и использование в программе функции *sumfunct*, вычисляющей произведение двух чисел.

```
#include <iostream>
using namespace std;
float  sumfunct(int a,  int b) // Объявление функции sumfunct
{                              // с параметрами a и b типа int
int y = b*a;
return  y; // Функция возвращает значение y
}
int main()
{ int rez;
rez = sumfunct (78,5);/* Переменной rez присваивается значение,
        возвращаемое функцией sumfunct, вызванной с аргументами 78 и 5  */
                      // Число 78 присваивается параметру a функции
                      // Число 5 присваивается параметру b функции
cout << "rez =" << rez;
return 0;}
```

Результат выполнения программы:

```
rez =390
```

Пример. Реализация функции *sumfunct*, не возвращающей значение.

```
#include <iostream>
using namespace std;
void  sumfunct(int a,  int b); // Прототип функции sumfunct
   /* В C++ функции, не возвращающие значений, объявляются с
использованием ключевого слова void  */
int main()
{ int rez;
sumfunct (78,5); // Вызывается функция sumfunct с аргументами 78 и 5
return 0;}
void  sumfunct(int a,  int b) // Объявление функции sumfunct
{                             // с параметрами a и b типа int
int y = b*a;           // Вычисление выражения
cout << "rez =" << y; // Вывод результата внутри функции
}
```

Результат выполнения программы:

```
rez =390
```

8.2. Использование аргументов

Функции при вызове можно передать одно или несколько значений. Значение, передаваемое функции, называется ***аргументом***. При создании функции, которая принимает один или несколько аргументов, необходимо объявить переменные, которые получат значения этих аргументов. Эти переменные называются ***параметрами функции***.

8.3. Использование инструкции return

Инструкция **return** имеет две формы применения: первая позволяет возвращать значение из функции, вторая – нет.

8.3.1. Версия инструкции return, которая не возвращает значение

Если тип значения, возвращаемого функцией, определяется ключевым словом **void** (то есть функция не возвращает значения вообще), то для выхода из функции достаточно использовать такую форму инструкции **return**:

return;

При обнаружении инструкции **return** управление программой немедленно передается инициатору ее вызова. Любой код в функции, расположенный за инструкцией **return**, игнорируется.

Пример. Функция *power*(), показанная в следующей программе, отображает результат возведения целочисленного значения в положительную целую степень. Если показатель степени окажется отрицательным, инструкция **return** немедленно завершит эту функцию еще до попытки вычислить результат.

```
#include <iostream>
void power(int base, int exp); // Прототип функции power
int main()
{int a =10; int b =2;
power(a, b); // Вызов функции power
power(a, -b) ; // Вызов функции power
return 0;
}
void power (int base, int exp)
{ if (exp<0) return; /* Если показатель степени отрицательный,
                то - выход из функции */
/* Если показатель степени положительный, то будут выполнены
                 следующие операторы  */
int y=1;
while(exp!=0)
{y*=base;// Возводим целое число в положительную степень
exp--;}
std::cout << "Rezultat = " << y;}
```

Результат выполнения этой программы:

```
Rezultat = 100
```

Если значение параметра *exp* отрицательно (как при втором вызове функции *power*), вся оставшаяся часть функции опускается.

Функция может содержать несколько инструкций **return**. В этом случае возврат из функции будет выполнен при обнаружении одной из них. Однако следует иметь в виду, что слишком большое количество инструкций **return** может ухудшить ясность алгоритма и ввести в заблуждение тех, кто будет в нем разбираться. Несколько инструкций **return** стоит использовать только в том случае, если они способствуют ясности функции.

8.3.2. Версия инструкции return, которая возвращает значение

Функция может возвращать значение инициатору своего вызова. Таким образом, возвращаемое функцией значение – это средство получения информации из функции. Для возврата функцией значения используется вторая форма инструкции **return**:

return *значение*;

Эту форму инструкции **return** нельзя использовать с **void**-функциями.

Функция, которая возвращает значение, должна определить тип этого значения. Тип возвращаемого значения должен быть совместимым с типом данных, используемых в инструкции **return.** В противном случае неминуема ошибка во время компиляции программы. Функция может возвращать данные любого допустимого в C++ типа, за исключением массива.

Пример. Перепишем уже известную функцию *power*. В этой версии функция *power* возвращает значение.

```
#include <iostream>
int power(int base, int exp); // Прототип функции power
int main()
{int a =10; int b =2; int answer;
answer = power(a, b); // Переменной answer присваивается
                      //значение, возвращаемое функцией
std::cout << "Answer is " <<  answer;
return 0;}
int power (int base, int exp) // Описание функции
{ if (exp<0)
{ return 0; /* Если показатель степени отрицательный,
               то - выход из функции */
}
/* Если показатель степени положительный, то будут выполнены
                следующие операторы  */
int y=1;
while(exp!=0)
{y*=base;// Возводим целое число в положительную степень
exp--;}
return y;}
```

Результат выполнения этой программы:

```
Answer is 100
```

8.4. Правила действия областей видимости

Правила действия областей видимости любого языка программирования – это правила, которые позволяют управлять доступом к объекту из различных частей программы. Другими словами, правила действия областей видимости определяют, какой код имеет доступ к той или иной переменной. Эти правила также определяют продолжительность «жизни» переменной. В C++ определены две основные области видимости: *локальные* и *глобальные*. В любой из них можно объявлять переменные.

8.4.1. Локальная область видимости

Локальная область видимости создается блоком кода {}. Каждый раз при создании нового блока создается новая область видимости. Переменная, объявленная внутри любого блока кода, называется локальной по отношению к этому блоку. Важно понимать, что локальные переменные существуют только во время выполнения программного блока, в котором они объявлены. Это означает, что локальная переменная создается при входе в «свой» блок и разрушается при выходе из него. А поскольку

локальная переменная разрушается при выходе из «своего» блока, ее значение теряется.

Самым распространенным программным блоком кода является функция. Код функции и ее данные – это ее «частная собственность», и к ней не может получить доступ ни одна инструкция из любой другой функции, за исключением инструкции ее вызова. Тело функции надежно скрыто от остальной части программы, и она не может оказать никакого влияния на другие части программы, равно, как и те на нее. Таким образом, содержимое одной функции совершенно независимо от содержимого другой. Другими словами, код и данные, определенные в одной функции, не могут взаимодействовать с кодом и данными, определенными в другой, поскольку две функции имеют различные области видимости. Так как каждая функция определяет собственную область видимости, переменные, объявленные в одной функции, не оказывают никакого влияния на переменные, объявленные в другой, причем даже в том случае, если эти переменные имеют одинаковые имена.

Пример. Демонстрация использования переменной *a* в функции *somefunct* и в функции ***main***.

```
#include <iostream>
using namespace  std;
void somefunct() ;
int main()   {
int a = 5; //  val локальна по отношению к main()
cout << "The value of a variable in the function main:" << a << endl ;
somefunct();
cout << "The value of a variable in the function main: " << a << endl  ;
return 0;
}
void somefunct()
{  int a = 10; // val локальна по отношению к somefunct
cout << "The value of a variable in the function somefunct:" << a << endl;
}
```

Вот результаты выполнения этой программы.

```
The value of a variable in the function main:5
The value of a variable in the function somefunct:10
The value of a variable in the function main: 5
```

Переменная *a*, объявленная в функции **main**, не имеет никакого отношения к одноименной переменной из функции

somefunct. Каждая переменная (в данном случае *a*) известна только блоку кода, в котором она объявлена. Несмотря на то, что при выполнении функции *somefunct* переменная *a* устанавливается равной числу 10, значение переменной *a* в функции **main** остается равным 5.

Поскольку локальные переменные создаются с каждым входом и разрушаются с каждым выходом из программного блока, в котором они объявлены, они не хранят своих значений между активизациями блоков. При вызове функции ее локальные переменные создаются, а при выходе из нее – разрушаются, то есть локальные переменные не сохраняют своих значений между вызовами функций. Если объявление локальной переменной включает инициализатор, то такая переменная инициализируется при каждом входе в соответствующий блок. Содержимое неинициализированной локальной переменной будет неизвестно до тех пор, пока ей не будет присвоено некоторое значение.

Локальные переменные можно объявлять внутри любого блока в любом месте – главное, чтобы это произошло до ее использования.

Пример. Демонстрация использования локальных переменных.

```
1  //Переменные могут быть локальными по отношению к блоку
2  #include <iostream>
3  int main() {
4  int x = 5; //Переменная x известна всему коду функции
5  if(x==5) {
6  int y = 50; // Переменная y локальна для if-блока
7  std::cout << "x + y = " << x + y << std::endl;
8  }
9  //  y = 100; // Ошибка!  Переменная y здесь неизвестна
10 return 0; }
```

Результаты выполнения программы:

```
x + y = 55
```

Пояснение к программе. Переменная *x* объявляется в начале области видимости функции **main** и доступна для всего последующего кода этой функции. В блоке **if**-инструкции объявляется переменная *y*. Поскольку рамками блока и определяется область видимости, переменная *y* видима только для кода внутри этого блока.

Поэтому строка *y=100;* (она находится за пределами этого блока) отмечена как комментарий. Если убрать символ комментария (//) в начале этой строки, компилятор отреагирует сообщением об ошибке, поскольку переменная *y* невидима вне этого блока. Внутри **if**-блока вполне можно использовать переменную *x*, поскольку код в рамках блока имеет доступ к переменным, объявленным во включающем блоке.

8.4.2. Сокрытие имен переменных

Если имя переменной, объявленной во внутреннем блоке, совпадает с именем переменной, объявленной во внешнем блоке, то «внутренняя» переменная *скрывает*, или переопределяет, «внешнюю» в пределах области видимости внутреннего блока.

Пример. Демонстрация скрытия переменных.

```
1  #include <iostream>
2  using namespace  std;
3  int main ()
4  {
5  int i, j;
6  i = 10;
7  j = 100;
8  if(j > 0) {
9  int i; // Эта переменная i отделена от внешней переменной i
10 i = j / 2;
11 cout<<"Internal variable i: "<<i<<endl; //Внутренняя переменная i
12 }
13 cout<<"External variable i: "<<i<<endl; //Внешняя переменная i
14 return 0; }
```

Вот как выглядят результаты выполнения этой программы:

```
Internal variable i: 50
External variable i: 10
```

Пояснение к программе. Здесь переменная *i*, объявленная внутри **if**-блока, скрывает внешнюю переменную. Изменения, которым подверглась внутренняя переменная *i*, не оказывают никакого влияния на внешнюю *i*. Более того, вне **if**-блока внутренняя переменная *i* больше не существует, и поэтому внешняя переменная *i* снова становится видимой.

8.4.3. Параметры функции

Параметры функции существуют в пределах области видимости функции. Таким образом, они локальны по отношению к

функции. Если не считать получения значений аргументов при вызове функции, то поведение параметров ничем не отличается от поведения любых других локальных переменных внутри функции.

8.4.4. Глобальная область видимости

Локальные переменные известны только в пределах функции, в которой они объявлены. Чтобы создать переменную, которую могли бы использовать сразу несколько функций, необходимо создать переменную в глобальной области видимости. *Глобальная область видимости* – это декларативная область, которая «заполняет» пространство вне всех функций. При объявлении переменной в глобальной области видимости создается *глобальная переменная.*

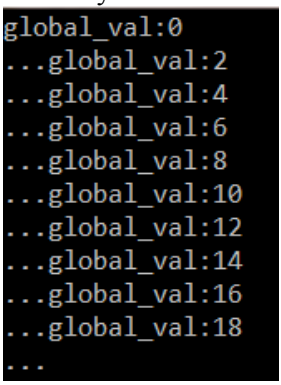
Глобальные переменные известны на протяжении всей программы, их можно использовать в любом месте кода, и они поддерживают свои значения во время выполнения всего кода программы.

Если в функции глобальная и локальная переменные имеют одинаковые имена, то при обращении к этому имени используется локальная переменная, не оказывая при этом никакого влияния на глобальную. Это и означает, что локальная переменная скрывает глобальную с таким же именем.

Пример. Переменная *global_val* объявлена вне всех функций (в данном случае до функции **main**), следовательно, она глобальная. Лучше объявлять глобальные переменные поближе к началу программы. Но формально они просто должны быть объявлены до их первого использования.

```
#include  <iostream>
using namespace  std;
void  func1();
void  func2();
int  global_val;     //Это глобальная переменная
int  main () {
int  i;   //Это локальная переменная
for (i=0; i<10; i++)  {
global_val= i*2; //Результат сохраняется в глобальной переменной global_val
func1() ; } //Вызов func1 в main
return  0; }
void  func1()
{cout<< "global_val:"<<global_val;   // Доступ к глобальной  переменной global_val
cout  << endl;   // Вывод  символа  новой  строки
func2();  } //Вызов func2 в func1
void func2()
{  int global_val; //  Это локальная переменная
// Здесь используется локальная переменная global_val
for(global_val=0; global_val<3; global_val++) cout << '.';  }
```

Результаты выполнения этой программы:



Пояснение к программе. Функция **main**() и функция *func1*() используют глобальную переменную *global_val*. Но в функции *func2*() объявляется локальная переменная *global_val*. Поэтому здесь при использовании имени *global_val* подразумевается именно локальная, а не глобальная переменная *global_val*.

8.5. Перегрузка функций

В С + + несколько функций могут иметь одинаковые имена, но при условии, что их параметры будут различными. Такую ситуацию называют *перегрузкой функций*, а функции, которые в ней задействованы, – *перегруженными*. Чтобы перегрузить функцию, достаточно объявить различные ее версии. Об остальном позаботится компилятор. При этом важно помнить, что тип и/или количество параметров каждой перегруженной функции должны быть уникальными, поскольку именно эти факторы позволят компилятору определить, какую версию перегруженной функции следует вызвать.

Таким образом, перегруженные функции должны отличаться типами и/или числом параметров. Несмотря на то, что перегруженные функции *могут* отличаться и типами возвращаемых значений, этого вида информации недостаточно для С++, чтобы во всех случаях компилятор мог решить, какую именно функцию нужно вызвать.

Пример. Демонстрация "трехкратной" перегрузки функции *f*.

```
#include <iostream>
using namespace  std;
// Для каждой версии перегруженной функции необходимо
//включить в программу отдельный прототип
void f(int i) ;       //Функция содержит один целочисленный параметр
void f(int i, int j); //Функция содержит два целочисленных параметра
void f(double j);     //Функция содержит один параметр типа double
int main() {
f(10);            //   вызов 1-го варианта функции  f
f(10,   20);    //  вызов 2-го варианта функции  f
f(12.23);      //  вызов 3-го варианта функции  f
return   0; }
// Первая реализация функции f
void f(int i) {
cout << "In the function f (int), i = " << i << endl; }
// Вторая реализация функции f
void f(int i, int j) {
cout << "In the function f(int, int), i = " << ", j = " << j << endl;}
// Третья реализация функции f
void f(double j) {
cout << "In the function f(double), j = " << j << endl;}
```

Результат выполнения программы:

```
In the function f (int), i = 10
In the function f(int, int), i = , j = 20
In the function f(double), j = 12.23
```

Принципиальная значимость перегрузки состоит в том, что она позволяет обращаться к связанным функциям посредством одного, общего для всех, имени. Еще одно достоинство перегрузки состоит в том, что она позволяет определить версии одной функции с небольшими различиями, которые зависят от типа обрабатываемых этой функцией данных.

8.6. Способы передачи аргументов в функции

В C++ существует два механизма передачи аргументов функциям: *по значению* и *через ссылку*. В первом случае значение аргумента копируется в параметр функции. Следовательно, изменения, внесенные в параметры функции, не влияют на аргументы, используемые при ее вызове. Во втором случае в параметр копируется адрес аргумента (а не его значение). В пределах вызываемой подпрограммы этот адрес используется для доступа к реальному аргументу, заданному при ее вызове. Это значит, что изме-

нения, внесенные в параметр, окажут воздействие на аргумент, используемый при вызове подпрограммы. Второй метод позволяет в качестве аргументов использовать *указатели* и *массивы*.

8.6.1. Передача аргумента по значению

По умолчанию для передачи аргументов в функцию в C++ используется *метод передачи по значению.*

Пример. Функция *swap* меняет местами значения своих параметров. Передаваемые аргументы остаются без изменения. Аргументы в функцию передаются по значению.

```
#include <iostream>
using namespace  std;
void swap (int x, int y);
int main()
{      int x = 3;  int y = 7;
       cout << "The initial values of x and y:" << endl;
       cout << "x= " << x << endl; cout << "y= " << y << endl;
       swap (x,y);
       cout << "The functions Main.  After swap: " << endl;
       cout << "x= " << x << endl; cout << "y= " << y << endl;
    return 0;
}
//Функция  swap (x,y) определена следующим образом
void swap (int x, int y)
{
int temp;
temp = x;
x = y;
y =temp;
cout << "The functions swap.  After swap: " << endl;
cout << "x= " << x << endl; cout << "y= " << y << endl;
}
```

Результат выполнения программы:

```
The initial values of x and y:
x= 3
y= 7
The functions swap.  After swap:
x= 7
y= 3
The functions Main.  After swap:
x= 3
y= 7
```

8.6.2. Передача аргумента по ссылке

При использовании метода передачи по ссылке функции будет передаваться адрес аргумента (то есть указатель на аргумент). Это позволит внутреннему коду функции изменять значение аргумента, которое хранится вне функции. Указатели передаются функциям подобно значениям любого другого типа. Безусловно, для этого необходимо объявить параметры функции с типом указателей.

Пример. Функция *swap* меняет местами значения своих параметров. Передаваемые аргументы также меняются местами, так как аргументы в функцию передаются по ссылке.

```
#include <iostream>
using namespace  std;
void swap (int *x,int *y);
int main()
{     int  i= 3;  int j = 7;
      cout << "The initial values of i and j:" << endl;
      cout << "x= " << i << endl; cout << "y= " <<j << endl;
      swap (&j, &i); // Вызываем swap с адресами переменных i и j
      cout << "The functions Main.  After swap: " << endl;
      cout << "x= " << i << endl; cout << "y= " << j << endl;
    return 0;
}
void swap(int *x, int *y)
{int temp;
temp = *x;//Временно  сохраняем  значение, расположенное по адресу x
*x = *y;//Помещаем значение, хранимое по адресу y, по адресу x
*y =temp;//Помещаем  значение, которое раньше
                        //хранилось по адресу x, по адресу y
}
```

Результат выполнения программы:

```
The initial values of i and j:
x= 3
y= 7
The functions Main.  After swap:
x= 7
y= 3
```

Функция *swap* объявляет два параметра-указателя (x и y). Поскольку функция *swap* ожидает получить два указателя, надо помнить, что функцию *swap* необходимо вызывать с *адресами* переменных, значения которых надо поменять. Корректный вызов этой функции продемонстрирован в следующей программе.

Пример. Реализация функции *swap* с использованием динамической памяти.

```
#include <iostream>
void swap (int *x,int *y);
int main( ) {
int* px = new int(3); int* py = new int(7);
    std::cout << "Main.  Before swap: " << std::endl;
std::cout << "*px= " << *px << std::endl;
std::cout << "*py= " << *py << std::endl;
swap (px, py);
    std::cout << "Main.  After swap: " << std::endl;
std::cout << "*px= " << *px << std::endl;
std::cout << "*py= " << *py << std::endl;
        delete py;   py=0;
        delete px;   px=0;
        return 0;}
void swap(int *x, int *y)
{int temp;temp = *x;*x = *y;*y =temp;}
```

Результат выполнения программы:

```
Main.  Before swap:
*px= 3
*py= 7
Main.  After swap:
*px= 7
*py= 3
```

Передавая указатель функции, необходимо понимать следующее. При выполнении некоторой операции в функции, которая использует указатель, эта операция фактически выполняется над переменной, адресуемой этим указателем. Это значит, что такая функция может изменить значение объекта, адресуемого ее параметром.

8.6.3. Ссылочные параметры

Ссылка – это неявный указатель. В C++ можно сориентировать компилятор на автоматическое использование вызова по ссылке для одного или нескольких параметров конкретной функции. Такая возможность реализуется с помощью *ссылочного параметра*. При его использовании функции автоматически передается адрес аргумента. При выполнении операций над ссылочным параметром обеспечивается его автоматическое разыменование, и

поэтому не нужно прибегать к операторам, используемым с указателями.

Ссылочный параметр объявляется с помощью символа "&", который должен предшествовать имени параметра в объявлении функции. Операции, выполняемые над ссылочным параметром, оказывают влияние на аргумент, используемый при вызове функции, а не на сам ссылочный параметр. Например,

```
void swap (int &x, int &y)
{
int temp;
temp = x;
x = y;
y = temp;}
```

Итак, подведем некоторые итоги. После создания ссылочный параметр автоматически ссылается (то есть неявно указывает) на аргумент, используемый при вызове функции. Более того, при вызове функции не нужно применять к аргументу оператор "&". Кроме того, в теле функции ссылочный параметр используется непосредственно, то есть без оператора "*". Все операции, включающие ссылочный параметр, автоматически выполняются над аргументом, используемым при вызове функции. Наконец, присваивая некоторое значение параметру-ссылке, вы в действительности присваиваете это значение переменной, на которую указывает эта ссылка. Поэтому, применяя ссылку в качестве параметра функции, при вызове функции вы в действительности используете переменную.

На применение ссылочных переменных налагается ряд следующих ограничений:

– нельзя ссылаться на ссылочную переменную;
– нельзя создавать массивы ссылок;
– нельзя создавать указатель на ссылку, то есть нельзя к ссылке применять оператор "&".

8.7. Передача функции массива

Если массив является аргументом функции, то при вызове такой функции ей передается **только адрес первого элемента**

массива, а не полная его копия. (В C++ имя массива без индекса представляет собой указатель на первый элемент этого массива.) Это означает, что объявление параметра должно иметь тип, совместимый с типом аргумента.

Существует три способа объявить параметр, который принимает указатель на массив.

Во-первых, параметр можно объявить как массив, тип и размер которого совпадает с типом и размером массива, используемого при вызове функции. Этот вариант объявления параметра-массива продемонстрирован в следующем примере.

```
#include<iostream>
using namespace  std;
void display (int num[10]); //Пототип функции display
int main() {
int t[10], i; //Объявление массива t и переменной i
for (i=0; i<10; ++i) t[i]=i*2-1; // Инициализация элементов массива в цикле
display(t);  // Вызов функции и передача в качестве аргумента массива t
return 0;  }
// Функция display выводит все элементы массива
void display(int num[10])// Параметр объявлен как массив заданного размера
{ int i;
for (i=0; i<10; i++) cout << num[i]<< ' ';
}
```

Результат выполнения программы:

```
-1 1 3 5 7 9 11 13 15 17
```

Второй способ объявления параметра-массива состоит в его представлении в виде безразмерного массива, как показано ниже.

```
void display(int num[])
// Параметр объявлен как безразмерный массив
{ int i;
for(i=0; i<10; i++)  cout << num[i] << ' '; }
```

Здесь параметр *num* объявляется как целочисленный массив неизвестного размера. Поскольку C++ не обеспечивает проверку нарушения границ массива, то реальный размер массива – нерелевантный фактор для подобного параметра (но, безусловно, не для программы в целом). Целочисленный массив при таком способе объявления также автоматически преобразуется C++ - компилятором в указатель на целочисленное значение.

Третий способ объявления параметра-массива. При передаче массива функции ее параметр можно объявить как указатель, как показано ниже.

```
void display (int *num) /* Параметр здесь
объявлен как указатель */
{ int i;
for (i=0; i<10; i++)  cout << num [i]<< ' '; }
```

Возможность такого объявления параметра объясняется тем, что любой указатель (подобно массиву) можно индексировать с помощью символов квадратных скобок ([ ]). При вызове функции *display* в качестве первого аргумента передается адрес массива значений, а в качестве второго – размер.

Таким образом, все три способа объявления параметра-массива приводятся к одинаковому результату, который можно выразить одним словом – *указатель*.

Примечание. Если массив используется в качестве аргумента функции, то функции передается адрес этого массива. Это означает, что код функции может потенциально изменить реальное содержимое массива, используемого при вызове функции.

Пример. Программа, в которой функция *cube* преобразует значение каждого элемента массива в куб этого значения. При вызове функции *cube* в качестве первого аргумента необходимо передать адрес массива значений, подлежащих преобразованию, а в качестве второго его размер.

```
1  #include<iostream>
2  using namespace  std;
3  void cube (int *n, int num);
4  int main ( )
5  {  int i, nums [10];
6  for(i=0; i<10; i++) nums[i] = i + 1;
7  for(i=0; i<10; i++) cout << nums[i] << ' ';
8  cube(nums, 10); //Передаем функции cube() адрес
9                  //массива nums
10 cout << endl;
11 for(i = 0; i<10; i++) cout << nums[i] << ' ' ;
12 return   0; }
13 void cube(int *n, int num)
14 {while (num){
15 *n = *n**n **n; // Это выражение изменяет значение элемента
16                                    //массива, адресуемого указателем n
17 num--;  n++;  }  }
```

Результат выполнения программы:

```
1 2 3 4 5 6 7 8 9 10
1 8 27 64 125 216 343 512 729 1000
```

Передача многомерных, и в частности двумерных, массивов функции осуществляется по следующему принципу: указываются все размеры, кроме первого.

Пример. Передача функции двумерного массива, которая выводит его на экран.

```
1  #include <iostream>
2  using namespace std;
3  void print(int massiv[][3],int size);
4  int main(){
5  int massiv[][3]={{1,2,3},{4,5,6}};
6  print(massiv,2);
7  return 0;
8  }
9  void print(int massiv[][3],int size)
10 {int i,j;
11 for(i=0;i<size;i++){
12    for(j=0;j<3;j++){
13       cout<<massiv[i][j]<<" ";}
14    cout<<endl;}
15 }
```

Результат выполнения программы:

```
1 2 3
4 5 6
```

В приложениях 1 и 2 представлены еще два примера, демонстрирующий передачу функциям одномерных и двумерных массивов.

8.8. Передача функциям строк

Поскольку строки в C++ – это обычные символьные массивы, которые завершаются нулевым символом, то при передаче функции строки реально передается только указатель (типа **char**) на начало этой строки.

Пример. Программа, в которой определяется функция *strInvertCase*, которая инвертирует регистр букв строки, то есть заменяет строчные символы прописными, а прописные – строчными.

```
#include <iostream>
#include <cstring>
#include <ctype.h>
void  strInvertCase(char *str);
int   main()
{ char  str[80];
  strcpy(str,   "This   Is  A  Test");
  strInvertCase(str);
  std::cout << str; //Отображаем модифицированную  строку
return 0;
}
void strInvertCase(char *str)
{while(*str) {
if(isupper(*str)) *str = tolower(*str);
else if(islower(*str)) *str = toupper(*str);
str++; }// Переход к следующей букве
}
```

Вот как выглядит результат выполнения этой программы:

```
tHIS   iS  a  tEST
```

8.9. Возвращение функциями указателей и ссылок

Функции могут возвращать указатели. Указатели возвращаются подобно значениям любых других типов данных и не создают при этом особых проблем. Чтобы вернуть указатель, функция должна объявить его тип в качестве типа возвращаемого значения. Вот как, например, объявляется тип возвращаемого значения для функции *f()*, которая должна возвращать указатель на целое число.

```
int  *f();
```

Если функция возвращает указатель, то значение, используемое в ее инструкции **return**, также должно быть указателем. (Как и для всех функций, **return**-значение должно быть совместимым с типом возвращаемого значения.)

Когда функция возвращает ссылку, она фактически возвращает указатель на возвращаемое значение. Функцию можно использовать с левой стороны присваивания.

Пример. Функция, возвращающая ссылку, возвращает значение.

```
1   #include<iostream>
2   using namespace  std;
3   int  &increase(int &var); //Прототип функции increase
4   int main()
5   {
6    int my_var =5;
7    increase(my_var);
8    cout << "my_var= " << my_var << endl;
9    increase(my_var)=10;
10   cout << "my_var= " << my_var << endl;
11  return 0;
12  }
13  int  &increase(int &var) // Определение функции increase
14  {var++;
15  return var;
16  }
```

Результат выполнения программы:

```
my_var= 6
my_var= 10
```

Нельзя возвращать ссылки на локальные переменные!

Приведенные ниже программные коды ошибочны.

```
int &increase(int
var)
{
var++;
return var;
} /* Ошибка
компиляции!! */
 / * В функцию
передается копия
переменной var. После
выхода из функции
локальная переменная
уничтожается, а
ссылка на нее
остается, что
является не
допустимым */
```

```
int &badexample(void)
{int var;
var++;
return var;
} //Ошибка компиляции!!


/* Здесь переменная var
также является
локальной */
```

Следующий пример корректен:

```
#include<iostream>
using namespace  std;
int  goodexample(void);
int main()
{
 int my_var =5;
 my_var = goodexample();
 cout << "my_var= " << my_var << endl;
return 0;
}
int  goodexample(void)
{int var=0;
var++;
return var;
} //Корректно!!
```

Результат выполнения программы:

```
my_var= 1
```

Однако, если присвоить функции значение переменной, то программа не скомпилируется:

```
int main()
{
int my_var = 5;
goodexample()=my_var; //Ошибка компиляции!!
cout << "my_var= " << my_var << endl;
return 0;
}
```

Сказанное выше относится и к указателям! Следующие примеры программ демонстрируют это:

```
#include<iostream>
using namespace  std;
int * increase(int *var);
int main()
{
 int my_var =5;
 int * increase(int *var);
 increase(&my_var);
 std::cout << "my_var= " << my_var << std::endl;
return 0;
}
int * increase(int *var)
{
(*var)++;
return var;
}
```

Результат:

```
my_var= 6
```

```
#include<iostream>
using namespace  std;
int * increase(int *var);
int main()
{
 int my_var =5;
 int * increase(int *var);
 *(increase(&my_var))=10;
 std::cout << "my_var= " << my_var << std::endl;
return 0;
}
int * increase(int *var)
{
(*var)++;
return var;
}
```

Результат:

```
my_var= 10
```

Нельзя возвращать указатели на локальные переменные!

Следующий код скомпилируется, но будет опасен при выполнении, поскольку возвращается указатель на переменную, которая после выхода из функции разрушается.

```
#include<iostream>
using namespace  std;
int *badexample(int val)
{ val++;
return &val;}
int main()
{ int my_var = 5;
 *(badexample(my_var)) = 10;
   cout << "my_var= " << my_var << endl;
   cout << "badexample= " <<
*(badexample(my_var)) << endl;
return 0;}
```

Результат выполнения программы:

```
my_var=10
badexample=6
```

9. ФУНКЦИИ И СОСТАВНЫЕ ТИПЫ ДАННЫХ

9.1. Функции и структуры

Разрешенными операциями над структурами являются копирование и присваивание структуры как целого, взятие ее адреса операцией &, а также обращение к ее элементам. Копирование и присваивание включают в себя также передачу аргументов в функции и возвращение значений из функции. Структуры нельзя сравнивать между собой, складывать. Автоматическую структуру (определенную в стеке), как было показано выше, можно инициализировать.

Рассмотрим вопросы работы со структурами на конкретных примерах – напишем ряд функций для манипулирования точками и векторами. Здесь возможно несколько подходов к решению: передавать отдельные компоненты структур, целые структуры, указатели и ссылки на них. В каждом подходе есть свои сильные и слабые стороны.

Пример. Определить функцию, вычисляющую квадрат расстояния между двумя точками.

```
#include<iostream>
using namespace  std;
struct point
{ double x; double y; double z;};
double square_distance(point A, point B);
int main()
{double d;
point X1={1.11, 2.22, 3.33};
point X2={2.34, 3.22, 1.14};
d=square_distance(X1,X2);
cout << d << endl;
return 0;
}
double square_distance(point A, point B)
{return ((B.x-A.x)*(B.x-A.x)+(B.y-A.y)*(B.y-A.y)+(B.z-A.z)*(B.z-A.z));}
```

Результат выполнения программы:

```
7.309
```

В данной функции аргументами являются структуры, а возвращаемым значением – обычный тип данных с плавающей точкой.

Пример. Определить функцию, возвращающую структуру: точку, которая находится между точками X1 и X2.

```
#include<iostream>
using namespace  std;
struct point  // Определение структуры
{ double x; double y; double z;};
point middle_point(point A, point B); // Прототип функции
int main() {
point middle; // Объявление переменной middle типа point
 point X1={1.11, 2.22, 3.33}; //Объявление и иниц-ия перем. X1 типа point
 point X2={2.34, 3.22, 1.14}; //Объявление и иниц-ия перем. X2 типа point
//Присваивание перем-й middle рез-та, возвращаемого функцией middle_point
middle=middle_point(X1,X2); //Вызов функции middle_point с арг-ми X1 и X2
cout<<"x= "<<middle.x<<"  y= "<<middle.y<<"  z= "<< middle.z<< endl;
return 0;
}
point middle_point(point A, point B) // Определение функции
{point temp; // Объявление переменной temp типа point
temp.x=(A.x+B.x)/2; //Присваивание знач-я полю x структурной перем-й temp
temp.y=(A.y+B.y)/2; //Присваивание знач-я полю y структурной перем-й temp
temp.z=(A.z+B.z)/2; //Присваивание знач-я полю z структурной перем-й temp
return temp; }//Функция возвращает структуру temp
```

Результат выполнения программы:

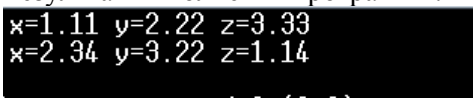
```
x= 1.725  y= 2.72  z= 2.235
```

Если в функцию необходимо передать большую структуру, то это лучше сделать, передав указатель на нее, а не копию всех ее данных. Поскольку в больших структурах количество полей иногда может измеряться десятками, то для выполнения копирования структурных переменных может понадобиться большое количество времени. Указатели на структуры обладают всеми свойствами указателей на обычные переменные. Следующий пример показывает, что *ptr_my_vect* является указателем на структуру типа *my_vector*.

```
#include<iostream>
using namespace  std;
struct point  // Объявление структуры point
{ double x; double y; double z;};
struct my_vector // Объявление структуры my_vector
{ point A1; // 1-е поле структуры my_vector типа point
  point A2; };// 2-е поле структуры my_vector типа point
int main() {
point A={1.11, 2.22, 3.33};//Объявление и иниц-ия перем-й A типа point
point B={2.34, 3.22, 1.14}; //Объявление и иниц-ия перем-й B типа point
my_vector a={A,B}; //Объявление и иниц-ия перем-й a типа my_vector
my_vector *ptr_my_vect =&a; /*Объявление указателя ptr_my_vect типа
my_vector и инициализация адресом переменной a */
cout<<" x="<<(*ptr_my_vect).A1.x<<" y="<<(*ptr_my_vect).A1.y<<" z="<<
(*ptr_my_vect).A1.z<<endl;
cout<<" x="<<(*ptr_my_vect).A2.x<<" y="<<(*ptr_my_vect).A2.y<<" z="<<
(*ptr_my_vect).A2.z<<endl;
return 0; }
```

🔔 Скобки при обращении к полям необходимы, поскольку приоритет операции по разыменовыванию указателя ниже, чем обращения к полям структуры.

Результат выполнения программы:



Указатели на структуры используются так часто, что для них введено дополнительное обозначение: обратиться по указателю к полям структуры можно посредством операции, состоящей из знака «минус» и «больше», например:

```
cout << " x=" << ptr_my_vect->A1.x << " y=" << ptr_my_vect->A1.y << " z=" << ptr_my_vect->A1.z << endl;
cout << " x=" << ptr_my_vect->A2.x << " y=" << ptr_my_vect->A2.y << " z=" << ptr_my_vect->A2.z << endl;
```

Операции обращения к структурам (точка и ->) наряду с круглыми скобками для вызовов функций и квадратными скобками для индексов массивов находятся на самой вершине иерархии приоритетов, а потому применяются к своим аргументам всегда в первую очередь.

9.2. Указатель на функцию

В языке C++ функция сама по себе не является переменной. Однако можно определить указатели на функции, которые разрешено присваивать, хранить в массивах, назначать полями структур, передавать в функции и возвращать из функций.

Указатель на функцию определяется следующим образом:

тип_возврата (* *имя_указателя*)(*список типов параметров*);

Например, объявление функций:

```
float sum(float* s1, float* s2, int i);
float cmp(float* s1, float* s2, int i);
float div(float* s1, float* s2, int i);
```

и объявление указателя на функции:

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

Указателю можно присвоить адрес функции двумя способами, например:

```
#include<iostream>
using namespace  std;
float sum(float* s1, float* s2, int i);
float mult(float* s1, float* s2, int i);
float (*ptrfunct)(float*, float*, int);
int main()
{
float *s1= new float(4.6);
float *s2= new float(2.3);
ptrfunct=sum; //операция присвоения указателю адреса  функции
cout << ptrfunct(s1,s2,2) << endl;
ptrfunct=&mult; //также является операцией присвоения указателю адреса функции
cout << ptrfunct(s1,s2,2) << endl;
return 0;
}
float sum(float* s1, float* s2, int i)
{return (*s1+*s2)*i;}
float mult(float* s1, float* s2, int i)
{return i*(*s1)*(*s2);}
```

Результат выполнения программы:

```
13.8
21.16
```

🔔 Тип возвращаемого значения и типы аргументов указателя на функцию должны соответствовать типу возвращаемого значения и типу аргументов самой функции.

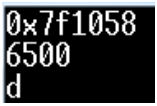
9.3. Указатели на тип void

Любой указатель на объект можно привести к типу **void*** без потери информации. Если результат подвергнуть обратному преобразованию, получится исходный указатель. В отличие от преобразований «указатель в указатель», которые требуют явного приведения типов, указатели типа **void*** можно использовать в операциях присваивания и сравнения. В других сочетаниях указателей стандарт требует явного приведения типов. Рассмотрим пример использования явного приведения указателей.

Пример. Преобразование указателя типа **int** в указатель типа **char**.

```
#include<iostream>
using namespace  std;
int main()
{
    int *pa = new int(6500); // создаем переменную в динамической памяти
    // и инициализируем ее значением 6500
    cout << pa << endl;
    cout << *pa << endl;
    char *pb = 0; // объявляем указатель на тип char
    //pb = static_cast<char*>(pa);
    pb = reinterpret_cast<char*>(pa);
    cout << *pb << endl;
    delete pa;
    return 0;
}
```

Результат:



Если число 6500 записать в двоичной системе исчисления, то оно будет выглядеть так: 1100101100100. Переменная типа **int** занимает в памяти четыре байта, и, с учетом того, что в архитектуре Intel переменные в памяти хранятся в обратном порядке, данное число запишется следующим образом:

01100100 00011001 00000000 00000000

Указатель *pa* указывает на ячейку памяти, в которой записана последовательность 01100100. После приведения указателя к типу **char*** и указатель по-прежнему будет указывать на данную ячейку. Поэтому, когда мы присваиваем адрес этой ячейки указателю *pb* и разыменовываем его, мы получаем значение 01100100. В десятичной записи это число 100. Если мы откроем таблицу символов ASCII, то увидим, что это число является кодом символа «d».

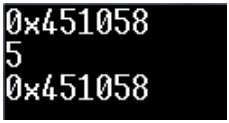
Указатель на тип **void*** нельзя разыменовывать, нельзя прибавлять к нему целое число, а также нельзя выделить для него динамическую память.

Однако, можно получить адрес ячейки, на которую он указывает, а также присвоить ему указатель любого другого типа без явного преобразования:

Пример. Присвоение указателю на тип **void** значения другого указателя.

```
1   #include<iostream>
2   int main()
3   {
4   int *pa = new int(5);
5   void *pv=pa;
6   std::cout << pa << std::endl;
7   std::cout << *pa << std::endl;
8   std::cout << pv << std::endl;
9   return 0;
10  }
```

Результат:



Указателю любого типа можно присвоить указатель на тип **void**, однако при этом требуется явное преобразование типов:

```
#include<iostream>
int main()
{
int  *pa = new int(5);
void *pv=pa;
pa=0;
int *pb = static_cast<int*>(pv);
std::cout << *pb;
delete pb;
return 0;
}
```

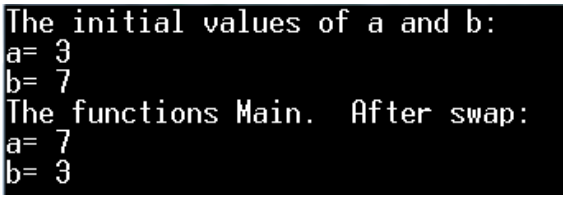
Результат:

9.4. Указатели на функции, аргументами или возвращаемыми значениями которых является указатель на void

При помощи указателей на тип **void** можно определить функцию, которая работает с произвольными типами данных. Рассмотрим пример функции *swap*, которая в отличие от аналогичной функции, определенной в предыдущих разделах, переставляет два значения произвольного типа:

```
#include <iostream>
using namespace  std;
void swap(void *v1,void *v2, unsigned int sz);
int main()
{     int  a= 3;  int b = 7;
      cout << "The initial values of a and b:" << endl;
      cout << "a= " << a << endl; cout << "b= " << b << endl;
      swap (&a, &b ,sizeof(int)); // Вызываем swap с адресами переменных a и b
      cout << "The functions Main.  After swap: " << endl;
      cout << "a= " << a << endl; cout << "b= " << b << endl;
    return 0;
}
void swap(void *v1,void *v2, unsigned int sz)
{
  char *a1 = static_cast<char*>(v1);
  char *a2 = static_cast<char*>(v2);
  for (int i=0; i<sz; i++)
  {
  char temp = *(a1+i);
  *(a1+i) = *(a2+i);
  *(a2+i) = temp;
  }
}
```

Результат выполнения программы:



В данную функцию передается массив элементов произвольного типа через указатель на **void**. Кроме того, передается размер типа передаваемого значения через переменную *sz*. В самой функции происходит преобразование указателя на тип **void** к указателю на тип **char**, поскольку переменная типа **char** занимает в памяти один байт. Далее в цикле побайтно копируется содержимое ячеек, в которых записаны значения *i* и *j* элементов массива.

```
/* Выполнение различных действий над одномерным массивом с
использованием указателей */
#include<iostream>
#include<conio.h>
#include<iomanip>
using namespace std;
//Прототипы функций
void copy_mass(float *a, float *b, int n);
void print(float *S, int n);
void Enter_mass(float *S, int n);
void copy_mass_del(float *a, float *b, int n, int cut);
void copy_mass_add(float *a, float *b, int n, int add, float element);
void delete_element(float **a, int *n, int cut);
void add_element(float **a, int *n, int add, float element);
int main()   // Начало основной программы
{     int N; //Объявление переменной N
      cout<<"Enter N:"; //Вывод сообщения
      cin>> N;  /* Ввод с клавиатуры размерности массива в переменную
N */
float *arr=new float [N];/*Определение массива arr размерности N
в динамической памяти */
cout<<"arr= "<<arr<<endl;/*Вывод адреса нулевого элемента массива */
cout<<endl; //Вывод пустой строки
      Enter_mass(arr,N);//Вызов функции для ввода элементов массива
      print(arr,N);//Вызов функции для вывода массива на экран
cout<<endl;//Вывод пустой строки
int m; //Объявление переменной m
      cout<<"Enter number element for delete:"; //Вывод сообщения
      cin>> m;//Ввод с клавиатуры значения в переменную m
      m=m-1; //Уменьшение переменной на единицу
      delete_element(&arr,&N,m);/*Вызов функции для удаления m-го
элемента массива */
cout<<endl;//Вывод пустой строки
      print(arr,N);/*Вызов функции для вывода на экран результирующего
массива */
cout<<endl;//Вывод пустой строки
// Вывод адреса нулевого элемента результирующего массива
      cout<<"arr= "<<arr<<endl;
      cout<<endl;//Вывод пустой строки
```

```
      cout<<"N= "<< N <<endl;// Вывод значения переменной N
/*Вызов функции для добавления 4-го элемента, имеющего значение 5.5
в массив */
      add_element(&arr,&N,3,5.5);
      cout<<endl;//Вывод пустой строки
      print(arr,N);/*Вызов функции для вывода на экран результирующего
массива */
      cout<<endl;//Вывод пустой строки
      /* Вывод адреса нулевого элемента результирующего массива */
      cout<<"arr= "<<arr<<endl;
      cout<<"N= "<< N <<endl;// Вывод N
      delete [] arr; // Удаление массива arr из динамической памяти
      arr=0; //Обнуление значения указателя
      _getch();
      return 0;   }
//Далее следуют определения функций
void copy_mass(float *a, float *b, int n) //Определение функции
{         // для копирования массива b в массив a
      for (int i=0; i<n; i++) {*(a+i)=*(b+i);}
}
void print (float *S, int n) //Определение функции для
{                    //вывода элементов массива
for (int i=0; i<n; i++) {cout<<setw(5) << *(S+i);       }
}
void Enter_mass(float *S, int n) //Определение функции
{     //для ввода элементов массива
      for (int i=0; i<n; i++)
      {     cout<<"Enter "<<i<<" element:";
            cin>> *(S+i); // Ввод i-го элемента массива
      }
}
/*Функция copy_mass_del копирует массив b в массив a, размерность
которого на единицу меньше, причем, элемент с индексом cut в массив a
не копируется */
void copy_mass_del(float *a, float *b, int n, int cut)
{
for (int i=0;i<n-1;i++)
      {
if (i<cut){*(a+i)=*(b+i);//Копирование элемента массива
            }
      else{*(a+i)=*(b+i+1);}
```

```
}
}
/*Функция copy_mass_add копирует массив b в массив a, размерность
которого на единицу больше, причем, в массив a добавляется элемент со
значением element, расположение которого указывается индексом add */
void copy_mass_add(float *a, float *b, int n, int add, float element)
{
for (int i=0;i<n+1;i++)
      {
if (i<add)
      {
        *(a+i)=*(b+i);
       }
      else if (i==add)
            {  *(a+i)=element;}
      else { *(a+i)=*(b+i-1); }
}
}
/* Функция delete_element удаляет из массива *a размерности *n элемент
с индексом cut */
void delete_element(float **a, int *n, int cut)
{
/* определение временного массива с размерностью на единицу меньше
размерности исходного массива */
float *temp= new float[*n-1];
/* копирование  массива *a в массив temp; элемент с индексом cut не
копируется */
copy_mass_del(temp,*a,*n,cut);
cout<<"*a= "<< *a<<endl;
delete [] *a; // удаление массива  *a  из памяти
(*n)--; // переменная  n  уменьшается на единицу
*a=new float[*n];/* указателю *a присваивается новое место в
динамической памяти */
cout<<"*a= "<< *a<<endl;
copy_mass(*a,temp,*n);/* копирование содержимого массива temp в
новый массив *a */
delete [] temp; // удаление временного массива из памяти
temp=0; // сброс указателя на ноль
}
/* Функция add_element добавляет в массив *a элемент element; индекс
элемента – add */
```

```
void add_element(float **a, int *n, int add, float element)
{
/* определение временного массива с размерностью
на единицу больше размерности исходного массива*/
      float *temp=new float[*n+1];
/* копирование  массива *a в массив temp; добавление элемента element
в массив temp; индекс добавленного элемента –add  */
      copy_mass_add(temp,*a,*n,add,element);
cout<<"*a= "<< *a<<endl;
delete [] *a;// удаление массива *a из памяти
(*n)++;// переменная n увеличивается на единицу
      *a=new float[*n];/* указателю *a присваивается новое место в
динамической памяти */
cout<<"*a= "<< *a<<endl;
copy_mass(*a,temp,*n);/* копирование содержимого массива temp в
новый массив *a  */
delete [] temp;// удаление временного массива из памяти
      temp=0;}// сброс указателя на ноль
```

Результат выполнения программы:

```
Enter N:5
arr= 0x321020

Enter 0 element:1
Enter 1 element:2
Enter 2 element:3
Enter 3 element:4
Enter 4 element:5
    1    2    3    4    5
Enter number element for delete:2
*a= 0x321020
*a= 0x321020

    1    3    4    5
arr= 0x321020

N= 4
*a= 0x321020
*a= 0x321020

    1    3    4  5.5    5
arr= 0x321020
N= 5
```

```
//Сортировка элементов каждой строки двумерного массива
#include<iostream>
#include<iomanip>
int get_minindex (double * array1, int left, int right); /* поиск индекса минимального элемента в подмассиве массива от элемента под номером left до элемента под номером right */
void swap (double * array1, int pos1, int pos2); /* перестановка элементов массива в позициях pos1 и pos2 */
void selection_sort (double * array1, int * size); /* сортировка элементов массива */
void print(double **SSS, int *r, int *c); // вывод на экран массива

int main()
{
	int * row = new int;
	std::cout <<"Enter the row:"<<std::endl;
	std::cin >> *row;
	int *coloumn = new int;
	std::cout <<"Enter the coloumn:"<<std::endl;
	std::cin >> *coloumn;
	double ** arr = new double * [*row]; /* объявление указателя на массив указателей в динамич. памяти */

	for (int i = 0; i < *row; i++) /* каждый указатель из заданного массива выделяет место в дин. памяти под одномерный массив */
	{
		*(arr+i) = new double[*coloumn];
		for (int j =0; j < *coloumn; j++) /* инициализация массивов */
		{
		std::cout <<"Enter "<< i <<" "<< j <<" element of arr"<<std::endl;
		std::cin >> *(*(arr+i)+j);
		}
	}
	print(arr, row, coloumn); // вывод массива
	for (int i = 0; i < *row ; i++)
	{
	selection_sort (*(arr+i), coloumn); // сортировка строк массива
	}
	std::cout << std::endl;
	print(arr, row, coloumn); // вывод массива
```

```
    for (int i = 0; i< *row; i++) /* удаление одномерных массивов из
динамической памяти */
    {
        delete [] *(arr+i);
    }
delete [] arr; //удаление из дин. памяти массива указателей
arr = 0;
    delete coloumn;
coloumn = 0;
    delete row;
row = 0;
    return 0;
}
// Функция ищет минимальный элемент и возвращает его индекс
int get_minindex (double * array1, int left, int right)
{
    int min_index = left;
    for (int i = left+1; i <= right; i++)
    {
    if (*(array1+i) < *(array1 + min_index))
     {
      min_index = i;
     }
    }
    return min_index;
}

/* Функция переставляет элементы массива в позициях pos1 и pos2 */
void swap (double * array1, int pos1, int pos2)
{
    double temp;
    temp = *(array1+pos1);
    *(array1+pos1) = *(array1+pos2);
    *(array1+pos2) = temp;
}
/* Функция сортирует элементы массива */
void selection_sort (double * array1, int * size)
{
    int min_index;
    for (int i = 0; i < * size; i++)
    {
    min_index = get_minindex (array1, i, *size - 1);
```

```
        if (min_index != i)
        {
        swap (array1, i, min_index);
        }
        }
}
/* Функция выводит массив на экран */
void print(double **SSS, int *r, int *c)
{
        for (int i =0; i < *r; i++)
        {
            for (int j =0; j < *c; j++)
            {
                std::cout << std::setw(6) << *(*(SSS+i)+j);
            }
            std::cout<<std::endl;
        }
}
```

Результат выполнения программы:

```
Enter the row: 4
Enter the coloumn: 5
Enter 00 element of arr 2
Enter 01 element of arr 6
Enter 02 element of arr 5
Enter 03 element of arr 6
Enter 04 element of arr 7
Enter 10 element of arr 3
Enter 11 element of arr 2
Enter 12 element of arr 1
Enter 13 element of arr 8
Enter 14 element of arr 5
Enter 20 element of arr 4
Enter 21 element of arr 2
Enter 22 element of arr 9
Enter 23 element of arr 7
Enter 24 element of arr 3
Enter 30 element of arr 1
Enter 31 element of arr 5
Enter 32 element of arr 0
Enter 33 element of arr 3
Enter 34 element of arr 6
     2     6     5     6     7
     3     2     1     8     5
     4     2     9     7     3
     1     5     0     3     6

     2     5     6     6     7
     1     2     3     5     8
     2     3     4     7     9
     0     1     3     5     6
```

ЛИТЕРАТУРА

1. Шилдт, Г. C++: базовый курс : пер. с англ. / Г. Шилдт. – 3-е издание. – М. : Издательский дом «Вильямс», 2010. – 624 с.

2. Шилдт, Г. Полный справочник по C++ : пер. с англ. / Г. Шилдт. – 4-е издание. – М. : Издательский дом «Вильямс», 2008. – 800 с.

3. Павловская, Т. А. C/ C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2009. – 461 с.

4. Шилдт, Г. C++: для начинающих : пер. с англ. / Г. Шилдт. – М. : ЭКОМ Паблишерз, 2007. – 640 с.

5. Павловская, Т. А. C/ C++. Структурное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2007. – 239 с.

6. Шилдт, Г. C++: руководство для начинающих : пер. с англ. / Г. Шилдт. – 2-е издание. – М. : Издательский дом «Вильямс», 2005. – 672 с.

7. Либерти, Д. Освой самостоятельно C++ за 21 день / Д. Либерти, Б. Джонс. – М. : Издательский дом «Вильямс», 2007. – 784 с.

8. Либерти, Д. Освой самостоятельно C++ за 24 часа / Д. Либерти, Д. Хорват. – М. : Издательский дом «Вильямс», 2007. – 448 с.

9. Васильев, А. Н. Самоучитель C++ с примерами и задачами / А. Н. Васильев. – СПб. : Наука и Техника, 2010. – 480 с.

10. Кузнецов, М. В. C++. Мастер-класс в задачах и примерах / М. В. Кузнецов, И. В. Симдянов. – СПб. : БХВ-Петербург, 2007. – 480 с.

11. Савич, У. Программирование на C++ / У. Савич. – 4-е изд. – СПб. : Питер ; Киев : BHV, 2004. – 781 с.

12. Прата, С. Язык программирования C++. Лекции и упражнения / С. Прата. – М. : ДиаСофтЮП, 2005. – 1104 с.

13. Страуструп, Б. Язык программирования C++. Специальное издание / Б. Страуструп ; пер. с англ. – М. : «Бином-Пресс», 2008. – 1104 с.

14. Дейтел, Х. М. Как программировать на C++ / Х. М. Дейтел, П. Дж. Дейтел. – Бином-Пресс, 2007. – 1456 с.

15. Культин, Н. Microsoft Visual C++ в задачах и примерах / Н. Культин. – Киев : BHV, 2010. – 274 с.

16. Хортон, А. Visual C++ 2008. Базовый курс / А. Хортон. – М. : Издательский дом «Вильямс», 2009. – 1280 с.

17. Оверленд, Б. C++ без страха : [учеб. пособие : пер. с англ.] / Брайан Оверленд. – М. : Триумф, 2005. – 432 с.

18. Лафоре, Р. Объектно-ориентированное программирование в C++. Классика ComputerScience. – 4-е изд. – СПб. : Питер, 2008. – 928 с.

19. Лоудон, К. C++. Карманный справочник / К. Лоудон. – СПб. : Питер, 2004. – 224 с.

20. Липпман, С. Язык программирования C++. Вводный курс / С. Липпман, Барбара Му, Жози Лажойе. – М. : Издательский дом «Вильямс», 2007. – 896 с.

21. Пахомов, Б. C/C++ и MS Visual C++ 2008 для начинающих / Б. Пахомов. – Киев : BHV, 2008. – 624 с.

22. Крупник, А. Б. Изучаем C++ / А. Б. Крупник – СПб. : Питер, 2004. – 251 с.

Учебно-методическое пособие

ЯЗЫК ПРОГРАММИРОВАНИЯ С++

Структурное программирование

Алейников Дмитрий Вячеславович
Холодова Елена Петровна

Компьютерная верстка М.И. Александровой
Дизайн обложки Т. В. Пешиной

Подписано в печать 31.10.2014. Формат 60x84 1/16. Гарнитура Таймс.
Цифровая печать. Усл. печ. л. 6,74. Уч.-изд. л. 4,12. Тираж 50 экз. Заказ 543.

Полиграфическое исполнение:
государственное учреждение «Национальная библиотека Беларуси».

Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий 1/398 от 02.07.2014.

Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий 2/157 от 02.07.2014.

Пр. Независимости, 116, 220114, Минск.
Тел. (+375 17) 293 27 68. Факс (+375 17) 266 37 23. E-mail: edit@nlb.by.