Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)
Кафедра промышленной электроники (ПрЭ)

Михальченко С. Г.

Информационные технологии. Часть 1. Программирование на C++.

Руководство по организации самостоятельной работы

ТОМСК 2016

У программистов есть свой собственный
покровитель - святой Исидор.
К сожалению, больше ничего святого
у них нет...

Михальченко Сергей Геннадьевич

Информационные технологии. Часть 1. Программирование на C++: Руководство по организации самостоятельной работы студентов. / С. Г. Михальченко; Томский государственный университет систем управления и радиоэлектроники, Кафедра промышленной электроники – Томск: ТУСУР, 2016. – 162 с. : ил., табл. – Библиогр.: с. 162.

Руководство по организации самостоятельной работы студентов предназначено для студентов направления «электроника и наноэлектроника».

Настоящее руководство имеет целью получение профессиональных компетенций в области информационных технологий, курс базируется на изучении языка C++ и применении полученных знаний в различных видах деятельности (инженерной, научно–исследовательской, управленческой, и др.).

Руководство может быть использовано для проведения практических занятий и лабораторных работ как аудиторно, так и в режиме самоподготовки.

Наличие вариантов индивидуальных заданий позволяет использовать настоящее руководство для проведения контрольных работ и итогового тестирования.

Рекомендуется для организации самостотятельной работы студентов.

Для освоения дисциплины *Информационные технологии* достаточно знаний, полученных студентом в школьном курсе информатики.

Содержание

```
- Назовите Ваш родной язык.
- Как это - родной язык?
- Ну, который Вы с детства знаете...
- Basic.
- Да нет, настоящий.
- А! Настоящий! - Тогда Си!
```

1. Основы программирования на C++

Язык программирования служит двум связанным между собой целям: он дает программисту аппарат для задания действий, которые должны быть выполнены, и формирует концепции, которыми пользуется программист, размышляя о том, что делать. Первой цели идеально отвечает язык, который должен быть настолько «близок к машине», что всеми основными машинными аспектами можно легко и просто оперировать достаточно очевидным для программиста образом. С таким умыслом первоначально задумывался *C*.

Второй цели идеально отвечает язык, который настолько «близок к решаемой задаче», чтобы концепции ее решения можно было выражать прямо и коротко. Именно для этого разрабатывались средства, добавленные к *C* для создания *C++*. [*1, 2*]

1.1. Процесс создания программного кода

Центральный процессор компьютера способен выполнять достаточно небольшой набор команд, представленных в виде последовательностей двоичных цифр, называемый *машинным кодом*. Но писать программу в машинных кодах для человека весьма не просто. В программировании применяется концепция, при которой программист пишет программу на *языке высокого уровня*, затем осуществляется перевод (*трансляция*) нашего (*исходного*) кода в машинный код. Специализированная программа – *компилятор*, производит трансляцию и генерирует *исполняемый файл*, содержащий весь необходимый машинный код, требующийся компьютеру для выполнения задания.

В самом общем, приблизительном смысле процесс компиляции состоит из нескольких стадий.

1 этап. Препроцессирование

На данном этапе работа идёт только с текстовыми файлами (**.c, *.cpp*). Здесь препроцессор вставляет в наш исходный текстовый файл и все заголовочные файлы (**.h, *.hpp*) подключаемых при помощи директивы (*#include*) библиотек.

Листинг 1

```
// макроопределения
#define CONST_A 10
// макроопределения с условием
#if CONST_A > 3
#define CONST_B 20
#else
#define CONST_B 0
#endif
// основная программа
void main ()
{     int a= CONST_A;
      int b= CONST_B;
      int c;
      c= a+b; // результат
}
```

Кроме того, препроцессор заменяет все константы (*#define*) их значениями, осуществляет условную обработку макроопределение исходного файла (*#ifdef*, *#endif* и

др.) и уничтожает комментарии.

Например, текст, приведенный на листинге (*Листинг 1*) будет *препроцессирован* компилятором к следующему виду (см. *Листинг 2*):

Листинг 2

```
void main ()
{      int a= 10;
       int b= 20;
       int c= a+b;
}
```

Можно видеть, что препроцессор заменил макроопределения *CONST_A* и *CONST_B* их значениями и удалил комментарии. В программе остались только переменные **a, b** и **c** целого типа (*int*). *Переменные* (именованные ячейки памяти определенного типа) используются в языках высокого уровня исключительно для удобства программиста, ассемблерные программы и тем более, машинные коды, в переменных не нуждаются. Подробно об использовании переменных и их типах будет рассказано ниже (см. *п.п. 1.4, 1.5*).

2 этап. Трансляция

Полученный после препроцессирования единый текстовый файл программы передается *транслятору*. Транслятор – это часть отладочной среды, проверяющая текст на отсутствие синтаксических ошибок, преобразующая конструкции языка С в конструкции ассемблера, выполняющая оптимизацию программного кода и генерирующая при необходимости отладочную информацию.

На выходе транслятора получается файл на языке *ассемблера* (**.asm*), содержащий откомпилированный текст – прообраз машинного кода.

В ассемблерном тексте содержатся символьные имена *меток* – переходов к внутренним и внешним подпрограммам. У каждой метки есть так называемая *область видимости: локальная или глобальная.* Локальные метки видны только внутри данного модуля, глобальные метки видны из других модулей. Например, метка *_main* (та *входная точка*, с которой начнется выполнение программы) является глобальной, т.к. она должна быть видна извне.

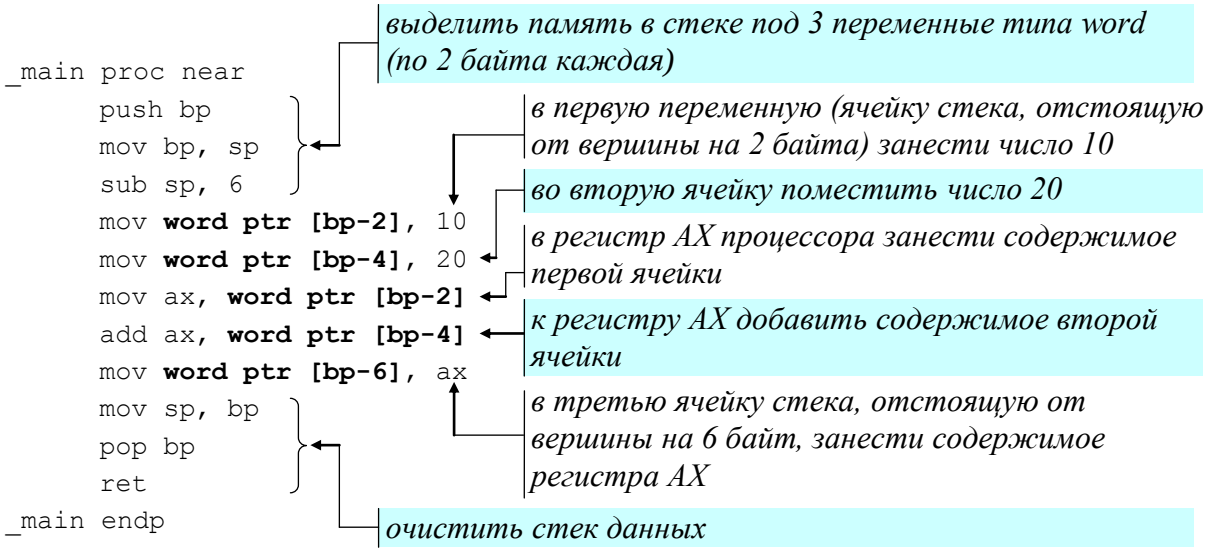

Рис. 1. Приблизительный ассемблерный код программы (Листинг 2)

На *Рис. 1* показан ассемблерный код рассматриваемой нами программы. Пояснения со стрелками указывают, как команды ассемблера заменяют команды языка *C++*.

Как было сказано, ассемблер не нуждается в переменных. Для хранения данных будет использован *стек* оперативной памяти (ОЗУ), в этом стеке вместо переменных *a*, *b* и *c* будут использованы ячейки памяти размером 2 байта (*word*) не имеющие имен. Но как же к ним обращаться? – По смещению (сдвигу) относительно вершины стека. Переменной *a* будет соответствовать ячейка, сдвинутая относительно вершины стека *bp* на 2 байта, ее адрес *word ptr [bp-2]*, переменной *b* – на 4 байта *word ptr [bp-4]*, а переменной *c* – ячейка, сдвинутая на 6 байт: *word ptr [bp-6]*.

Для хранения данных процессор использует стек, а для произведения операций, в нашем случае – сложения, используются *регистры* центрального процессора. Регистры – это ячейки памяти, расположенные непосредственно в самом процессоре, поэтому доступ к ним наиболее быстрый по сравнению со всеми остальными видами памяти ПК. Регистров не много, все они имеют собственные имена. В нашем примере используется один двухбайтовый регистр *AX*.

3 этап. Ассемблирование

Полученный ассемблерный текст далее передаётся программе-ассемблеру, которая преобразует его в *объектный файл* (**.obj*). Объектный файл представляет собой бинарные коды *процессора*, дополненные информацией о метках и их использовании. Информация, содержащаяся в объектном файле, принципиально ничем не отличается от информации, содержащейся в ассемблерном тексте. Только весь код вместо мнемоник, понятных человеку, содержит *двоичный код*, понятный машине. А вместо меток ассемблерного текста, объектный файл содержит специальную *таблицу символов* (*symbol table*), описывающую все метки из нашего ассемблерного текста и *таблицу перемещений* (*relocations table*), описывающую точки, где метки использовались.

Важным моментом является то, что в объектном файле адреса глобальных (внешних) меток ещё не настроены, т.к. они будут известны только на этапе линковки. А все обращения к локальным меткам ассемблер меняет на *смещения* внутри объектного файла.

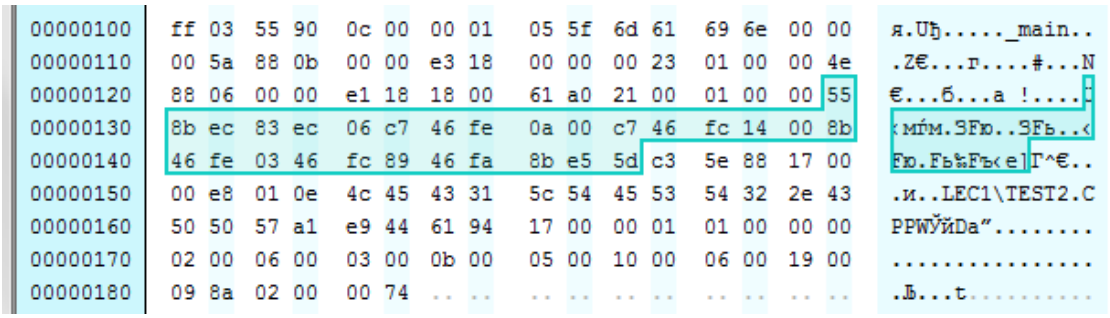

Рис. 2. Фрагмент программы (Листинг 2) в машинных кодах

На *Рис. 2* приведен полученный ассемблированием фрагмент нашей программы в машинных кодах (для просмотра использован редактор бинарных файлов). Выделенная на рисунке область соответствует рассматриваемому фрагменту программы в файле **.obj* (это результат процесса компиляции ассемблерного кода **.asm*). На следующем ниже рисунке показано как команды ассемблера записываются в виде машинных кодов.

Можно видеть (*Рис. 3*), что машинные команды и данные сгруппированы в пары шестнадцатеричных чисел. Каждая такая пара – один байт, описывающий выполняемую процессором команду или данные – параметры команды.

```
_main proc near
      push bp                    //    ;     55
      mov bp, sp                 //    ;     8B EC
      sub sp, 6                  //    ;     83 EC 06
```

```
        mov word ptr [bp-2], 10         //    ;     C7 46 FE 0A 00
        mov word ptr [bp-4], 20         //    ;     C7 46 FC 14 00
        mov ax, word ptr [bp-2]         //    ;     8B 46 FE
        add ax, word ptr [bp-4]         //    ;     03 46 FC
        mov word ptr [bp-6], ax         //    ;     89 46 FA
        mov sp, bp                      //    ;     8B E5
        pop bp                          //    ;     5D
        ret
_main endp
```

Рис. 3. Соответствие команд ассемблера машинным кодам

4 этап. Линковка (компоновка)

Полученный объектный файл (а их может быть несколько) отдаётся *линковщику (link)*. Линковщик склеивает между собой все поданные ему (**.obj*)-файлы и формирует один большой *исполняемый файл*. Помимо объектных файлов компилятор подаёт в линковщик ещё и *библиотеки* (заранее откомпилированные). Какие-то библиотеки компилятор подаёт невидимым для пользователя образом (т.е. пользователь непосредственно в этом процессе не участвует). Какие-то библиотеки пользователь сам просил компилятор передать линкеру (при помощи директив *#include*).

Упрощенно к библиотекам можно относиться следующим образом. Где-то кто-то написал реализации некоторых нужных нам функций, далее все эти реализации были откомпилированы до объектного файла, названы словом "библиотека" и помещены в файл с расширением **.lib*, или **.dll*. Т.е. *библиотека* - это набор уже откомпилированных кодов. Далее эту библиотеку, совместно с заголовочными файлами к ним (**.h, *.hpp*), включили в состав компилятора или в состав операционной системы.

Помимо склеивания файлов линковщик ещё и занимается настройкой адресов. В соответствии с таблицей перемещений линковщик однозначно определяет, по какому адресу будет располагаться каждая функция или переменная уже с учетом адресов внешних подпрограмм во внешних библиотеках. Для каждого импортируемого имени находится его определение в других модулях, упоминание имени меняется на его адрес.

Линковщик (также редактор связей, компоновщик) – программа, которая принимает на вход один или несколько объектных модулей и библиотек и собирает по ним исполнимый модуль. (**.exe, *.com*).

1.2. Программирование на Visual C

Современные отладочные программные средства – *компиляторы*, производят все описанные выше этапы трансляции и генерируют исполняемый файл. Но, помимо этого, имеют возможности для удобного написания *программного кода*, его отладки и тестирования. К наиболее популярным *средам разработки* программ на языке C++ относятся *Visual Studio (Microsoft)* и *C++Builder (Borland)*. Использование этих сред имеет свои особенности, с которыми можно будет познакомиться позже.

Рассмотрим процесс создания программы в программной среде *Visual Studio*. Первый шаг – создание проекта (*Рис. 4*). Среда Visual Studio представляет собой интегрированный комплекс разработки программных продуктов, объединяющий *несколько* языков программирования, при создании проекта в нашем случае, разумеется, следует выбрать в качестве среды разработки Visual C++.

После этого в диалоговом режиме пользователю будет предложено выбрать шаблон (*Template*) и ввести имя проекта (*Name, Location*). При выборе шаблона *Visual Studio* автоматически создает проект с указанным именем и помещает в главный рабочий файл первые строки программного кода основной программы *int _tmain()*.

В созданном файле (*.cpp) и предстоит писать вашу первую программу.

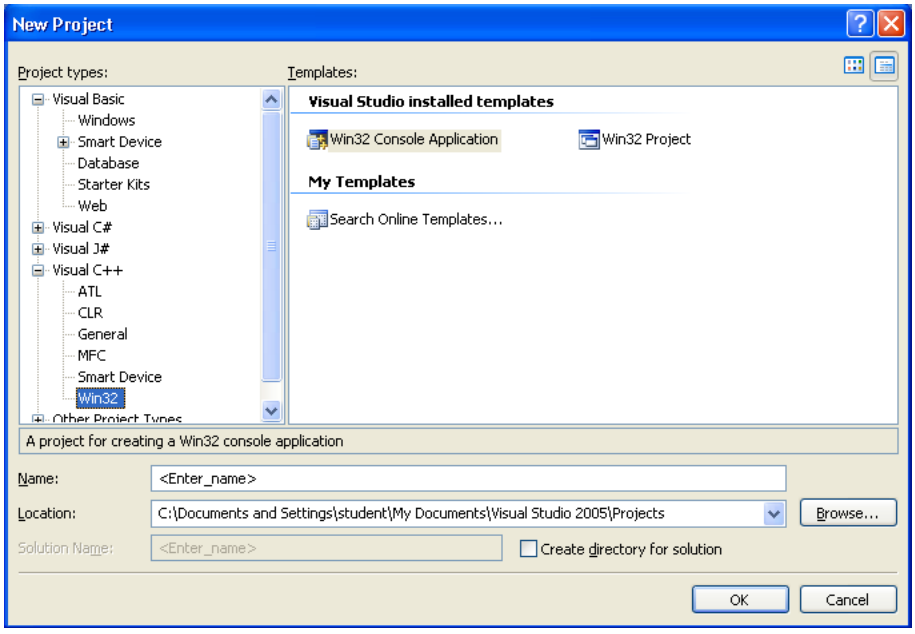

Рис. 4. Создание проекта на Visual C: Выбор шаблона, задание имени проекта

1.3. Структура программы на языке C++

Программа на языке C++ имеет следующую структуру [1, 2]:

Листинг 3

```
#include "stdafx.h" // подключение библиотек
#include <stdio.h>
int _tmain(int argc, char* argv[]) // заголовок основной программы
{
      // тело основной программы
}
```

Функция *main* (или *_tmain*) – главная функция, с нее начинается выполнение программы. Если имеются другие подпрограммы-функции, то они вызываются из главной функции. Именно функция *main( )* определяет входную точку – метку *main*. Фигурные скобки { } служат для обозначения блока последовательно выполняемых команд, в данном примере – тела функции.

Директива *#include* используется для подключения библиотек. После служебного слова в треугольных скобках < > или в двойных кавычках " " указывается имя заголовочного файла библиотеки.

Листинг 4 содержит пример простейшей программы на C++. Рассмотрим его. В первой строке – подключение библиотеки *<stdio.h>* форматного ввода-вывода. Во второй строке – главная функция *int main( )* – в круглых скобках нет никаких параметров – входных переменных, но служебное слово *int* обозначает, что сама функция должна иметь возвращаемое значение целого типа.

В теле главной функции – между фигурными скобочками *{ }* – в третьей строке задается переменная *str* – строка из 30 символов и заполняется следующим текстом: *"Всем привет! Это программа..."*. Каждая команда в C++ заканчивается символом точка с запятой – ";". Четвертая строка начинается с символов "//" – комментарий, это означает, что всё написанное правее символа "//" не воспринимается и не обрабатывается компилятором, комментарии нужны для удобства программиста.

В пятой строке программы стоит вызов подпрограммы *printf( )*, осуществляющей форматный вывод строки *str* на экран. В круглых скобках функции *printf("%s\n", str)* указан формат вывода строки *"%s\n"* и, через запятую, имя

выводимой текстовой переменной – *str*. Как задавать формат вывода будет рассказано в следующем параграфе.

Листинг 4

```
1  #include <stdio.h>
2  int _tmain(int argc, char* argv[])
3  {       char str[30]= {"Всем привет! Это программа..."};
4          // задана строка из 30 символов
5          printf("%s\n", str); // вывод строки на экран
6          system("pause"); // пауза
7          return 0; // возврат из главной программы
8  }
```

В 6 строке вызов системной функции, обеспечивающей паузу – ожидание нажатия пользователем любой кнопки. В 7 строке оператор *return 0*, возвращающий нулевой значение главной функции и прерывающий ее выполнение.

Это ваша первая программа на C++. Нажав комбинацию клавиш <Ctrl-F9> или вызвав из меню опцию «*Build Solution*» (*Рис. 5*) можно запустить ее на выполнение.

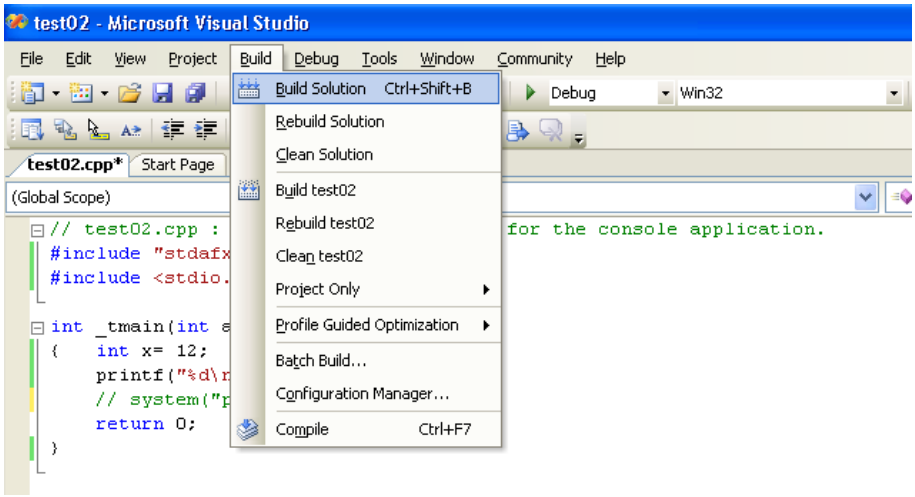

Рис. 5. Запуск созданного программного кода на компиляцию и выполнение

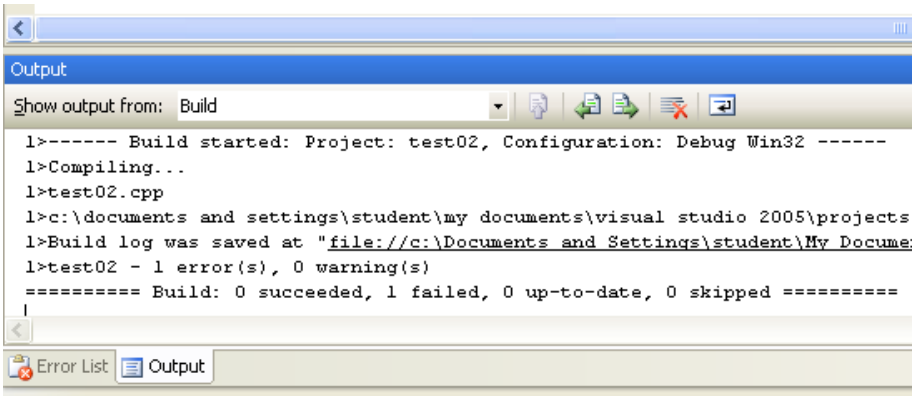

Рис. 6. Окно Output: вывод информации о компиляции проекта и наличии ошибок

В нижней части рабочего окна *Visual C++* имеется несколько вкладок, отражающих процесс выполнения программы. На *Рис. 6* можно видеть окно вывода «*Output*», в котором отражается процесс компиляции и выполнения программы, а так же наличие в ней ошибок. На *Рис. 6*, например, показано, что процесс компиляции файла *test02.cpp* прерван из-за обнаруженной ошибки.

Для просмотра информации об обнаруженных компилятором ошибках необходимо перейти во вкладку «Error List» (*Рис. 7*). В этом окне выводятся сообщения

трех видов: ошибки (error), предупреждения (warning) и сообщения (messages). По каждой ошибке указывается имя файла и номер строки, в которой она обнаружена, код ошибки и текстовое объяснение этой ошибки (на английском языке). В примере (*Рис. 7*) указывается, что ошибка найдена в седьмой строке, ее код *C2065*, а ее суть – обращение к неописанному выше идентификатору *x* (*"x": undeclared identifier*). Список всех ошибок находится в интерактивной справочной системе *Visual C++*, вызываемой командной клавишей *<F1>*.

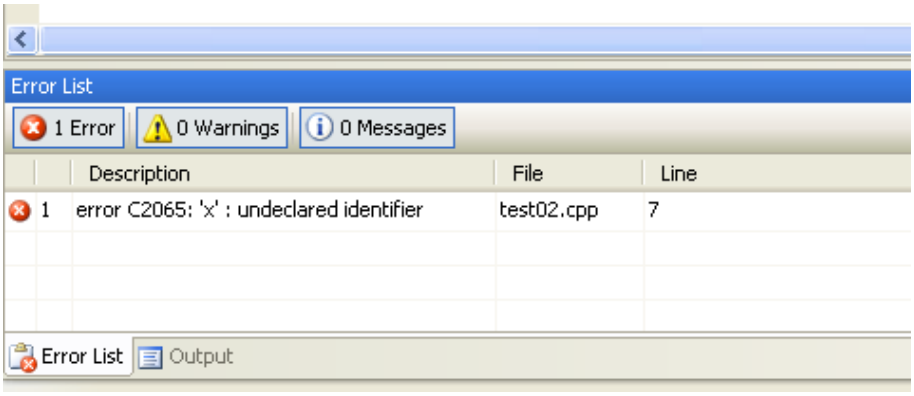

Рис. 7. Окно «Error List»: вывод информации об ошибках

Если ошибок не обнаружено, то проект (полученный исполняемый файл «*.exe») направляется на выполнение (*Рис. 8*).

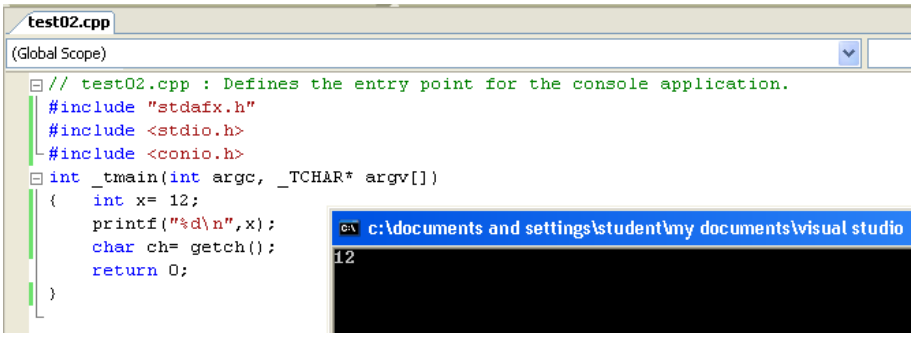

Рис. 8. Выполнение программы в консольном окне

1.4. Стандартные типы данных языка C++

```
Настоящий программист считает, что
в километре 1024 метра.
```

Сердце любой программы это *переменные* – именованные ячейки памяти соответствующего типа.

В языке C++ имеется следующий набор стандартных числовых *типов данных*:

- целые числа (int, short int, long);
- символьные переменные (char, wchar_t);
- вещественные числа с плавающей точкой (float, double, long double);
- указатели;
- логические переменные (bool);
- тип *void*.

Целочисленные переменные хранятся в памяти в виде довичного кода и могут занимать лишь целое число байт, так как любые данные в памяти адресуются побайтно. Все байты в памяти пронумерованы, номер байта – это его *адрес*. Программа может получить доступ к переменной (иначе говоря – группе байтов) *по имени* либо *по адресу* – по номеру того байта, с которого начинается переменная. *Таблица 1* демонстрирует названия стандартных числовых типов данных, их размер и диапазон их значений.

Таблица 1. Диапазоны значений стандартных типов данных C

Тип	Размер (байт)	Диапазон значений
bool	1	true и false; 1 или 0
char	1	-128 .. 127
unsigned char	1	0 .. 255
wchar_t	2	0 .. 65535
short int	2	-32768 .. 32767
unsigned short int	2	0 .. 65535
int	4	-2147483648 .. 2147483647
unsigned int	4	0 .. 4294967295
long	4	-2147483648 .. 2147483647
unsigned long	4	0 .. 4294967295
float	4	1.17549e-38 .. 3.40282e+38
double	8	2.22507e-308 .. 1.79769e+308
long double	10	3.4e-4932 .. 1.1e+4932

Размер переменной – это просто количество занимаемых ею байт. Например, переменная типа *char* занимает 1 байт или 8 бит. Значит, в ячейке памяти типа *char* может располагаться любое значение от самого маленького – 00000000 до самого большого – 2^8-1 = 11111111. В десятичном формате этот диапазон от 0 до 255, но числа все будет положительными. Каким образом задаются отрицательные числа?

Очень просто: если нам нужны только положительные значения – то, указав служебное слово *unsigned* (беззнаковый), мы весь диапазон числа от 0 до максимального значения 2^R-1 выделяем для положительных чисел – *unsigned char, unsigned int, unsigned long*. Если же нам нужны числа со знаком – *signed* (знаковые), то диапазон разбивается надвое, как бы сдвигается наполовину в отрицательную область. И теперь самое маленькое двоичное число 00000000 обозначает не 0, а число -128, а самое большое двоичное число 11111111 обозначает не 255, а десятичное 127. В ячейках *signed char* и *unsigned char* располагаются комбинации нулей и единичек из одного и того же диапазона, но интерпретируются по-разному. Служебное слово *signed* принято не писать (*Таблица 1*).

Аналогично разбивается диапазон двухбайтовых чисел *short int*: беззнаковый от 0 = 00000000 00000000 до 2^{16}-1 = 11111111 11111111 = 65 535, а знаковый от 00000000 00000000 = -32 768 до 11111111 11111111 = 32 767. Так же для четырехбайтовых целых чисел: *unsigned int* (или *unsigned long*) от 0 до 2^{32}-1 = 4 294 967 295, а знаковый *int* (или *long*) от -2 147 483 648 до 2 147 483 647.

Тип данных определяет не только размер ячейки памяти для хранения переменной и вид интерпретации двоичного кода, но и операции, производимые с переменной, способ выводе на экран и т.п.

Операции

Наиболее популярными операциями языка C++ являются следующие: *операция присваивания (=), арифметические операции (+) сложение, (-) вычитание, (*) умножение, деление (/)* и *остаток от деления (%)*. Операция деления применима к операндам арифметического типа, если оба операнда целочисленные, результат операции округляется до целого числа, в противном случае – до вещественного. Широко используются *операции увеличения и уменьшения на 1 (++)*инкремент и *(--)* декремент. Операции сравнения (<, <=, >, >=, ==, !=) сравнивают первый *операнд* (переменную) со вторым, результатом операции является логическое (*bool*) значение *true* или *false*. Для работы с переменными типа *bool* применяются *логические* операции (!) – отрицание, (&&) - *логическое «И»* и (||) – *логическое «ИЛИ»*.

Листинг 5

```
double x= 1.5, y=3.5, z=4.0;    // задать переменные и присвоить им значения
double a, b;                    // задать переменные
a= x+y*z; // a==15.5
b= (x+y)*z; // b==20.0
```

Полный список операций языка *C++* приводится в таблице (*Таблица 2*) вместе с их приоритетами *[3, 4]*. Операции выполняются в соответствии с приоритетами (очередностью) аналогично математическим операциям. *Приоритетом* математической операции называется очередность ее выполнения по сравнению с другими операциями. Для изменения порядка выполнения операций, как и в математике, используются круглые скобки.

Таблица 2 Приоритеты операций языка C++

Операция	Краткое описание
Унарные операции	
++	увеличение на 1
—	уменьшение на 1 (пробелы между символами не допускаются)
sizeof( )	размер
~	поразрядное отрицание
!	логическое отрицание
–	арифметическое отрицание (унарный минус)
+	унарный плюс
&	взятие (получение) адреса переменной
*	разадресация – получение значения переменной по известному адресу
new	выделение памяти
delete	освобождение памяти
(type)	преобразование типа
Бинарные операции	
*	умножение
/	деление
%	остаток от деления
+	сложение
–	вычитание
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
==	равно
!=	не равно
&	поразрядная конъюнкция (И)
^	поразрядное исключающее ИЛИ
|	поразрядная дизъюнкция (ИЛИ)
&&	логическое И
||	логическое ИЛИ
? :	условная операция
=	присваивание
*=	умножение с присваиванием
/=	деление с присваиванием

Операция	Краткое описание
%=	остаток отделения с присваиванием
+=	сложение с присваиванием
–=	вычитание с присваиванием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное И с присваиванием
|=	поразрядное ИЛИ с присваиванием
^=	поразрядное исключающее ИЛИ с присваиванием
,	последовательное вычисление

1.5. Двоичный формат хранения данных

Как известно еще из школьного курса информатики, вся информация хранится в памяти компьютера в двоичном виде: и тексты, и программы, и видео, и изображение, и звук – все данные представляют собой не более чем последовательность нулей и единиц.

Рассмотрим, в каком виде хранятся в памяти компьютера переменные рассмотренных в предыдущем параграфе типов.

1-байтное целое число

Его 8 разрядов (бит) пронумерованы слева направо от 7 до 0, "*номера разрядов*" соответствуют степеням 2. Самое большое число, которое может содержать этот байт, имеет во всех разрядах 1 : $11111111_2=255_{10}$.

$$11111111_2= 1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*1^0 = 255_{10}$$
$$10110011_2= 1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*1^0 = 179_{10}$$

Для визуального представления чисел могут использоваться так же *восьмеричный* и *шестнадцатеричный* форматы.

Восьмеричный формат числа

"*Восьмеричными*" (или *oct*) называются числа в системе счисления по основанию 8. В этой системе различные позиции в числе представляют степени числа 8. Мы используем для этого цифры от 0 до 7. Например, восьмеричное число 451_8 (записываемое как 0451 на языке С) представляется в виде:

$$451_8= 4*8^2 + 5*8^1 + 1*8^0 = 297_{10}$$

Шестнадцатеричный формат числа

"*Шестнадцатеричными*" (или *hex*) называются числа в системе по основанию 16. Поскольку у нас нет отдельных цифр для представления значения от 10 до 15, мы используем в этих целях буквы от A до F. Например, шестнадцатеричное число A3F (записанное как 0xA3F на языке Си) представляется как:

$$A3F_{16}= 10*16^2 + 3*16^1 + 15*16^0 = 2623_{10}$$

Числа с плавающей точкой

Для представления в компьютере числа с плавающей точкой (*float, double и long double*) некоторое количество (в зависимости от системы) разрядов выделяется для хранения двоичной дроби и, кроме того, дополнительные разряды содержат показатель степени. В десятичной арифметике для записи таких чисел используется алгебраическая форма. При этом число записывается в виде мантиссы, умноженной на 10 в степени, отображающей порядок числа:

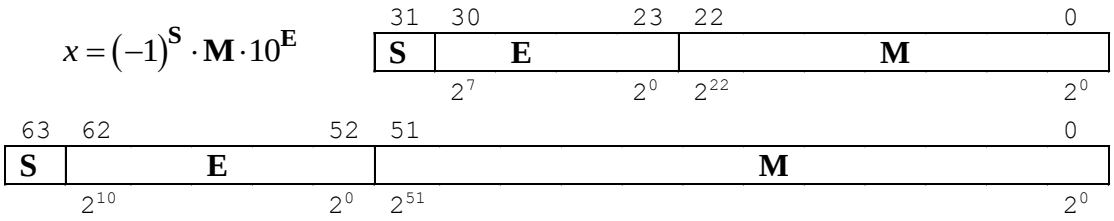

Рис. 9. Форматы числа в формате с плавающей запятой

Для записи двоичных чисел тоже используется такая форма записи. Эта форма записи называется запись числа с плавающей точкой. На рисунке (*Рис. 9*) буквой (**S**) обозначен *знак числа*, 0 - это положительное число, 1 - отрицательное число, (**E**) обозначает *смещённый порядок числа*. Напомним, что мантисса (**M**) не может быть больше единицы и после запятой в мантиссе не может записываться ноль.

Изначально процессоры персонального компьютера и микропроцессоры имели возможность работать только с целочисленными данными, а для работы с вещественными числами и математическими операциями требовались соответствующие (весьма громоздкие) подпрограммы поддержки и значительное время для их выполнения. Поэтому вплоть до процессоров *i386* параллельно с центральным процессором использовался сопутствующий *математический сопроцессор* — сопроцессор для расширения множества команд и обеспечивающий возможность работы с числами типа *float, double* или *long double*.

Начиная с *i486* процессоров модуль операций с плавающей точкой (*floating point unit – FPU*) интегрирован в центральный процессор для выполнения широкого спектра математических операций над вещественными числами. Это позволяет значительно ускорить такие операции.

Символьный тип данных char

Тип *char* предназначен для хранения символов, таких как буквы различных языков, цифры и знаки препинания. Самым распространенным набором символов является ASCII, Каждый символ в этом наборе представлен числовым кодом (ASCII). Например, символу 'A' соответствует код 65, и т.д.

Для примера рассмотрим фрагмент программного кода, приведенный ниже (*Листинг 6*).

Листинг 6

```
#include "stdafx.h" // подключение библиотек
#include <stdio.h>
#include <conio.h>

int _tmain(int argc, _TCHAR* argv[]) // главная программа
{     setlocale(LC_ALL, "Russian"); // подключение русского шрифта
      char ch= 'M'; // объявление переменной char и присвоение ей значения
M
      printf("вывод символа %c на экран\n", ch); // вывод символа на экран
      return 0;
}
```

В верхней части программы расположены опции компилятору *#include*, подключающие три программных библиотеки *"stdafx.h"*, *<stdio.h>* и *<conio.h>*. Причем, библиотеки, названия которых приводятся в угловых скобках < > будут разыскиваться компилятором в стандартных папках библиотек *Windows* или *Visual Studio*. А библиотеки, названия которых заключены в двойные кавычки - *" "*, размещаются в той же папке, где и исполняемый файл проекта. Затем идет заголовок главной исполняемой программы:

int _tmain(int argc, _TCHAR argv[])*. Первое слово *int* означает тип возвращаемого функцией значения (результат), после идет название программы *_tmain*, а затем – в круглых скобочках список аргументов (параметров) функции, о нем мы поговорим позже.

Открывающаяся фигурная скобка *{* – это начало «тела» программы, ее алгоритма, а закрывающаяся фигурная скобка *}* – конец программы. Между этими скобками располагается сама программа, в данном случае, состоящая из четырех строк. Первая из них *setlocale(LC_ALL, "Russian")* предназначена для подключения русского шрифта, как она работает, вы узнаете позже.

Во второй строке тела программы *char ch= 'M';* совмещены две операции: *char ch* – объявление компилятору о том, что в программе задана *статическая* переменная символьного типа с именем *ch*. Кроме того, этой переменной присвоено значение *'M'*.

Следующая строка *printf("вывод символа %c на экран\n", ch);* - это вызов подпрограммы *printf( )*, предназначенной для вывода на экран текста, заключенного в двойные кавычки: *«вывод символа M на экран»*. Как работают подпрограммы ввода-вывода, будет рассказано в следующих параграфах (*п. 1.6, 1.7*). Последняя строка *return 0;* прерывает работу программы и возвращает значение, в данном случае 0.

Обратите внимание, что каждая команда в C++ завершается символом «точка с запятой» – это обязательное правило. На самом деле переменная *ch* – это числовая ячейка памяти размером в 1 байт и в ней хранится число 77 – код символа *M* в ASCII таблице. Программист вводит символ *'M'*, а в ячейку попадает код символа, равный 77. При выводе на экран подпрограмма *printf( )* выводит на экран символ *'M'*, а не код 77 – то значение, которое хранится в переменной *ch*. Это объясняется не свойствами типа *char*, а работой подпрограммы *printf( )* по интерпретации выводимого на экран кода.

При написании символьной константы в языке C++ символ заключается в одинарные кавычки: *' '*. Заметьте, что для строковой константы используются двойные кавычки: *" "*.

Некоторые символы невозможно ввести в программу прямо с клавиатуры. Например, символ новой строки, символы двойной кавычки, символ табуляции и т.п. Для некоторых таких символов в языке C++ используются специальные обозначения, называемые *управляющими последовательностями* (*Таблица 4*). Например, последовательность *'\t'* представляет собой табуляцию, а последовательность '\n' – переход на новую строку.

Если требуется больше символов: wchar_t

Иногда программа должна обрабатывать наборы символов, которые не вписываются в 8 битов ASCII кода (пример – система японских иероглифических писем или символы кириллицы). На этот случай в языке C++ имеется пара способов. Во-первых, если большой набор символов является базовым набором символов для реализации, то производитель компилятора может определить *char* как 16-битовый тип, или даже больше. Во-вторых, реализация может поддерживать как малый базовый набор символов, так и расширенный набор. Традиционный 8-битовый тип *char* может представлять базовый набор символов, а другой тип, называемый *wchar_t (wide character type)* – расширенный тип символов. Тип *wchar_t* является целочисленным типом, имеющим достаточно памяти для представления самого большого расширенного набора символов в системе.

1.6. Функции форматного ввода-вывода printf( ) и scanf( )

В распоряжении программиста находится довольно большой список функций ввода-вывода (*Таблица 5*) наиболее применимыми из них являются подпрограммы форматного ввода-вывода *printf( ) и scanf( )*.

Функция *printf( )* позволяет выводить информацию на экран при программировании в консольном режиме в определенном *формате*. Данная функция определена в библиотеке <*stdio*> и имеет такой синтаксис:

```
int printf("формат", список выводимых переменных );
```

Здесь первый аргумент *"формат"* – *форматная спецификация* определяет вид строки, которая выводится на экран и может содержать специальные управляющие символы для вывода переменных (КАК выводить данные на экран). Далее, идет необязательный *список выводимых переменных* – тех переменных, значения которых необходимо вывести на экран в составе форматной строки (ЧТО выводить на экран). Функция возвращает либо число отображенных символов, либо, в случае неправильной работы – отрицательное число.

```
printf("Выводимый текст");
```

Структура форматной спецификации *"формат"* имеет следующий общий вид:

```
%<флаги><ширина><.точность><модификатор>тип
```

Она предназначена для вывода переменных разного типа, от целочисленных до строковых. Для этого используются специальные управляющие символы, которые называются спецификаторами и которые начинаются со знака «процент» "%" :

Таблица 3 Модификаторы форматного ввода\вывода

Модификатор	Описание типа переменной
%c	***char*** при вводе читается и передается 1 байт; при выводе переменная преобразуется к типу *char*; в файл передается 1 байт
	int
%d	десятичное целое со знаком
%i	десятичное целое со знаком
%o	8-ричное целое без знака
%u	10-ное целое без знака
%x	16-ричное целое без знака. При выводе использует a:f
%X	16-ричное целое без знака. При выводе использует A:F.
	float
%f	число со знаком в формате <->dddd.ddd
%e	число со знаком в формате <->dddd.ddde<+ или ->ddd
%E	число со знаком в формате <->dddd.dddE<+ или ->ddd
%g	число со знаком в формате e или f в зависимости от указанной точности
%G	число со знаком в формате E или F в зависимости от указанной точности.
%s	***char **** или ***char[ ]*** (массив символов, строка) при вводе принимает символы без преобразования, пока не встретится разделитель '\n' или не достигнута указанная точность; при выводе выдает в поток все символы, пока не встретится '\0'.
%p	***pointer*** (указатель) выводит аргумент как адрес, формат зависит от модели памяти, в общем случае, включает 16-ричные сегмент и смещение

Пример, приведенный ниже (*Листинг 7*), иллюстрирует задание двух переменных *i* и *x*, присвоение им значений и вывод их на экран.

Листинг 7

```
#include "stdafx.h" // подключение библиотек
#include <stdio.h>
#include <conio.h>

int _tmain(int argc, _TCHAR* argv[]) // главная программа
{      setlocale(LC_ALL, "Russian"); // подключение русского шрифта

       int i =5; // задана переменная целого типа, ей присвоено значение 5
       float x = 10.5; // задана переменная вещественного типа, ей присвоено
10.5
       printf("переменная i = %d\n", i); // вывод на экран переменной i в
формате %d
       printf("переменная x = %f\n", x); // вывод на экран переменной x в
формате %f
       getch(); // ожидание нажатия клавиши
       return 0; // выход из программы
}
```

```
переменная i = 5
переменная x = 10,500000
```

Рисунок (*Рис. 10*) иллюстрирует структуру вызова подпрограммы форматного вывода, аналогичную тем, которые содержит *Листинг 7*.

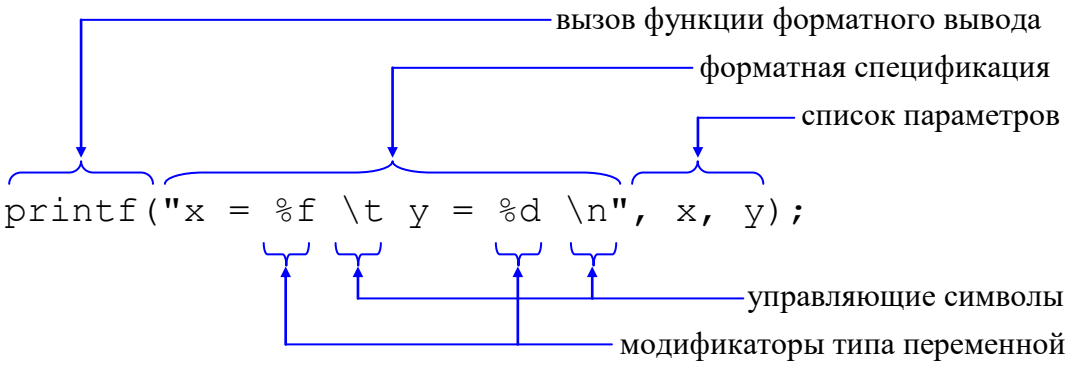

Рис. 10. Структура вызова подпрограммы форматного вывода

Параметрами функции *printf( )* выступают *строка форматной спецификации* и *список параметров* из двух переменных *x* и *y*, перечисленных через запятую. Форматная спецификация представляет собой символьную строку *"x = %f \t y = %d \n"*, заключенную в кавычки. Эта строка содержит невыводимые *управляющие последовательности* *'\t'* и *'\n'*, которые обеспечивают табуляцию и переход на новую строку (*Таблица 4*). Строка форматной спецификации содержит также *модификаторы типов переменных* *%f* и *%d* (*Таблица 3*), вместо которых будут *последовательно* подставлены значения переменных *x* и *y*. Причем, как следует из модификаторов типов выводимых переменных, первая переменная *x* будет отражена на экране как вещественная (*%f*), а вторая переменная *y* будет выведена в формате целого числа (*%d*).

```
printf("вывод переменных на экран\ni = %d,\t x = %f\n", i, f);
```

Строкой выше приводится пример использования нескольких управляющих последовательностей, их еще называют управляющими символами. Полный список управляющих символов содержит *Таблица 4*.

Таблица 4 Управляющие последовательности.

Символ	Управляющие символы
\r	возврат каретки в начало строки
\n	новая строка
\t	горизонтальная табуляция
\v	вертикальная табуляция
\"	двойные кавычки
\’	апостроф
\\	обратный слеш
\0	нулевой символ, конец строки символов
\?	знак вопроса
\a	сигнал бипера (спикера) компьютера

Все управляющие символы, при использовании, обрамляются кавычками, но если необходимо вывести какое-то сообщение, то управляющие символы можно записывать сразу в сообщении, в любом его месте, см. пример (*Листинг 8*).

Листинг 8

```
printf("\t\tВсем привет! Это программа.\n"); // две табуляции и печать
printf("1234567890-1234567890-1234567890\rЭто - то же.\n");
// возврат на начало строки и печать сообщения
printf("\'в кавычках\'  без кавычек  \"в двойных кавычках\"");
```

```
                Всем привет! Это программа.
Это - то же.234567890-1234567890
'в кавычках'  без кавычек  "в двойных кавычках"
```

Аналогично используется и функция *scanf( )*, эта функция предназначена для форматного *ввода* данных, она также располагается в библиотеки <*stdio*> и имеет приведенный ниже синтаксис:

```
int scanf(“формат”, список адресов вводимых переменных);
```

Переменная *«формат»* определяет форматную строку для определения типа вводимых данных и может содержать модификаторы (*Таблица 3*). Затем через запятую идет ***список адресов*** вводимых переменных.

Листинг 9

```
int _tmain(int argc, _TCHAR* argv[]) // главная программа
{     setlocale(LC_ALL, "Russian"); // подключение русского шрифта
      int age; // описание переменных
      float weight;
      char name[100];
      printf("Введите ваше имя: ");
      scanf("%s", &name); // ввод символьной строки
      printf("Введите ваш возраст: ");
      scanf("%d", &age); // ввод целого числа
      printf("Введите ваш вес: ");
      scanf("%f", &weight); // ввод вещественного числа
      printf("\nЗдравствуйте, %s.\tВаш возраст = %d, ваш вес = %f.\n",
name,
            age, weight);
      getch(); // ожидание нажатия клавиши
      return 0; // выход из программы
}
```

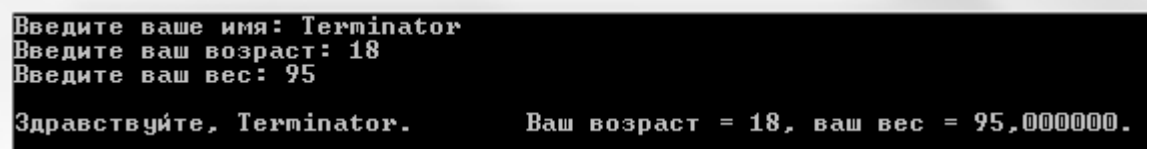

Рис. 11. Результаты работы программы

Особенно важно при вводе данных посредством функции *scanf( )* перед каждой переменной ставить знак «**&**» обозначающий, что данные вводятся «по адресу». Подробнее данная тема будет изучена в *главе 21* при изучении подпрограмм.

Функция *scanf( )* может работать сразу с несколькими переменными. Предположим, что необходимо ввести два целых числа с клавиатуры. Вообще, можно дважды вызвать функцию *scanf( )*, однако лучше воспользоваться такой конструкцией:

```
scanf(" %d %d ", &n, &m);
```

Функция *scanf( )* возвращает число успешно считанных элементов. Если операции считывания не происходило, что бывает в том случае, когда вместо ожидаемого цифрового значения вводится какая-то буква, то возвращаемое значение будет равно 0.

Листинг 10

```
int i =5;
float x = 10.513;
char c= 'A';
double z = -123567E-89;
printf("переменная i = %5d\n", i); // формат %d в 5 позициях
printf("переменная x = %8.5f\n", x);
// формат %f в 8 позициях и 5 знаков после запятой
printf("переменная c = %c, ее код равен %d\n", c, c);
// формат %c - символ %d - число
printf("переменная z = %9.3G или иначе %E\n", z, z); // формат %G и %E
```

```
переменная i =     5
переменная x = 10,51300
переменная c = A, ее код равен 65
переменная z = -1,24E-084 или иначе -1,235670E-084
```

Функции форматного ввода-вывода *scanf( )* и *printf( )* позволяют осуществлять ввод и вывод данных с указанием количества символов. Для этого при указании формата ввода-вывода между знаком «%» и модификатором формата ставится одна цифра для целочисленных аргументов и две – для вещественнозначных. Эти цифры указывают, сколько знаков выделяется под число и сколько под дробную часть.

В примере, приведенном выше (см. *Листинг 10*), строка форматной спецификации "*переменная i = %5d*", показывает, что переменную *i* требуется выводить на экран как целое число в пяти позициях. Строка "*переменная x = %8.5f*" задает вывод переменной *x* как вещественного целого числа в восьми позициях, из которых на дробную часть отводится пять позиций (еще одна позиция на десятичную точку и две на целую часть числа *x*).

Как уже обсуждалось выше, символьная переменная (в примере *Листинг 10*– *char c*) на самом деле – ячейка памяти, содержащая код символа. На этапе вывода на экран вместо кода 65 будет выводиться символ *'A'*. Рассмотрев форматную спецификацию "*переменная c = %c, ее код равен %d*", можно убедиться в этом – здесь одна и та же переменная *c* выводится дважды: сперва со спецификатором *%c* как символ, а затем со спецификатором *%d* как целое число.

1.7. Функции потокового ввода-вывода cin/cout и оператор <<

Потоковые объекты ввода/вывода *cout* и *cin* – это объекты классов *istream* (от *Input Stream* - поток ввода) и *ostream* (от *Output Stream* - поток вывода) соответственно, они используются при помощи операторов *извлечения из потока (>>)* и *вставки в поток (<<)* [6, 8]. Система классов ввода/вывода довольна сложна. Базовым классом является класс *std::ios* (от *Input/Output Stream* - потоковый ввод/вывод). У класса *ios* довольно много производных классов. На данный момент нас интересует лишь некоторые – *std::cin* и *std::cout* (префикс принадлежности к адресному пространству *std::* обычно опускается).

По умолчанию, в адресном пространстве *using namespace std*, поток вывода *std::cout* связан с видеодрайвером ОС, а поток ввода *std::cin* – с буфером клавиатуры, но они могут быть перенастроены на другие устройства.

Для использования потокового ввода и вывода необходимо подключить файл *<iostream.h>* и задать объект адресного пространства *using namespace std*, в котором создаются и настраиваются потоковые объекты *std::cout* и *std::cin*. Объекты *cout* и *cin* содержат ряд подпрограмм (методов) (*Таблица 5*) и важнейшими являются операторы *извлечения из потока (>>)* и *вставки в поток (<<)* для ввода с клавиатуры и вывода на экран, как показано в программе (*Листинг 11*):

Листинг 11

```
#include <stdio>

#include <iostream.h> // библиотека потокового ввода-вывода
using namespace std; // подключение адресного пространства

int _tmain(int argc, char* argv[])
{     cout << "Hello, World"; // оператор вывода в поток cout
      cin.get(); // подпрограмма чтения строки символов
      return 0;
}
```

Строка *cout << "Hello, World";* выводит на экран строку, вторая строка вызывает функцию *cin.get( )*, которая необходима для того чтобы организовать задержку до нажатия клавиши, последняя строка программы возвращает значение 0. Оператор << объекта *cout* позволяет последовательно выводить несколько данных подряд.

```
int a=10, b=5;
cout << a << "+" << b << "=" << a+b;
cout << "\n";
cout << a << "-" << b << "=" << a-b;
```

Разберем ввод данных с клавиатуры. Для этого разработаем приложение, спрашивающее у пользователя его имя.

Листинг 12

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

#include <iostream.h> // библиотека потокового ввода-вывода
using namespace std; // подключение адресного пространства

int _tmain(int argc, char* argv[])
{     char a[35]; // для хранения строки - массив из 35 символов
      cout << "Enter you name";
      cin >> a;
      cout << "\n" << "Hello, " << a;
      char ch= getch();
      return 0;
}
```

Как видно из приведенных выше листингов, в строковые константы, выводимые объектом *cout* в поток вывода и получаемые *cin* из потока ввода, можно включать управляющие последовательности (управляющие символы – см. *Таблица 4*).

Заголовочный файл <*conio.h*> включен в *Листинг 12* для организации задержки экрана посредством функции *getch( )*. Функция *ch = getch( )* ожидает нажатия символа без «эхо-ввода» (синхронного отражения на экране) и возвращает код нажатой клавиши в переменную *ch*.

При работе с символьными строками – массивами элементов типа *char* необходимо понимать, что в последней ячейке массива должен лежать символ конца строки: '**\0**', поэтому под строку символов нужно использовать массив на единицу большей размерности. Более подробно работа с символьными массивами (строками) будет разобрана в *главе 6*.

Листинг 13

```
#include <stdio>
#include <iostream.h>
using namespace std; // подключение адресного пространства

int _tmain(int argc, char* argv[])
{     char st[21]; // массив под 20 символов + 1 символ конца строки '\0'
      std::cin >> st; // ввод строки с клавиатуры
      cout << st; // вывод строки на экран
}
```

Альтернативой операторам *извлечения из потока (>>)* и *вставки в поток (<<)* могут выступать функции *gets( )* и *puts( )* потокового ввода-вывода см. *Листинг 14*. Отличия состоят в том, что оператор *std::cin >> st* вводит с клавиатуры только одно слово из строки, а функция *gets(st)* вводит всю строку, даже если она содержит пробелы, табуляцию и другие служебные символы.

Листинг 14

```
#include <stdio>
#include <iostream.h>
using namespace std; // подключение адресного пространства

int _tmain(int argc, char* argv[])
{     char st[21]; // массив под 20 символов + 1 символ конца строки '\0'
      gets(st); // ввод строки с клавиатуры
      puts(st); // вывод строки на экран
      puts("Hello world from puts!");
}
```

В таблице ниже (*Таблица 5*) при водятся стандартные функции ввода-вывода как форматного типа, так и потоковые. Нужно добавить, что в библиотеке работы со строками <*string.h*> содержатся свои собственные подпрограммы ввода-вывода – методы классов *string*.

Таблица 5 Стандартные функции ввода-вывода[1]

Функция	Описание
форматные	
printf	*int printf(const char *format<, argument, ...>);* запись данных в поток *stdout* по формату. В случае успеха вернет число записанных байт, в случае ошибки – *EOF*

[1] Здесь константа *EOF* – код, возвращаемый как признак конца файла, константа *NULL* – значение указателя, который не содержит адрес никакого реально размещенного в оперативной памяти объекта.

scanf	*int scanf(const char *format<, address, ...>);* чтение данных из потока *stdin* по формату.
sprintf	*int sprintf(char *buffer, const char *format<, argument, ...>);* запись данных в строку *buffer* по формату. В случае успеха вернет число записанных байт, в случае ошибки – *EOF*. Не включает нулевой байт в число записанных символов.
sscanf	*int sscanf (const char *buffer, const char *format<, address, ...>);* чтение данных из строки *buffer* по формату.
потоковые	
gets	*char *gets(char *s);* чтение строки из потока *stdin*. Заменяет символ конца строки нулевым байтом. Все символы, включая перевод строки, пишутся в строку s и возвращается указатель на нее. На конец файла или ошибку возвращает *NULL*
getc	*int getc (FILE *stream);* считывает один байт из указанного аргументом stream потока данных. В случае успешного чтения байта возвращается код считанного байта (символа), если достигнут конец файла, то возвращается *EOF* и устанавливается признак конца файла.
getch	*int getch(void);* чтение символа из потока *stdin*. Работает как *getc* для потока *stdin*
putc	*int putc (int sym, FILE *stream);* выводит один символ, код которого указывается в аргументе *sym*, в файл, привязанный к потоку данных на который указывает аргумент *stream*. В случае успешной выдачи байта возвращается код выведенного байта (символа), если при выводе байта произошла ошибка, то возвращается *EOF*, а переменной *errno* присваивается код ошибки.
putch	*int putch(int c);* запись символа в поток *stdout*. Возвращает записанный символ c. В случае ошибки вернет *EOF*
puts	*int puts(const char *s);* запись строки *s* в поток *stdout*. Добавляет символ перевода строки. В случае успеха вернет неотрицательное число, на ошибку вернет *EOF*

1.8. Явное и неявное преобразование типов данных

Производя математические действия, мы не задумываемся над операциями с числами различного типа, например, складывая вещественные числа с целыми. Но в языке программирования целочисленные типы (int, short, long) и вещественнозначные (float, double) хранятся в памяти по-разному (это обсуждалось в *п. 1.5*), операции над ними организованы различным образом и даже выполняются различными блоками процессора.

При программировании при выполнении таких смешанных операций производится *явное (или неявное) преобразование одного типа в другой*.

Явное преобразование типа (если оно возможно) осуществляется таким образом – в требуемом месте необходимо указать нужный тип данных в круглых скобках. Например, в листинге ниже (*Листинг 15*) переменной *y* присваивается константа **4** явно преобразованная в вещественный вид **4.0**.

Неявное преобразование типа введено в язык для удобства программиста. Если вещественной переменной присваивается целое значение, то это целое число предварительно переводится в вещественный формат добавлением нулевой дробной

части. Если переменной целого типа присваивается вещественное число, то дробная часть отбрасывается (но не округляется).

Листинг 15

```
double x= 1.99999, y=3.578;
int a=67, b=90;
a= x; // неявное преобразование типа - отбрасывание дробной части
printf("a = %d\n", a);
x= b; // неявное преобразование типа - добавление нулевой дробной части
printf("b = %d x = %f\n", b, x);
y= (double)4; // явное преобразование типа
printf("y = %14.12f\n", y);
```

```
a = 1
b = 90 x = 90,000000
y = 4,000000000000
```

С понятием неявного преобразования типов напрямую связана операция деления. Нужно понимать, что в математике существует *деление целочисленное* (деление нацело с остатком) и *деление рациональное* (деление десятичных дробей). В программировании эти операции обозначаются одним значком «/» и определяются по типу того операнда, который стоит *справа от знака деления*: если правый операнд (делитель) целый, то и деление будет целочисленное (*Листинг 16*), а если вещественный – то вещественнозначное.

Листинг 16

```
double x;
x= 13/2; // целочисленное деление
printf("x = %f\n", x);
x= 13/2.0; // неявное преобразование типа
printf("x = %f\n", x);
x= 13/(double)2; // явное преобразование типа
printf("x = %f\n", x);
```

```
x = 6,000000
x = 6,500000
x = 6,500000
```

Полезно знать так же оператор «%», позволяющий вычислить *остаток от деления* нацело.

1.9. Практическая работа №1. Типы данных. Ввод-вывод

Продолжительность – 2 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Освоение следующих умений, навыков, компетенций:

Создать проект на *Visual C++*, написать простейшую программу на языке C++, отладить и запустить на выполнение программу. Научиться задавать переменные и константы следующих типов: целого, вещественного, символьного и строкового – в соответствии с индивидуальным заданием. Освоить два способа ввода с клавиатуры и вывода на экран значений этих переменных – форматный и потоковый способы. Изучить спецификаторы, форматы ввода-вывода и управляющие символы.

Задание на практическую работу

1. Создать проект на *Visual C++*, написать первую программу на языке C++ – программу, выводящую на экран приветствие, отладить и запустить на выполнение программу. Продемонстрировать преподавателю выполнение программы.

2. Задать в соответствии с вариантом индивидуального задания переменные и присвоить им значения. Вывести на экран значения этих переменных потоковым (*cout*) способом. Продемонстрировать преподавателю выполнение программы.
3. Запрограммировать ввод заданных переменных с клавиатуры двумя способами (форматным – *scanf* и потоковым – *cin*). Вывести переменные на экран форматным способом в заданном формате. Продемонстрировать преподавателю выполнение программы.
4. Составить отчет, в котором отразить листинг программного кода с комментариями и привести скриншоты с результатами работы программ.

Варианты индивидуальных заданий

1. целую знаковую переменную **x** выводить в 5 позициях, вещественную **y** выводить в 3 позициях до десятичной точки и в 3 позициях после, символьную **z** выводить вместе с кодом, строковой **w** присвоить строку, содержащую вашу Фамилию И.О., номер группы, адрес и год рождения. Использовать не менее 3х различных управляющих символов.

2. целую беззнаковую переменную **A** выводить в 6 позициях, вещественную **B** выводить в 4 позициях до десятичной точки и в 4 позициях после, символьную **C** выводить вместе с кодом, строковой **W** присвоить строку, содержащую вашу Фамилию И.О., инициалы, город и год рождения. Использовать не менее 2х различных управляющих символов.

3. целую знаковую переменную **P** выводить в 7 позициях, вещественную **R** выводить в 7 позициях до десятичной точки и в 2 позициях после, символьную Q выводить вместе с кодом, строковой **S** присвоить строку, содержащую вашу Фамилию И.О., номер группы, рост и вес. Использовать не менее 4х различных управляющих символов.

4. целую беззнаковую переменную **x1** выводить в 8 позициях, вещественную **x2** выводить в 9 позициях до десятичной точки и в 3 позициях после, символьную **x3** выводить вместе с кодом, строковой **x4** присвоить строку, содержащую вашу Фамилию И.О., номер телефона, адрес и год рождения. Использовать не менее 3х различных управляющих символов.

5. целую знаковую переменную **I** выводить в 9 позициях, вещественную **J** выводить в 8 позициях до десятичной точки и в 1 позиции после, символьную **K** выводить вместе с кодом, строковой **L** присвоить строку, содержащую вашу Фамилию И.О., номер группы, адрес и имена ваших родителей. Использовать не менее 2х различных управляющих символов.

6. целую беззнаковую переменную **Q** выводить в 8 позициях, вещественную **W** выводить в 7 позициях до десятичной точки и в 2 позициях после, символьную **E** выводить вместе с кодом, строковой **R** присвоить строку, содержащую вашу Фамилию И.О., номер группы, адрес и название любимой песни. Использовать не менее 4х различных управляющих символов.

7. целую двухбайтовую переменную **T** выводить в 7 позициях, вещественную **Y** выводить в 6 позициях до десятичной точки и в 3 позициях после, символьную **U** выводить вместе с кодом, строковой **I** присвоить строку, содержащую вашу Фамилию И.О., адрес и год рождения и название любимой музыкальной группы. Использовать не менее 3х различных управляющих символов.

8. целую беззнаковую переменную **O** выводить в 6 позициях, вещественную **P** выводить в 5 позициях до десятичной точки и в 4 позициях после, символьную **A** выводить вместе с кодом, строковой **S** присвоить строку, содержащую ваши номер

группы, Фамилию И.О., адрес и место жительства. Использовать не менее 4х различных управляющих символов.

9. целую знаковую переменную **D** выводить в 5 позициях, вещественную **F** выводить в 4 позициях до десятичной точки и в 5 позициях после, символьную **G** выводить вместе с кодом, строковой **H** присвоить строку, содержащую вашу Фамилию И.О., номер группы, рост, вес и цвет волос. Использовать не менее 2х различных управляющих символов.

10. целую беззнаковую переменную **J** выводить в 4 позициях, вещественную **K** выводить в 2 позициях до десятичной точки и в 6 позициях после, символьную **L** выводить вместе с кодом, строковой **Z** присвоить строку, содержащую вашу Фамилию И.О., номер группы, и размер противогаза. Использовать не менее 3х различных управляющих символов.

11. целую двухбайтовую переменную **PQ17** выводить в 4 позициях, вещественную **N2N** выводить в 7 позициях до десятичной точки и в 3 позициях после, символьную **B31** выводить вместе с кодом, строковой **V4V** присвоить строку, содержащую вашу Фамилию И.О., номер группы, рост и национальность. Использовать не менее 3х различных управляющих символов.

12. целую двухбайтовую переменную **X** выводить в 3 позициях, вещественную **C** выводить в 3 позициях до десятичной точки и в 7 позициях после, символьную **V** выводить вместе с кодом, строковой **B** присвоить строку, содержащую ваши номер группы, Фамилию И.О., и группу крови. Использовать не менее 2х различных управляющих символов.

13. целую беззнаковую переменную **N** выводить в 4 позициях, вещественную **M** выводить в 4 позициях до десятичной точки и в 8 позициях после, символьную **K** выводить вместе с кодом, строковой **L** присвоить строку, содержащую вашу Фамилию И.О., номер группы, любимое лакомство. Использовать не менее 4х различных управляющих символов.

14. целую беззнаковую переменную **OoO** выводить в 6 позициях, вещественную **P** выводить в 7 позициях до десятичной точки и в 7 позициях после, символьную **AaA** выводить вместе с кодом, строковой **SsS** присвоить строку, содержащую ваши номер группы, Фамилию И.О., адрес и место жительства. Использовать не менее 2х различных управляющих символов.

15. целую двухбайтовую переменную **M1** выводить в 5 позициях, вещественную **N2** выводить в 5 позициях до десятичной точки и в 9 позициях после, символьную **B3** выводить вместе с кодом, строковой **V4** присвоить строку, содержащую вашу Фамилию И.О., номер группы, адрес и национальность. Использовать не менее 3х различных управляющих символов.

16. целую беззнаковую переменную **C5** выводить в 6 позициях, вещественную **X6** выводить в 6 позициях до десятичной точки и в 3 позициях после, символьную **Z2** выводить вместе с кодом, строковой **L3** присвоить строку, содержащую ваши номер группы, Фамилию И.О., национальность и место проживания. Использовать не менее 4х различных управляющих символов.

17. целую беззнаковую переменную **KJ** выводить в 15 позициях, вещественную **h5** выводить в 5 позициях до десятичной точки и в 5 позициях после, символьную **g9** выводить вместе с кодом, строковой **F8** присвоить строку, содержащую вашу Фамилию И.О., номер группы, адрес и количество ушей. Использовать не менее 2х различных управляющих символов.

18. целую двухбайтовую переменную **D6** выводить в 7 позициях, вещественную **A1** выводить в 4 позициях до десятичной точки и в 6 позициях после, символьную **S2**

выводить вместе с кодом, строковой **P5** присвоить строку, содержащую вашу Фамилию И.О., номер телефона и год рождения. Использовать не менее 3х различных управляющих символов.

19. целую беззнаковую переменную **x3** выводить в 8 позициях, вещественную **O7** выводить в 3 позициях до десятичной точки и в 3 позициях после, символьную **I1** выводить вместе с кодом, строковой **u8** присвоить строку, содержащую вашу Фамилию И.О., номер группы и место рождения. Использовать не менее 2х различных управляющих символов.

20. целую беззнаковую переменную **O** выводить в 4 позициях, вещественную **F** выводить в 2 позициях до десятичной точки и в 6 позициях после, символьную **L** выводить вместе с кодом, строковой **M** присвоить строку, содержащую вашу Фамилию И.О., номер группы, и рост. Использовать не менее 3х различных управляющих символов.

21. целую беззнаковую переменную **Y** выводить в 3 позициях, вещественную **C** выводить в 3 позициях до десятичной точки и в 7 позициях после, символьную **Q** выводить вместе с кодом, строковой **W** присвоить строку, содержащую ваши номер группы, Фамилию И.О., и группу крови. Использовать не менее 2х различных управляющих символов.

22. целую двухбайтовую переменную **DB** выводить в 5 позициях, вещественную (double) **AA** выводить в 8 позициях до десятичной точки и в 7 позициях после, символьную **SA** выводить вместе с кодом, строковой **PA** присвоить строку, содержащую вашу Фамилию И.О., номер телефона и год рождения. Использовать не менее 3х различных управляющих символов.

23. целую беззнаковую переменную **C5C** выводить в 6 позициях, вещественную **X6xX** выводить в 3 позициях до десятичной точки и в 6 позициях после, символьную **Z22** выводить вместе с кодом, строковой **L3OU** присвоить строку, содержащую ваши номер группы, Фамилию И.О., национальность и место проживания. Использовать не менее 3х различных управляющих символов.

Контрольные вопросы:

1. Что представляет собой программа на языке C++?
2. Что такое библиотеки и как их подключить к своей программе?
3. Как задается переход на новую строку при выводе на экран?
4. Как использовать функции потокового ввода-вывода?
5. Как в форматном спецификаторе указать, в скольких ячейках выводить число?
6. Для чего нужна строка форматной спецификации, как она выглядит?
7. Как использовать функции форматного ввода-вывода?
8. Что хранится в ячейке символьного типа (char)? Как это значение выводится на экран?

2. Алгоритмические конструкции языка C++

```
Программист – это конвертер
пожеланий заказчика в
жесткую формальную логику.
```

Архитектура любой программы, создаваемой программистом, может быть представлена в виде комбинации конечного набора конструкций. В языке C++ эти конструкции реализуются при помощи операторов [*1 - 8*].

Операторы выполняются последовательно, один за другим, как написано в программном коде. Рассмотрим их.

2.1. Операторы выбора

Оператор if-else

Условный оператор *if* выполняет указанный оператор, если условие истинно (не равно 0). Иначе, если присутствует оператор *else*, выполняется альтернативный оператор. Допускаются вложения условных операторов, при этом *else* относится к ближайшему *if*.

```
if (условие_истинно)
{       оператор; }
else // если условие не выполняется
{       альтернативный_оператор; }
```

Пример:

```
if (i==5)
        cout << "i=" << I << endl;
```

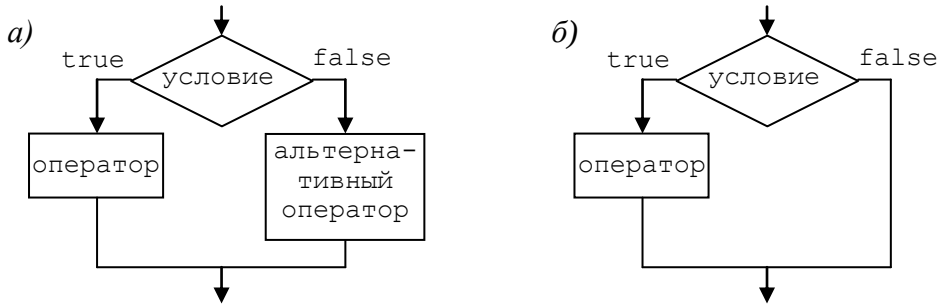

Рис. 12. Блок-схемы условного оператора а) if-else; б) if

Возможна ситуация, когда альтернативного оператора может не быть (*см. Рис. 12,б*).

Листинг 17

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

int _tmain(int argc, char* argv[])
{       int a;
        cout << "введите переменную a: ";
        cin >> a;
        if (a>0) // если a>0
        {       cout << "переменная a=" << a << " положительная\n"; }
        else // если условие a>0 не выполняется
        {       if (a<0) // если a<0
                {       cout << "переменная a=" << a << " отрицательная\n"; }
                else // если условие a>0 не выполняется
```

```
            {     cout << "переменная a=" << a << " равна нулю\n"; }
      }
      getch();
      return 0;
}
```

```
введите переменную a: -5
переменная a=-5 отрицательная
```

Оператор switch

Оператор выбора *switch* передает управление на одну из меток *case* в зависимости от значения целочисленного выражения. Если значение выражения не предусмотрено в блоке *switch* и метка по умолчанию *default* отсутствует, то ничего не происходит. В качестве метки используются только целочисленные (или перечисляемые – *enum*) константы. Для выхода из блока *case* служит оператор *break*. Если после выполнения блока *case* отсутствует оператор *break*, то немедленно начнет выполняться следующий блок *case*.

На листинге ниже (*Листинг 18*) представлен примитивный фрагмент программы, в котором по коду нажатого символа *ch* происходит распознавание нажатой клавиши.

Листинг 18

```
char ch=getch();
switch(ch)
{       case 27: cout << "нажата клавиша Esc" << endl; break;
        case 13: cout << "нажата клавиша Enter" << endl; break;
        case 49: cout << "нажата клавиша 1" << endl;
        case 50: cout << "нажата клавиша 2" << endl; break;
        default: cout << "нажата другая клавиша" << endl;
};
```

```
нажата клавиша 1
нажата клавиша 2
```

Обратите внимание на то, что при нажатии клавиши «1» будут выполнены операторы *case 49* и *case 50* так как оператор *break* после выбора *case 49* отсутствует. Проверьте выполнение программного кода на листинге (*Листинг 18*).

В языке C++ некоторые управляющие символьные последовательности (*Таблица 4*) заданы специальными константами, например символ *'\n'* - «конец строки» определен в библиотеке *<stdio.h>* константой *endl* (см. *Листинг 18*).

2.2. Перечислимый тип данных (enum)

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения. Для этого удобно воспользоваться *перечисляемым типом данных* - *enum*, все возможные значения которого задаются списком целочисленных констант.

```
enum имя_типа { список_констант };
```

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа имеет значение нуля, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

Листинг 19

```
enum Err {ERR__READ, ERR__WRITE, ERR_CONVERT}; // задан тип
Err error; // определена переменная типа Err

switch (error)
{       case ERR_READ:          /* операторы */ break;
        case ERR_WRITE:         /* операторы */ break;
        case ERR_CONVERT:       /* операторы */ break;
}
```

Константам *ERR_READ, ERR_WRITE, ERR_CONVERT* в примере (*Листинг 19*) присваиваются значения 0, 1 и 2 соответственно. Однако если программисту удобно нумеровать константы перечисляемого типа иначе, он может это сделать:

```
enum {two = 2, three, four, ten = 10, eleven, fifty = ten + 40};
// Константам three и four присваиваются значения 3 и 4, константе eleven — 11
```

Имена перечисляемых констант должны быть уникальными, а значения могут совпадать. Компилятор при инициализации констант выполняет проверку типов.

2.3. Операторы цикла

В языке *C++* три вида *цикла*, которые, в общем, взаимозаменяемы. Предназначение циклов многократное выполнения *оператора*, пока *истинно* (не равно 0) проверяемое на каждом цикле *условие.*

Если условие цикла всегда истинно, в теле цикла необходимо предусмотреть дополнительное условие выхода из цикла, иначе получится бесконечный цикл. Каждый шаг выполнения тела цикла называется итерацией. Для немедленного выхода из цикла служит оператор *break*.

Оператор *continue* выполняет переход с текущей позиции на начало следующей итерации.

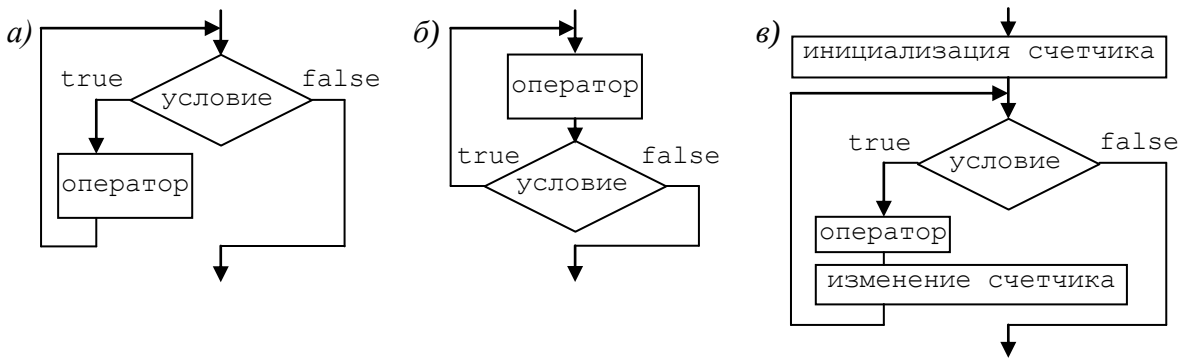

Рис. 13. Операторы цикла а) с предусловием; б) с постусловием; в) со счетчиком

Цикл с предусловием

Данный цикл определяется оператором *while* (*Рис. 13,а*). На каждой итерации сначала проверяется условие, а затем выполняется оператор или выход из цикла. Если условие изначально ложно, то оператор не будет выполнен ни разу.

Листинг 20

```
int i=0;
while (i<=10) // пока условие истинно
{       cout << i ; // оператор
        i++; // для выхода из цикла
}
```

Цикл с постусловием

В цикле с постусловием *do...while* (*Рис. 13,б*) условие проверяется после выполнения тела цикла. Таким образом, всегда выполняется хотя бы одна итерация цикла.

Листинг 21

```
int i;
do // цикл с постусловием
{      cout<<i; // оператор
       i++; } // для выхода из цикла
while (i<=10); // условие
```

Цикл по счетчику

В цикле по счетчику *for* (*Рис. 13,в*) *условие* проверяется до выполнения тела цикла так же как у *while*. Однако здесь дополнительно можно указать *начальные значения* для списка локальных переменных цикла и *поститерационные действия*, производимые после выполнения каждой итерации цикла.

```
for (инициализация_локальных_переменных; условие;
поститерационные_действия)
{ оператор; }
```

Перед началом выполнения тела цикла производится *инициализация локальных переменных* цикла и перед каждой итерацией проверяется *условие*. В зависимости от истинности условия выполняется *оператор* или выход из цикла. После каждой итерации производятся *поститерационные действия*:

Листинг 22

```
For (int i=0; i<10; i++)
{       cout<<i; }
```

Напомним, что имеются три оператора немедленного выхода из цикла – *break*, *continue* и *return*. Оператор *return* выбрасывает рабочую точку программы из цикла и из текущей программы (или подпрограммы). Оператор *break* служит для немедленного выхода из цикла, а оператор *continue* – для безусловного перехода из текущей итерации цикла к началу следующей.

2.4. Использование переменных логического типа (bool)

При формировании условия выхода из цикла (*Рис. 13*) или условия выбора (*Рис. 12*) довольно часто приходится организовывать весьма сложные логически конструкции на базе логических операторов (*см. Таблица 6*). Результатом действия какой-либо логической операции может являться одно из двух логических значений (*boolean*): ***false*** (ложь) или ***true*** (истина).

Таблица 6 Логические операции языка C++

Операция	Краткое описание	Пример
Операции сравнения		
<	меньше	(7<8) == true; (5<5) == false;

Операция	Краткое описание	Пример
<=	меньше или равно	(8<=7) == false; (4<=4) == true;
>	больше	(6>5) == true; (6>6) == false;
>=	больше или равно	(7>=7) == true; (8>=9) == false;
==	равно	(4==4) == true; (6==7) == false;
!=	не равно	(4!=4) == false; (6!=7) == true;
Логические операции		
&&	логическое И	(true && true) == true; (true && false) == false;
составное условие истинно, если истинны оба простых условия		
||	логическое ИЛИ	(true || true) == true; (true || false) == true;
составное условие истинно, если истинно, хотя бы одно из простых условий		
!	логическое отрицание	(!true) == false; (!false) == true;
Тернарная логическая операция		
? :	тернарная логическая операция	min = (a < b) ? a : b;

Тернарная логическая операция имеет следующий формат:

```
Логическое_выражение ? операнд2 : операнд3;
```

Она возвращает свой *второй* или *третий* операнд в зависимости от значения *логического выражения,* заданного первым операндом. С точки зрения математической логики тернарная условная операция реализует алгоритм: «*Если логическое_выражение истинно, то операнд2, иначе операнд3*», или иначе: «*операнд2 или операнд3, в зависимости от того, истинно или ложно логическое_выражение*».

Для упрощения сложных логических конструкций используются переменные логического типа (*bool*), принимающие два логических значения – *false* или *true*.

Листинг 23

```
bool flag= true; // логическая переменная “истина”
int i; // переменная целого типа
while (flag) // пока flag == true цикл будет повторяться
{       cin >> i ; // ввести переменную i с клавиатуры
        flag = !( ((i%13)<7) && (!(i<14)));
}
```

В примере, приведенном выше (*Листинг 23*), задается логическая переменная *flag* и ей сразу же присваивается значение «*true*». Цикл *while* повторяется до тех пор, пока переменная *flag* сохраняет истинное значение. В теле цикла вводится с клавиатуры простое число i, после чего логической переменной *flag* присваивается значение запутанного логического выражения, которое можно понимать так: *«переменная flag станет ложной, если остаток от деления числа i на 13 меньше 7 и одновременно с этим неверным будет являться выражение, что число i меньше 14»*. При помощи логических переменных такого типа конструкцию можно упростить, разбив на простые для понимания куски.

На самом деле переменные логического типа (*bool*) являются целыми числовыми. И не будет ошибкой присваивание логической переменной любой целой константы:

```
bool x = 3; // x == true
```

Такие присваивания - это неявное преобразование типов данных (*п. 1.8*). Любое число неравное нулю интерпретируется компилятором как *true* (истина), а нулевое значение понимается как ***false*** (ложное).

Несмотря на то, что для хранения значения типа *bool* достаточно всего одного бита – *false* или *true*, однако в памяти переменная такого типа занимает целый байт. Это связано с технологией *побайтной* адресации памяти ПК.

2.5. Оператор безусловного перехода

Оператор *goto* безо всяких проверок сразу же передает управление программой оператору, помеченному *меткой*, которая задана в операторе *goto*.

```
goto метка;
...
метка: оператор
```

Все метки должны оканчиваться двоеточием и не должны вступать в конфликт ни с какими ключевыми словами или именами функций. Кроме того, оператор *goto* может передавать управление только внутри текущей функции (не от одной функции к другой).

Современная культура программирования требует от программиста использовать оператор *goto* только в самых крайних случаях, когда без него обойтись невозможно. Это требование обусловлено тем, что в программе изобилующей этим оператором практически невозможно разобраться.

Несмотря на то, что следует избегать использования оператора *goto* в качестве универсальной формы управления циклами, в особых случаях его все-таки можно применять с большим успехом:

Листинг 24

```
for (int i=0; i<100; i++)
{       for (int j=0; j<10; j + +)
        {       for (int k=0; k<20; k++)
                {       cout << i << j << k << endl;
                        if ( i*j*k == 259)
                                goto done; // переход на метку done
                }
        }
}
done:
        cout << "i*j*k=259 - стоп";
```

2.6. Организация диалога с пользователем

Теперь, ознакомившись с алгоритмическими конструкциями языка C++ и используя подпрограммы ввода-вывода (*п. 1.6, 1.7*), можно построить простейшую программу, реализующую диалог с пользователем (*Листинг 25*). Рассмотрим ее.

В этой программе сначала, при помощи перечисляемого типа *enum*, задаются глобальные константы, содержащие коды необходимых нам клавиш – служебных *«enter»* и *«esc»,* буквенных *«N», «Y», «n»* и *«y»,* а также числовых – *«0», «1», «2»* и *«3».*

Программа *main* представляет собой вечный цикл *while(true) {...},* в котором считывается код нажатой клавиши *ch* и при помощи оператора множественного выбора *switch(ch)* выполняется один из блоков *case* – *default*, в зависимости от кода, хранящегося в переменной *ch.*

Листинг 25

```
enum {    kbEsc=27, kbEnter=13,
     kbY=89, kby=121, kbN=78, kbn=110,
     kb0=48, kb1=49, kb2=50, kb3=51 }; // используемые клавиши

int _tmain(int argc, char* argv[])
{ setlocale (LC_ALL, "Russian");

  char ch=0; // переменная для кода нажатой клавиши
  cout << "нажмите клавиши 0, 1, 2 или 3. Выход - клавиша Esc или Enter\n";

  while (true) // бесконечный цикл
  {  ch= getch(); // получить код нажатой клавиши
     switch(ch)
     {    case kbEsc: return 0; // выход из программы по нажатию «esc»

          case kbEnter: break; // выход из switch

          case kb0: continue; // следующая строка не выполнится
               cout << "выполнена команда continue"; break;

          case kb1: { cout << "любите программировать на C++? (Y\\N):";
               ch= getch();
               if ((ch==kbY)||(ch==kby)||(ch==kbEnter))
                    cout << "это очень хорошо!" << endl;
               else
                    if ((ch==kbN)||(ch==kbn)||(ch==kbEsc))
                         cout << "вы попробуйте, и вам понравится!";
                    else
                         cout << "непонятненько..." << endl;
               ch=0; break; }

          case kb2: cout << "клавиша 2, после нее не стоит break\n";
                    // отсутствует break;

          case kb3: cout << "клавиша 3, после нее стоит break\n"; break;

          default: printf("нажата непонятная клавиша %c код %d\n",ch,ch);
               break;

     } // end of switch
     if (ch==kbEnter) break; // выход из цикла по нажатию «enter»
  } // end of while

  cout << "вышли из цикла по нажатию Enter";
  system("pause");
  return 0;
}
```

Если нажата клавиша *«esc»,* то немедленно выполняется команда выхода из программы *return 0.*

Если нажата клавиша *«enter»,* то не выполняется никаких действий внутри оператора *switch,* однако после выхода из оператора *switch,* производится проверка *if (ch==kbEnter),* после чего выполняется оператор *break* – выход из цикла *while.* Протрассируйте этот код и сравните технологии выхода по нажатию кнопки *«esc»* и *«enter».*

Обратите внимание на использование оператора *continue* , срабатывающего в том случае, когда нажата клавиша *«0».* Оператор *continue* предназначен для того, чтобы пропустить все остальные операторы на этом шаге цикла и сразу перейти к началу следующей итерации цикла. Поэтому строчка, подсвеченная в листинге серым цветом *cout << "выполнена команда continue"; break,* никогда выполнена не будет.

Если в переменную *ch* попадает код клавиши *«1»,* то в операторе *switch* срабатывает выбор *case kb1,* который содержит целый блок операторов. В этом блоке выводится вопрос пользователю, еще раз считывается код нажатой клавиши *ch* и проверяется её содержимое – если в ней код клавиши *«Y», «y»* или *«enter»,* то

печатается некоторое сообщение, если в *ch* код клавиши *«N», «n»* или *«esc»,* то другое и во всех прочих случаях – третье.

Заметили ли вы, что каждый блок *case* завершается оператором *break*? Для чего он нужен и что будет, если его не поставить? На этот вопрос отвечает вариант выбора *case kb2*. Протрассируйте программу (*Листинг 25*) и вы увидите, что при нажатии кнопки *«2»* выполняется блок операторов *case kb2*., и сразу же за ним блок операторов *case kb3*. Т.е. оператор *break* необходим для того, чтобы прервать выполнение *switch.*

2.7. Практическая работа № 2. Операторы языка С.

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Изучить операторы языка С++, научиться применять их при написании программ. Научиться отслеживать значения числовых и логических переменных и выражений в процессе отладки программы. Освоить умение переводить мнемонические выражения в последовательность операторов.

Задание на практическую работу

1. Преобразовать индивидуальное задание из мнемонического описания в последовательность операторов языка С++, построить блок-схему алгоритма и доказать его адекватность текстовому заданию.
2. Создать проект на Visual C++, написать программу на языке С++ в соответствии с разработанным алгоритмом, отладить ее и запустить на выполнение. Продемонстрировать преподавателю выполнение программы.
3. Все числовые константы по заданию вводить с клавиатуры. Ввод данных организовать в диалоговом интерактивном режиме.
4. Выводить на экран не только конечные значения переменных, но и их текущие значения в процессе расчета.
5. Произвести тестирование программы, задавая, по возможности, нереальные и не предусмотренные заданием значения переменных. На этапе тестирования программы произвести проверку на зацикливание – предусмотреть невозможность попадания программы в «вечный цикл».
6. Составить отчет, в котором отразить листинг программного кода с комментариями, привести блок-схему алгоритма и скриншоты с результатами работы программы.

Таблица 7. Варианты индивидуальных заданий

1. задать переменную i равную -10. задать переменную j равную 28. пока i меньше 23, повторять следующие действия: { если i равно j, то i увеличить вдвое, иначе j уменьшить на 1. увеличить i на 2 }.	2. задать переменную q равную -30. задать переменную r равную 6. пока q меньше или равно 20, повторять следующие действия: { если q равно r, то q увеличить втрое, иначе r уменьшить на 2. увеличить q на 3 }.
3. задать переменную M равную 28. задать переменную L равную 10. пока L меньше 19, повторять следующие действия: { если L не равно M, то M уменьшить на 4, иначе L увеличить вдвое. увеличить L на 1 }.	4. задать переменную D равную -6. задать переменную W равную 8. пока D не станет равно 12, повторять следующие действия: { если D не равно W, то W увеличить втрое, иначе W уменьшить на 5. увеличить D на 3 }.

5. задать переменную I равную 10. задать переменную Y равную 8. пока I больше -3, повторять следующие действия: { если Y равно I, то I уменьшить вдвое, иначе Y уменьшить на 10. уменьшить I на 3 }.	6. задать переменную U равную 13. задать переменную V равную 6. пока U больше или равно -13, повторять следующие действия: { если U равно V, то V увеличить втрое, иначе U уменьшить на 2. уменьшить U на 3 }.
7. задать переменную P равную 28. задать переменную Q равную 10. пока P больше 1, повторять следующие действия: { если P не равно Q, то Q уменьшить на 4, иначе P уменьшить вдвое. уменьшить P на 1 }.	8. задать переменную D равную 6. задать переменную W равную 18. пока D не станет равно -12, повторять следующие действия: { если D не равно W, то W увеличить втрое, иначе W уменьшить на 5. уменьшить D на 3 }.
9. задать переменную A равную -10.234. задать переменную B равную 0.28. пока A меньше 2.3, повторять следующие действия: { если A равно B, то A возвести в квадрат, иначе B уменьшить на 1.3. увеличить A на 2.5 }.	10. задать переменную x равную -3.14. задать переменную y равную 6.28. пока x меньше или равно 20.34, повторять следующие действия: { если x равно y, то x увеличить втрое, иначе y уменьшить на 2.36. увеличить x на 3.14 }.
11. задать переменную m равную 28.6. задать переменную k равную 10.7. пока k меньше 1.9, повторять следующие действия: { если m не равно k, то m уменьшить на 4, иначе k увеличить вдвое. увеличить k на 0.1 }.	12. задать переменную f равную -6.0. задать переменную g равную 8.4. пока f не станет равно 12.0, повторять следующие действия: { если f не равно g, то g увеличить втрое, иначе g уменьшить на 5.7. увеличить f на 3.0 }.
13. задать переменную H равную 10.5. задать переменную h равную 8.6. пока H больше -3.5, повторять следующие действия: { если h равно H, то H уменьшить вдвое, иначе h уменьшить на 10.3. уменьшить H на 3.5 }.	14. задать переменную z равную 13.2. задать переменную s равную 6.3. пока z больше или равно -13.1, повторять следующие действия: { если z равно s, то s увеличить втрое, иначе z уменьшить на 2.2. уменьшить z на 3.7 }.
15. задать переменную P равную 2.8. задать переменную Q равную 10.1. пока P больше 1.13, повторять следующие действия: { если P не равно Q, то Q уменьшить на 0.4, иначе P уменьшить вдвое. уменьшить P на 1.1 }.	16. задать переменную j равную 7.2. задать переменную i равную 1.8. пока j не станет равно -1.2, повторять следующие действия: { если j не равно i, то i увеличить втрое, иначе i уменьшить на 5.1. уменьшить j на 1.2 }.
17. задать переменную A равную -10. задать переменную B равную 8. повторять следующие действия: { если A равно B, то A поделить пополам, иначе B уменьшить на 3. увеличить A на 5 }, пока A меньше 3.	18. задать переменную x равную -3. задать переменную y равную 6. повторять следующие действия: { если x равно y, то x увеличить втрое, иначе y уменьшить на 2. увеличить x на 3 }, пока x меньше или равно 20.

19. задать переменную m равную 26. задать переменную k равную 17. повторять следующие действия: { если m не равно k, то m уменьшить на 4, иначе k увеличить вдвое. увеличить k на 1 }, пока k меньше 1.	20. задать переменную f равную -6. задать переменную g равную 14. повторять следующие действия: { если f не равно g, то g увеличить втрое, иначе g уменьшить на 5. увеличить f на 3 }, пока f не станет равно 12.
21. задать переменную H равную 11.51. задать переменную h равную 7.6. повторять следующие действия: { если h равно H, то H уменьшить вдвое, иначе h уменьшить на 1.3. уменьшить H на 3.5 }, пока H больше -13.5.	22. задать переменную z равную 17.2. задать переменную s равную 2.3. повторять следующие действия: { если z равно s, то s увеличить втрое, иначе z уменьшить на 0.2. уменьшить z на 0.7 }, пока z больше или равно -3.1.
23. задать переменную P равную 2.81. задать переменную Q равную 10.11. повторять следующие действия: { если P не равно Q, то Q уменьшить на 0.43, иначе P уменьшить вдвое. уменьшить P на 1.17 }, пока P больше 1.73.	24. задать переменную j равную 11.2. задать переменную i равную 1.18. повторять следующие действия: { если j не равно i, то i увеличить втрое, иначе i уменьшить на 5.11. уменьшить j на 1.12 }, пока j не станет равно -1.12.

Контрольные вопросы:

1. Чем отличаются операторы выбора от операторов цикла?
2. Как организовать бесконечный цикл?
3. Что такое «пошаговая трассировка программного кода»? Для чего она используется?
4. Сколько раз выполняется оператор, содержащийся в конструкции *if* ?
5. Как увидеть текущее значение переменной во время выполнения программы?
6. Как определить, сколько раз будет выполнен оператор, содержащийся в конструкции *for*?
7. Для чего используется оператор *break*? Что будет, если его применить в теле цикла?
8. Что такое счетчик цикла? Как его использовать в операторе *for*?
9. Какие значения могут принимать логические переменные? Можно ли их задавать числами? Какой объем памяти они занимают?
10. Сколько раз выполняется оператор, содержащийся в конструкции *if-else* ?
11. Каковы правила использования оператора множественного выбора *switch*? Для чего нужен блок *default*?
12. Чем отличается цикл с предусловием от цикла с постусловием? Приведите пример.
13. В чем особенность оператора continue? Как его применяют?
14. Что такое *тернарная логическая операция*? Какова семантика её применения?
15. Как используется оператор *break*? Что будет, если его пропустить в одном из *case*-блоков оператора *switch*?

3. Указатели

— Зайди сюда, сынок!
— Отсутствует указатель.
— НА КУХНЮ ЗАЙДИ!!!

Указатель - это *переменная, которая содержит адрес* некоторого объекта (переменной, ячейки памяти, функции) [*1 - 4*]. Говорят, что указатель *ссылается* на переменную, функцию и т.д. Понятно, что *адрес* – это целое число, являющееся, по сути, номером байта в памяти компьютера. Понимание и правильное использование указателей очень важно для создания хороших программ. Многие конструкции языка C++ требуют применения указателей. Например, указатели необходимы для успешного использования функций и динамического распределения памяти. С указателями следует обращаться очень осторожно, но обойтись без указателей в программах на языке C++ невозможно.

Указатель описывается при помощи символа "*":

```
тип_данных * имя_переменной_указателя;  // типизированный указатель
void * имя_переменной_указателя; // нетипизированный указатель
```

Специальных операций для работы с указателями всего две: это унарная операция "&", означающая *"взять адрес переменной"* – *адресация* и унарная операция "*", означающая *"взять значение переменной, расположенное по указанному адресу"* – *разадресация.*

Листинг 26

```
int x= 101; // задать переменную x целого типа и присвоить ей значение 101
int* y; // задать переменную y – указатель на переменную целого типа
y= &x; // присвоить переменной-указателю y адрес переменной x
int z = *y; // переменной z присвоить значение, хранящееся в ячейке с
адресом y
```

3.1. Типизированные и нетипизированные указатели

Типизированный указатель указывает на ячейку памяти определенного типа, и использование его для адресации ячейки другого типа будет ошибкой. Нетипизированный указатель значение *void** содержит абстрактный адрес, который может указывать на любую ячейку памяти, не фиксируя тип этой ячейки. Прежде чем использовать указатель типа *void**, его необходимо принудительно преобразовать в типизированный.

Указатели можно использовать как операнды в арифметических операциях. Над указателями можно выполнять унарные операции: инкремент и декремент. При выполнении операций "++" и "--" значение указателя увеличивается или уменьшается на 1. Но не на 1 байт, а на 1 ячейку того типа, на который он указывает.

Спецификация "*%p*" функции *printf()* в C++ позволяет вывести на экран адрес памяти в шестнадцатеричной форме (*Листинг 27*).

Из рисунка (*Рис. 14*), иллюстрирующего работу программы (*Листинг 27*), можно видеть, что адрес указателя *pw* после операции *pw++* изменился на 4 – на *sizeof(int)* – размер ячейки памяти типа *int*. Указатели и целые числа можно складывать.

Конструкция *pw+3* задает адрес объекта, смещенный на 3 ячейки относительно той, на которую указывает *pw*. Это справедливо для любых объектов (*int*, char*, float** и др.); транслятор будет масштабировать приращение адреса в соответствии с типом, указанным в определении объекта.

Это называется *косвенный доступ* к ячейкам памяти «*адрес + смещение*».

Листинг 27

```
int x= 101;
void *v= &x;        // нетипизированному указателю v присвоен адрес
переменной x
printf(" x= %d, v= %p \n", x, v);
int *pw= (int*)v;            // явное преобразование типа void* к типу int*
printf(" x= %d, v= %p, pw= %p\n", x, v, pw);
pw++;
printf(" v= %p, pw= %p\n", v, pw);
```

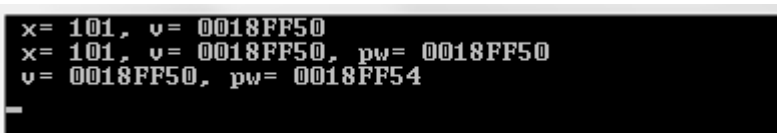

Рис. 14. Результаты выполнения программы

Значения двух указателей на одинаковые типы также можно сравнивать в операциях ==, !=, <, <=, >, >= при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен логическому значению 0 (*false*) или 1 (*true*).

Листинг 28

```
int *ptr1, *ptr2=NULL, a=10;
ptr1= &a+5;
ptr2= &a+7;
if (prt1>ptr2)
        cout << "ptr1>ptr2"; // не будет выполнено
```

Для задания *несуществующего (нулевого) адреса* заведена константа-указатель *NULL*. Любой адрес можно проверить на равенство (==) или неравенство (!=) со специальным значением *NULL*, которое позволяет определить ничего не адресующий указатель.

3.2. Статическое и динамическое распределение памяти

До сих пор мы пользовались *статическими* переменными – но самое главное в языке высокого уровня – это способность работать с объектами *динамическими*.

В C++ объекты могут быть размещены либо *статически* – во время компиляции, либо *динамически* – во время выполнения программы. Статическое размещение более эффективно, так как выделение памяти происходит до выполнения программы, однако оно гораздо менее гибко, потому что мы должны заранее знать тип, размер и количество размещаемых объектов.

Основные отличия между *статическим и динамическим* выделением памяти таковы:

- статические объекты обозначаются *именованными переменными*, и действия над этими объектами производятся напрямую, с использованием их имен. *Динамические объекты не имеют собственных имен*, и действия над ними производятся косвенно, с помощью указателей;
- выделение и освобождение памяти под статические объекты производится компилятором *автоматически*, программисту не нужно самому заботиться об этом. Выделение и освобождение памяти под динамические объекты целиком и полностью *возлагается на программиста*. Это достаточно сложная задача, при решении которой легко наделать ошибок;
- статические и динамические переменные размещаются в различных областях оперативной памяти (ОЗУ). Память под статические переменные отводится в *стеке* (*stack*), специально выделенном для данной программы. Динамические переменные

создаются в общедоступной области ОЗУ, называемой *куча* (*heap*); выделение памяти из кучи осуществляется по запросу к операционной системе.

Для манипуляции динамически выделяемой памятью служат операторы *new* и *delete*.

Листинг 29

```
int *pint = new int(1024);
```

Здесь оператор *new выделяет память* под безымянный объект типа int, *инициализирует* его значением 1024 и *возвращает адрес созданного объекта*. Этот адрес присваивается указателю *pint*. Все дальнейшие действия над таким *безымянным* объектом производятся только посредством указателя, т.к. явно манипулировать динамическим объектом невозможно.

С помощью оператора *new* можно выделять память под *массив* заданного размера, состоящий из элементов определенного типа:

```
int *pia = new int[4];
```

В этом примере память выделяется под массив из четырех элементов типа *int* и возвращает указателю *pia* адрес первого элемента массива из четырех ячеек памяти типа *int*. Работа с массивами будет рассмотрена в следующей главе (*см. Гл. 5*).

Когда динамический объект больше не нужен, мы должны явным образом освободить отведенную под него память. Это делается с помощью оператора *delete*:

```
delete pint; // освобождение единичного объекта
delete[] pia; // освобождение памяти, занимаемой массивом
```

В программе ниже (*Листинг 30, Рис. 15*) приводится пример работы с указателями. Указатель *a_ptr* хранит адрес статической ячейки с именем *a* (типа *double*). А указатель *x_ptr* указывает на *безымянную* динамическую ячейку памяти (она выделена цветом на *Рис. 15*).

Листинг 30

```
double a = 10.1;
double *a_ptr= &a;
double *x_ptr= new double(13.5);
delete x_ptr; // освобождение объекта
```

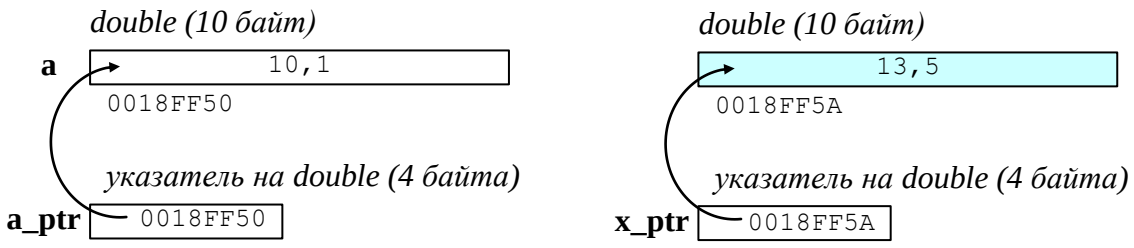

Рис. 15. Динамическое выделение памяти

Динамические объекты хранятся в динамической памяти – в «куче» (*heap*) как правило, это все пространство ОЗУ между стеком программы и верхней границей физической памяти. Именно механизм динамического распределения памяти позволяет динамически запрашивать из программы дополнительные области оперативной памяти.

Что случится, если мы забудем освободить выделенную память и переприсвоим указатель? Память будет расходоваться впустую: она окажется неиспользуемой, однако возвратить ее системе невозможно, поскольку у нас нет указателя на нее, а имен

динамические объекты не имеют. Такое явление получило специальное название *утечка памяти*, она трудно поддается обнаружению.

Рассмотрим пример, приведенный ниже (*Листинг 31*, *Рис. 16*):

Листинг 31

```
#include "stdafx.h"
#include <conio.h>
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{    setlocale(LC_ALL, "Russian"); // подпрограмма русского шрифта
     double *temp; // указатель на ячейку типа double
     int i= 0;
     while (i<=100)
     {    temp= new double(i*i); // Утечка памяти!!!
          printf("динамическая ячейка с адресом %p в ней лежит число %f\n",
          temp, *temp);
          i++;
     }
     delete temp; // освобождение только последнего объекта
     getch();
     return 0;
}
```

В приведенном примере заводится указатель *temp*, которому в цикле 101 раз присваиваются адреса новой выделяемой операционной системой динамической ячейки памяти. Из рисунка (*Рис. 16*) видно, что адрес выделяемой ячейки каждый раз новый (отличающийся от предыдущего на 8 байт – размер ячейки типа *double*).

На следующем шаге цикла выделяется новая динамическая ячейка, и ее адрес вновь присваивается указателю *temp*, а динамическая ячейка памяти, созданная на предыдущем шаге не освобождается (!). Поскольку у динамической ячейки нет имени, а в указателе *temp* лежит уже новое значение адреса, доступ к предыдущей ячейке теряется, а память при этом не освобождена.

В конце программы (*Листинг 31*) память, лежащая по указателю *temp*, очищается командой *delete temp*, но это делается один раз и удаляется только одна последняя динамическая ячейка по адресу *0021A418*, а предыдущие 100 ячеек с адресами *002177B8 – 0021A3D0* утеряны и не очищены.

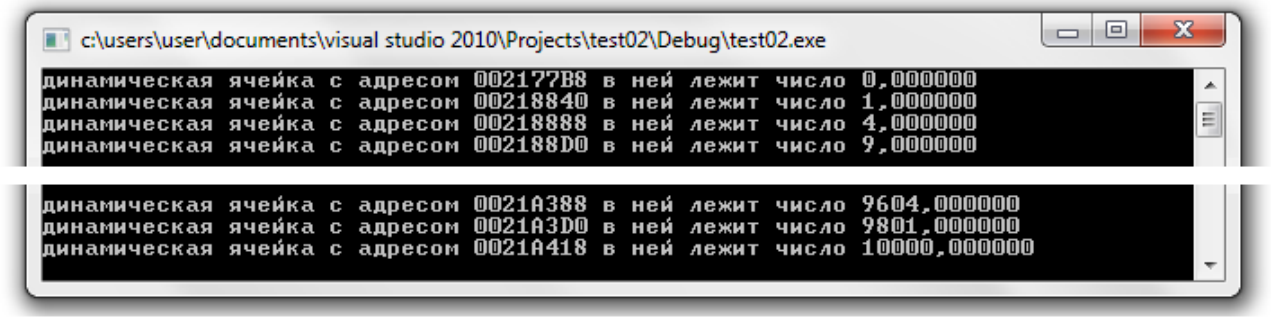

Рис. 16. Результаты выполнения программы

Приведем ниже (*Листинг 32*) исправленный программный код. В нем команда освобождения памяти *delete temp* перенесена в тело цикла *while*. Теперь на каждом шаге цикла динамическая ячейка создается, используется и корректно удаляется.

Листинг 32

```
double *temp; // указатель на ячейку типа double
int i= 0;
while (i<=100)
{     temp= new double(i*i); // Утечки памяти нет
      printf("динамическая ячейка с адресом %p в ней лежит число %f\n",
      temp, *temp);
      delete temp; // освобождение каждого объекта
      i++;
}
```

3.3. Функции динамического распределения памяти

Для динамического распределения памяти используются операторы *new* и *delete*, но кроме этого, существуют специализированные библиотеки подпрограмм (*Таблица 8*). Их прототипы содержатся в файлах *alloc.h* и *stdlib.h*.

Таблица 8. Подпрограммы динамического распределения памяти

Функция	Краткое описание
calloc	*void *calloc(size_t nitems, size_t size);* выделяет память под *nitems* элементов по *size* байт и инициализирует ее нулями
malloc	*void *malloc(size_t size);* выделяет память объемом *size* байт
realloc	*void *realloc (void *block, size_t size);* пытается переразместить ранее выделенный блок памяти, изменив его размер на *size*
free	*void free(void *block);* пытается освободить блок, полученный посредством функции *calloc*, *malloc* или *realloc*

Из описания функций видно, что *malloc( )* и *calloc( )* возвращают *нетипизированный* указатель *void **, следовательно, необходимо выполнять его *явное преобразование* в указатель объявленного типа:

Листинг 33

```
char *str = NULL; // создание указателя и обнуление его
str = (char *)calloc(10, sizeof(char)); // выделение памяти
free (str); // освобождение памяти
```

Во второй строке примера (*Листинг 33*) иллюстрируется, что при выделении памяти происходит последовательно 5 операций:

- функция *calloc( )* выделяет память под блок из 10 ячеек типа *char*;
- обнуляет все 10 символьных ячеек;
- возвращает адрес этого блока – нетипизированный указатель *void**;
- производится принудительная типизация указателя в тип char*;
- полученный типизированный адрес присваивается указателю *str*.

Листинг 34

```
char *str; // создание указателя
if ((str = (char *) malloc(10)) == NULL) // возвращаемый указатель NULL
{       printf("Not enough memory to allocate buffer\n");
        exit(1); } // выход из программы
free(str); // освобождение памяти
```

Если функции *malloc( )* и *calloc( )* по какой-то причине не могут выделить память, то они возвращают пустой указатель *NULL*. На этом знании основан пример, приведенный выше (*Листинг 34*), он иллюстрирует, как можно проверить, успешно ли

выделилась память. Функции *calloc* и *malloc* выделяют блоки памяти. Функция *malloc* выделяет просто заданное *число байт*, тогда как *calloc* выделяет *массив элементов* заданного размера, и инициализирует его нулями.

Листинг 35

```
char *str;
str= (char *)calloc(10,sizeof(char));
strcpy(str,"1234567890");
printf("Строка: {%s} Адрес: %p Размер: %d\n", str, str, strlen(str));
str= (char *)realloc(str, 20); // перевыделение большей памяти
strcpy(str,"12345678901234567890");
printf("Строка: {%s} Адрес: %p Размер: %d\n", str, str, strlen(str));
free (str);
```

```
Строка: {1234567890} Адрес: 006D77B8 Размер: 10
Строка: {12345678901234567890} Адрес: 006D8840 Размер: 20
```

В примере, приведенном выше (*Листинг 35*), сначала функцией *calloc( )* выделяется 10 байт динамической памяти, а потом функция *realloc( )* перераспределяет ранее выделенный блок памяти, изменив его размер на 20 байт.

Также этот пример иллюстрирует возможность работы с *массивом символов* (*блоком, буфером*), память под который выделена динамически. Адрес этого массива хранится в указателе *str*, при выводе на экран со спецификатором *'%p'*, будет выведен на экран адрес первого байта выделенного блока. При выводе со спецификатором *'%s'*, на экран будет выдана вся строка.

В примере применяются специальные подпрограммы для работы с массивами символов – подпрограмма копирования строк *ctrcpy( )* и подпрограмма получения длины строки *strlen( ),* более подробно разговор о строках пойдет в специальной главе (*Гл. 6*).

3.4. Генерация случайных чисел

В задачах различного типа довольно часто встречается необходимость программно получить случайное число. Однако, поскольку компьютерная логика жестко определена, создание случайного числа весьма не простая задача. Для задач обучающего уровня достаточно использования подпрограммы выдачи *псевдо-случайного числа* – функции *rand( ).*

Подпрограмма *rand( )* генерирует псевдослучайное целое число из диапазона от 0 до *RAND_MAX*, это стандартная константа, задаваемая в библиотеке <*stdlib.h*>. Для псевдослучайных чисел, принадлежащих заданному диапазону [*a, b*], можно написать собственный макрос:

```
#define rndm(a, b) (rand()%(b-a))+a
// Генерирует псевдослучайное число между a и b.
```

Если запускать данный макрос несколько раз, то он сгенерирует одни и те же случайные числа (поэтому они и называются «псевдослучайные»). Для более корректной работы необходимо обеспечить генератору *rand( )* каждый раз новые стартовые условия. Для этого используется стартовый генератор *srand(time(NULL))* получающий в качестве аргумента показания *time( )* компьютерных часов. Время постоянно меняется и обеспечивает функцию *srand( )* каждый раз новым аргументом. Для использования функции получения текущего времени *time( )* необходимо подключить библиотеку <*time.h*>.

Теперь обсудим, как задать требуемый диапазон [*a, b*] генерируемого числа. Допустим, у нас имеется некоторое случайное число *X* из неизвестного диапазона, тогда число *X%a* (остаток от деления на *a*) всегда лежит в диапазоне [*0, a*]. Значит число *X%(b-a)* лежит в диапазоне [*0, (b-a)*]. Тогда число *X%(b-a)+a* расположено в диапазоне [*a, b*].

В примере ниже (*Листинг 36*) показывается, как сгенерировать 50 случайных чисел из диапазона от 100 до 200.

Листинг 36

```
#include "stdafx.h"
#include <iostream>
#include <time.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{       srand(time(NULL)); // инициализация генератора случайных чисел
        int a= 100; int b= 200; // диапазон
        for (int i=0; i<50; i++)
        {       cout << rand()%(b-a)+a << " ";     }
        cin.get();
        return 0;
}
```

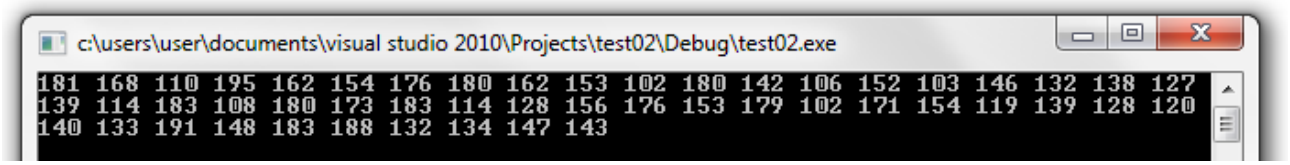

Рис. 17. Результаты выполнения программы

3.5. Лабораторная работа № 3. Указатели.

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Научиться работать со статическими и динамическими переменными. Освоить применение операций адресации и разадресации. Научиться выделять память под динамические переменные и освобождать ее при помощи операторов *new и delete.* Освоить обращение к динамическим объектам. Научиться выделять память под динамические переменные и освобождать ее при помощи функций *calloc, malloc и free*. Научиться визуализировать адреса статических и динамических объектов. Освоить функции *randomize()* и *rand()* генерирующие случайные числа.

Задание на лабораторную работу

1. Задать статическую переменную *X* заданного типа (*Таблица 9*) и присвоить ей случайное значение. Задать переменную-указатель *XPtr* и присвоить ей адрес переменной *X*. Задать указатель *YPtr* и присвоить ему адрес выделенной (оператором *new*) ячейки динамической памяти заданного типа, присвоить ей адрес переменной *X*. Присвоить содержимому этой ячейки случайное значение.
2. Вывести на экран адреса указателей и значения переменных, на которые они указывают. Удалить динамическую переменную.
3. Увеличить на 1 значение указателя *XPtr*, вычислить, на сколько байт изменился адрес, сравнить его с размером *sizeof( )* заданного типа.
4. Задать две динамические переменные *APtr* и *BPtr* при помощи функций *calloc* и *malloc*. Присвоить им случайные значения. Задать нетипизированный указатель *VPtr* и с его помощью поменять местами значения переменных *APtr* и *BPtr*.

5. Вывести на экран адреса указателей и значения переменных, на которые они указывают. Удалить динамические переменные.

Таблица 9. Варианты индивидуальных заданий

№	Тип переменной *X*	Диапазон случайного значения *rand()*	Тип переменных *APtr* и *BPtr*
1.	*double*	$-327.3*10^{34}$ … $435.6*10^{34}$	*int*
2.	*float*	-327.3 … 435.6	*unsigned int*
3.	*long double*	$-1*10^{318}$ … $1*10^{318}$	*long*
4.	*int*	-300 … 300	*double*
5.	*unsigned int*	12 … 1200	*long double*
6.	*long*	-456789 … 987654	*float*
7.	*unsigned long*	1 … 32456789	*double*
8.	*double*	$-567.3*10^{87}$ … $678.5*10^{87}$	*long*
9.	*float*	-234.5 … 543.2	*int*
10.	*long double*	$-5*10^{321}$ … $5*10^{123}$	*unsigned int*
11.	*short int*	-10 … 10	*float*
12.	*unsigned int*	0 … 100	*double*
13.	*long*	-987654 … 123456	*float*
14.	*unsigned long*	1 … 1092837465	*double*
15.	*double*	$-555.5*10^{55}$ … $444.4*10^{44}$	*unsigned int*
16.	*float*	-0.3 … 0.3	*long*
17.	*long double*	$1*10^{-18}$ … $2*10^{-18}$	*int*
18.	*short int*	-100 … 100	*float*
19.	*unsigned int*	1 … 5	*double*
20.	*long*	1 … 10	*float*
21.	*float*	-234.5 … 543.2	*int*
22.	*long double*	10^{321} … 10^{123}	*short int*
23.	*int*	-23 … 235	*float*
24.	*unsigned int*	0 … 450	*double*

Контрольные вопросы:

1. Что такое указатель и для чего он используется?
2. Как узнать значение переменной, если известен только ее адрес?
3. Какие действия выполняет оператор new?
4. Имеет ли статическая переменная адрес?
5. В чем разница между динамической переменной и указателем на нее?
6. Какое значение возвращают функции *malloc* и *calloc* при выделении памяти?
7. Как получить адрес переменной?
8. Какое имя можно задать динамической переменной?
9. Что такое адрес переменной?
10. Какие пять действий происходят при выделении памяти функциями *malloc, calloc*?
11. Какой размер занимает переменная-указатель?
12. Что такое динамическая переменная, чем она отличается от статической?
13. В каком случае нужно пользоваться оператором *delete*, а в каком – функцией *free( )* для освобождения памяти?

4. Подпрограммы

Подпрограмма (процедура, функция) – это именованная часть (блок) программы, к которой можно обращаться из других частей программы столько раз, сколько потребуется. Для того чтобы *вызывать* функцию в разных частях программы ее необходимо *описать* и *задать*. [1 - 8]

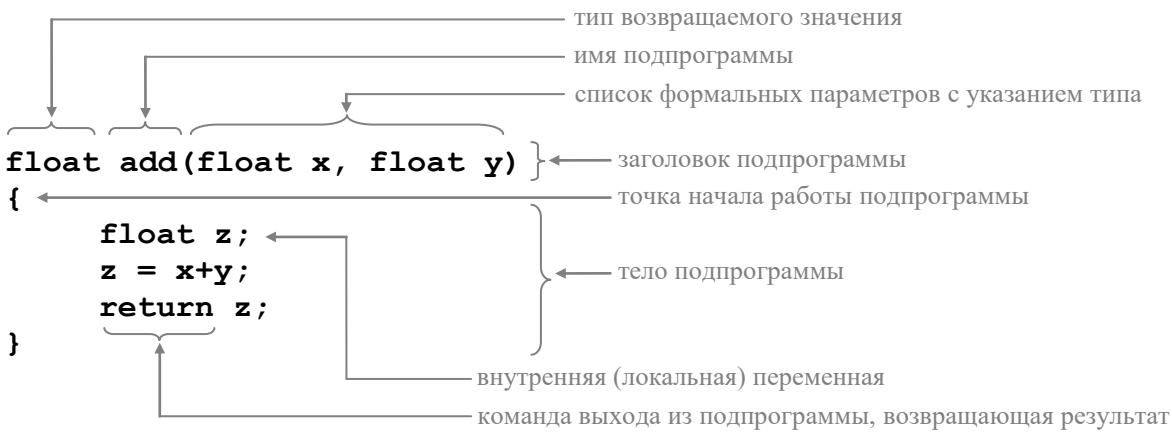

Рис. 18. Структура задания функции (подпрограммы) в языке C++

Описать функцию (описать заголовок, *прототип* функции) – означает определить ее *имя*, *тип_возвращаемого_значения* и *список_формальных_параметров*, записываемых через запятую как в первой строке листинга (см. *Листинг 37*).

Задать функцию – значит, помимо описания заголовка, задать и все команды *тела функции* – строки 7-12 (*Листинг 37*). Если функция не должна возвращать никакого значения, то *тип_возвращаемого_значения* нужно задать как *void*, иначе в теле функции должен присутствовать оператор *return*, возвращающий *значение* соответствующего типа (строка 11 *Листинг 37*).

Листинг 37

```
1  тип_возвращаемого_значения имя(список_формальных_параметров);
2  int _tmain(int argc, _TCHAR* argv[])
3  {     ...
4        x = имя(список_фактических_параметров); // вызов функции
5        ...
6  }
7  // задание функции
8  тип_возвращаемого_значения имя(список_формальных_параметров)
9  {       тело функции
10         ...
11         return возвращаемое_значение;
12 }
```

После того, как функция *имя( )* описана и задана, она может быть вызвана (запущена) из тела главной функции *main( )* или других функций программы. Та точка основной программы, в которой была вызвана функция, называется *точка вызова* (или *точка возврата*) – выполнение основной программы в этой точке приостанавливается (*прерывается*) и выполняется *тело_функции*, по завершению работы функции *возвращаемое_значение* передается в точку возврата, после чего выполнение основной программы будет продолжено. В нашем примере (*Листинг 37*) точка возврата находится в 4 строке после символов "x =".

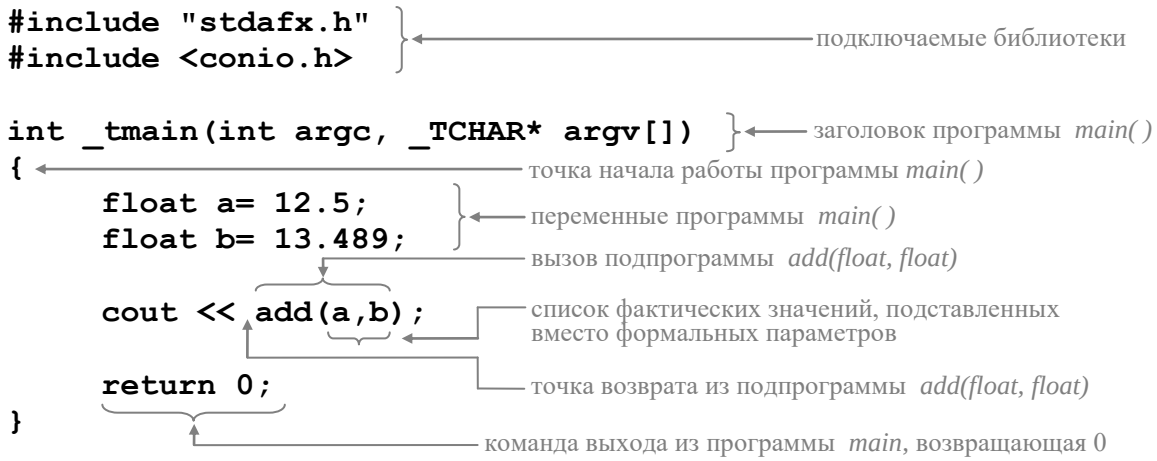

Рис. 19. Структура вызова (использования) функции в главной программе

Важным является следующий момент: при вызове функции вместо *формальных параметров*, данных в описании и задании функции, будут подставлены *фактические значения* этих параметров, и выполнение тела функции будет производиться над фактическими значениями переменных. Естественно, типы переменных из списков формальных и фактических параметров должны совпадать. Собственно, имена переменных в списке формальных параметров не важны, их можно не указывать – как в примере ниже.

В некоторых случаях удобно выполнить описание функции (прототип) в одном месте, а задать ее (прописать реализацию конкретных действий, выполняемых подпрограммой) – в другом.

Так устроены заголовочные файлы подключаемых библиотек – файлы с расширением **.h: "stdafx.h", <iostream.h>, <conio.h>* и др. В них содержатся только имена глобальных констант и заголовки (прототипы) подпрограмм, реализация функций в откомпилированном виде располагается в другом месте.

Рассмотрим программу, печатающую степени числа 2:

Листинг 38 (часть 1)

```
float pow(float, int); // прототип функции, которая задана ниже

void _tmain(int argc, _TCHAR* argv[]) // главная функция main()
{       for (int i=0; i<10; i++)
                cout << pow(2,i) << "\n"; // вызов функции pow(float, int)
}
```

Первая строка функции – ее *прототип* (описание), указывающее, что *pow* – функция, получающая параметры типа *float* и *int* и возвращающая *float*. Прототип функции используется для того, чтобы сделать определенными обращения к функции в других местах.

При вызове функции тип каждого параметра сопоставляется с ожидаемым типом. Это гарантирует надлежащую проверку и преобразование типов. Например, обращение *pow(12.3,"abcd")* вызовет недовольство компилятора, поскольку *"abcd"* является строкой, а не *int*. При вызове *pow(2,i)* компилятор производит неявное преобразование 2 к типу float, как того требует функция.

Функция *pow* может быть определена так:

Листинг 38 (часть 2)

```
float pow(float x, int n) // задание функции pow()
{     if (n < 0)
            error("\nsorry, negative exponent to pow()\n");
```

```
            // извините, отрицательный показатель для pow()
      switch (n)
      {     case 0: return 1; // выход из рекурсии
            case 1: return x;
            default: return x*pow(x,n-1); // рекурсия
      }
}
```

Первая часть определения функции задает имя функции, тип возвращаемого ею значения (если таковое имеется) и типы и имена ее параметров (если они есть). Значение возвращается из функции с помощью оператора *return*.

Если функция не возвращает значения, то ее следует описать как *void*:

Листинг 39

```
void swap(int* p, int* q) // функция не возвращающая никакого значения
{     int t = *p; // поменять местами переменные p и q
      *p = *q;
      *q = t;
}
```

Оператор *return* служит для немедленного выхода из тела функции, при этом по необходимости указывается возвращаемое значение.

Листинг 40

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

void outto5(int n)
{       for (int i=0; i<n; i++)
                if (i==5) return; // даже если n>5, выводиться будут числа
<=5
                else cout << i << " ";
}

int sumab(int a, int b)
{       return a+b;
}

int _tmain(int argc, _TCHAR* argv[])
{       outto5(10);
        cout<< "сумма 3 и 4 = " << sumab(3,4) << endl;
        return 0;
        cout << "...не важно, что..."; // эта часть кода никогда не
выполнится
}
```

4.1. Передача параметров в тело функции

В языке C++ существует три способа передачи параметров в тело функции: *«по значению»*, *«по ссылке»* и *«по указателю»*.

```
void F1(int p, int q); // передача параметров по значению
void F2(int &p, int &q) // передача параметров по ссылке
void F3(int *p, int *q) // передача параметров по указателю
```

Передача параметров по значению

Для того чтобы понять их специфику, рассмотрим процесс передачи данных в функцию более подробно на следующем примере (*Рис. 20*). Вызывающая функция *main( )* имеет 3 переменных целого типа – *x*, *y* и *z*. Под каждую из них выделено по одной ячейке памяти размером *sizeof(int)*. В точке вызова функции *F( )*

приостанавливается выполнение функции *main( )*, в памяти компьютера выделяется еще 3 ячейки под переменные *p*, *q* и *r*.

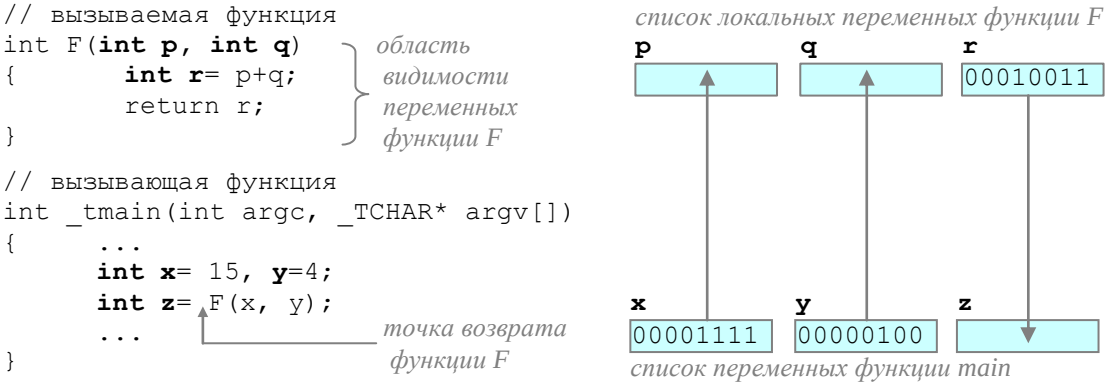

Рис. 20. Передача параметров по значению

Область видимости этих переменных *p*, *q* и *r* – та зона программного кода, где они могут быть использованы – ограничена телом функции *F( )*. Такие переменные называются *локальными*. В ячейки *p* и *q* копируются значения *глобальных* переменных *x* и *y* соответственно, после чего производится вычисление переменной *r* и оператор *return* возвращает значение *r* в точку вызова. Память, выделенная под локальные переменные *p*, *q* и *r* освобождается. После этого возобновляется выполнение функции *main( )* – переменной *z* присваивается возвращаемое значение.

В рассмотренном выше примере данные в подпрограмму *F( )* передаются *«по значению»*, это значит, что в программе *F( )* создаются локальные переменные *p* и *q* и в них *копируются значения* фактических параметров – переменных *x* и *y*. Локальные переменные *p* и *q* в теле основной программы не видны, и не могут использоваться. И, естественно, если в теле функции *F( )* переменные *p* и *q* меняют свои значения, это никак не скажется на переменных *x* и *y* – они не изменятся.

Рассмотрим пример, приведенный ниже (*Листинг 41*). В нем приводится подпрограмма *int sum2ab(int a, int b)*, в которую передаются *по значению* два целых числа – *a* и *b*, в теле подпрограммы значения их удваиваются, затем производится вывод на экран значений этих переменных и их адресов, после чего они суммируются, и в вызывающую программу возвращается сумма этих значений.

В основной программе создаются две глобальные переменные с такими же именами – *a* и *b*, задаются случайными числами, выводятся на экран вместе со своими адресами, потом производится вызов подпрограммы *sum2ab* и вновь значения *a* и *b* выводятся на экран. Результаты работы программы (*Листинг 41*) приведены ниже на рисунке (*Рис. 21*).

Листинг 41

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

int sum2ab(int a, int b) // Передача параметров в функцию «по значению»
{     a= 2*a; // удваиваем значения переменных
      b= 2*b;
      printf("\nвывод из подпрограммы sum2ab:\n");
      printf("a= %2d (address a= %p)  b= %2d (address b= %p)\n",a,&a,b,&b);
      return a+b; // возвращаем 2a+2b
}

void _tmain(int argc, _TCHAR* argv[])
```

```
{       setlocale(LC_ALL, "Russian"); // подпрограмма русского шрифта
        srand(time(NULL)); // инициализация генератора случайных чисел

        int a= rand()%10;
        int b= rand()%10;

        printf("\nвывод из главной подпрограммы main():\n");
        printf("a= %2d (address a= %p)  b= %2d (address b= %p)\n",a,&a,b,&b);

        printf("\n2a+2b= %d\n", sum2ab(a,b));

        // значения глобальных переменных a и b не изменились
        printf("\nповторный вывод из главной подпрограммы main():\n");
        printf("a= %2d (address a= %p)  b= %2d (address b= %p)\n",a,&a,b,&b);

          cin.get(); // ожидание нажатия клавиши
}
```

```
вывод из главной подпрограммы main():
a=  3 (address a= 0032FB80)  b=  1 (address b= 0032FB74)

вывод из подпрограммы sum2ab:
a=  6 (address a= 0032FA9C)  b=  2 (address b= 0032FAA0)

2a+2b= 8

повторный вывод из главной подпрограммы main():
a=  3 (address a= 0032FB80)  b=  1 (address b= 0032FB74)
```

Рис. 21. Результаты выполнения программы

Проанализируем результаты. Можно видеть, что сначала выводятся данные из основной программы, *глобальные* переменные *a* и *b* принимают значения 3 и 1 соответственно. В круглых скобках выведены адреса статических переменных *a* и *b*.

Затем производится вывод *локальных* переменных *a* и *b* из подпрограммы *sum2ab*. Можно видеть, что их адреса не совпадают с адресами глобальных переменных *a* и *b*, следовательно, это другие ячейки памяти. Значения этих переменных удвоены. После этого выводится сумма удвоенных значений – подпрограмма *sum2ab* корректно вычисляет и передает в главную программу результат сложения.

Однако заметьте, что повторный вывод глобальных переменных *a* и *b* (а это именно те исходные переменные – можно видеть по адресам) показывает, что их значения *не изменились*!

Передача параметров по ссылке

Передачу параметров по ссылке удобно рассмотреть на примере функции *S( )* меняющей местами свои аргументы *p* и *q* (*Рис. 22*). Если бы *p* и *q* передавались в функцию *S( ) по значению* – как в предыдущем примере, то обмен местами значений локальных переменных *p* и *q* в теле функции *S( )* никак не повлиял бы на значения глобальных переменных *x* и *y*. Действительно, ведь функция *S( )* оперирует лишь с копиями переменных *x* и *y*, и на момент возврата управления из функции *S( )* в функцию *main( )* локальные переменные *p* и *q* уже уничтожены.

При передаче данных в функцию *S( )* по ссылке ячейки памяти для ссылочных переменных *p* и *q* *не создаются*, а для работы используются ячейки памяти вызывающей программы – чьи *адреса* передаются в функцию *S( )*. Символ "&" перед переменным в списке формальных параметров в данном случае обозначает операцию «*обращение по ссылке*».

В примере (*Рис. 22*) переменные *p* и *q* в теле функции *S( )* не существуют как самостоятельные отдельные ячейки памяти, а служат лишь для обозначения ячеек x и y, вернее – значений тех ячеек памяти, чьи адреса *&p* и *&q* заданы в списке фактических параметров при вызове функции *S( )*.

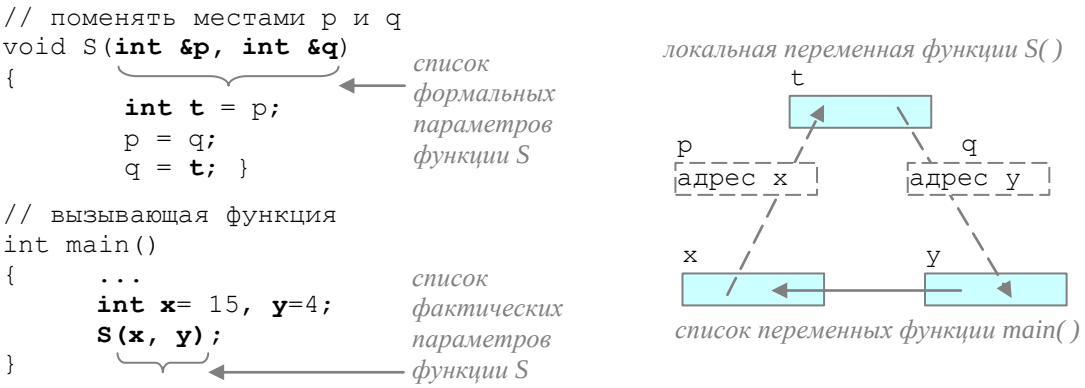

Рис. 22. Передача параметров по ссылке

Изменим пример, приведенный выше (*Листинг 41*) так, чтобы параметры передавались в функцию *по ссылке*. (*Листинг 42*). Для этого в заголовке функции *int sum2ab(int& a, int& b)*, изменим строку описания формальных параметров, добавив символы «**&**». Больше ничего изменять не станем – как и в предыдущем примере в теле подпрограммы значения *a* и *b* удваиваются, затем производится вывод на экран значений этих переменных с адресами, после чего они суммируются, и в вызывающую программу возвращается сумма этих значений.

Листинг 42

```
int sum2ab(int& a, int& b) // Передача параметров в функцию «по ссылке»
{       a= 2*a; // удваиваем значения переменных
        b= 2*b;
        printf("\nвывод из подпрограммы sum2ab:\n");
        printf("a= %2d (address a= %p)  b= %2d (address b=
%p)\n",a,&a,b,&b);
        return a+b; // возвращаем 2a+2b
}
```

Основную программу также оставим без изменений. Посмотрим, как изменился результат (*Рис. 23*). Во-первых, можно видеть, что адреса переменных *a* и *b*, выводимые из основной программы и из подпрограммы *sum2ab* одни и те же – мы работаем с одними и теми же ячейками памяти. И значения глобальных переменных как удвоились в подпрограмме, так и остались удвоенными при повторном выводе из программы *main( )*.

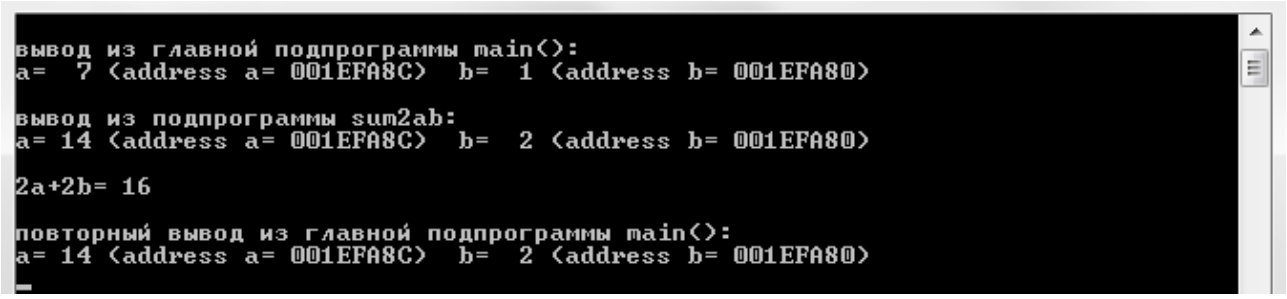

Рис. 23. Результаты выполнения программы

Передача параметров по указателю

Передача параметров по указателю (*Рис. 24*) практически ничем не отличается от передачи данных по значению – разница лишь в том, что аргументом функции выступают не статические переменные *S(int p, int q)*, а *переменные-указатели S(int *p, int *q)*.

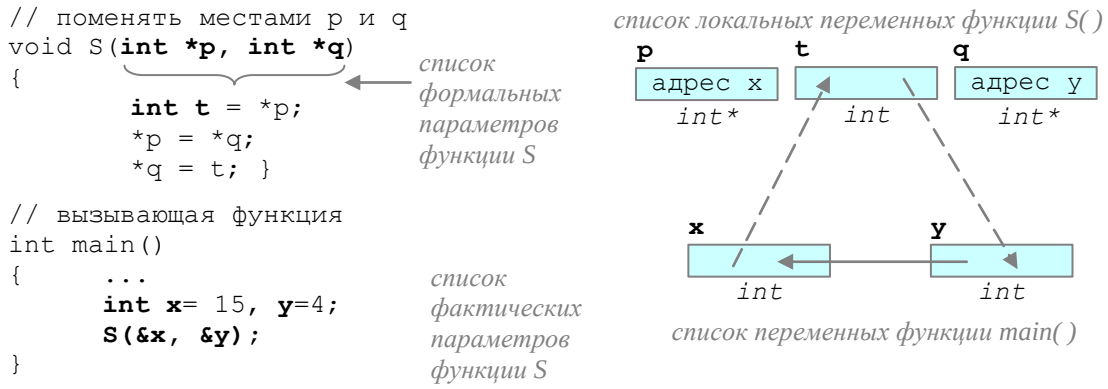

Рис. 24. Передача параметров по указателю

В точке вызова функции *S( )* приостанавливается выполнение функции *main( )*, в памяти компьютера выделяется еще 3 ячейки под переменные *p, q* каждая размером *sizeof(int*)* и ячейка *t* размером *sizeof(int)*. Поскольку *p* и *q* – указатели, то в них копируются не значения глобальных переменных *x* и *y*, а их адреса – при вызове функции необходимо это указать – *S(&x, &y)*. После чего производится обмен значениями между переменной *t* и ячейками с адресами, хранящимися в переменных *p* и *q*, а в них хранятся адреса глобальных переменных *x* и *y*. По выходу из функции *S( )*, память, выделенная под локальные переменные *p, q* и *t* освобождается, управление передается в функцию *main( )*.

Пример, приведенный ниже (*Листинг 43*) рассмотрим более подробно. В этом примере реализуется передача параметров *«по указателю»*. Таким образом, формальные параметры функции *void set_ab(int* a, int* b)* – это указатели, т.е. ячейки для хранения адресов. Поэтому, в основной программе *main( )* когда мы вызываем подпрограмму *set_ab(&a, &b)*, мы передаем в нее адреса глобальных переменных *a* и *b*.

В теле подпрограммы *set_ab* мы присваиваем глобальным переменным новые значения 100 и 200 соответственно. Естественно, мы изменяем не значения локальных переменных *a* и *b*, ведь они хранят адреса, а *содержимое* этих указателей.

Листинг 43

```
void set_ab(int* a, int* b) // Передача параметров в функцию «по указателю»
{ *a= 100; // содержимому указателя a присваиваем 100
  *b= 200; // содержимому указателя b присваиваем 200
 printf("вывод из подпрограммы set_ab:\n");
 printf("содержимое a=%3d указатель a=%p адрес указателя a=%p",*a,a,&a);
 printf("содержимое b=%3d указатель b=%p адрес указателя b=%p",*b,b,&b);
}

void _tmain(int argc, _TCHAR* argv[])
{     setlocale(LC_ALL, "Russian");
      srand(time(NULL)); // инициализация генератора случайных чисел

      int a= rand()%100;
      int b= rand()%100;
      printf("вывод из главной подпрограммы main():\n");
      printf("a= %3d (address a= %p)  b= %3d (address b= %p)\n",a,&a,b,&b);

      set_ab( &a, &b ); // передаем в подпрограмму адреса переменных

      printf("повторный вывод из главной подпрограммы main():\n");
      printf("a= %3d (address a= %p)  b= %3d (address b= %p)\n",a,&a,b,&b);
      cin.get();
}
```

Нужно понимать, что сами по себе переменные-указатели *a* и *b* в подпрограмме *set_ab* являются локальными переменными и имеют свои адреса (адреса адресных ячеек). На рисунке ниже (*Рис. 25*) приведены результаты работы программы, по которым можно видеть, что *содержимое* локальных переменных *a* и *b* равно 100 и 200 соответственно, их *значениями* являются адреса глобальных переменных 001DF9E0 и 001DF9D4 соответственно, и, кроме того, у них самих есть *адреса* 001DF8FC и 001DF900.

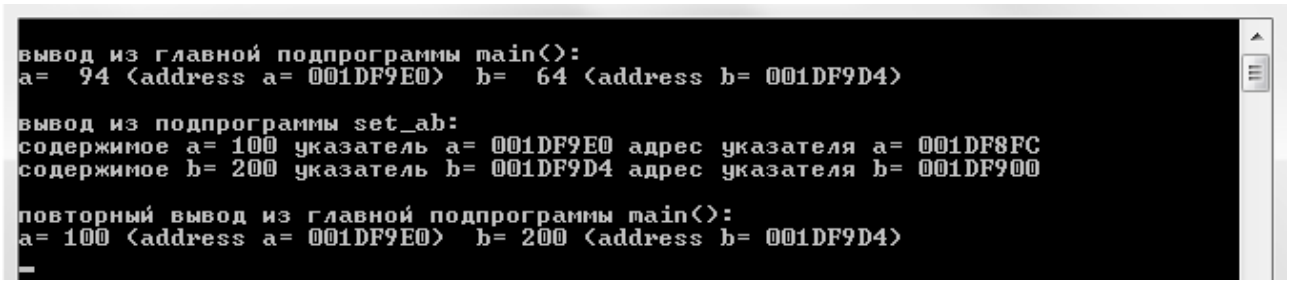

Рис. 25. Результаты выполнения программы

В результате выполнения программы глобальные переменные *a* и *b* изменили свое значение внутри подпрограммы *set_ab*.

Таблица 10. Сравнение способов работы с параметрами подпрограмм

Реализация подпрограммы	Вызов подпрограммы
*«**по значению**» данные передаются только из головной программы в подпрограмму*	

`void f(int a)`
`{ // a - локальная;`
`// a== 0;`
`a= 123;`
`// a== 123;`
`}`
	`void _tmain(int argc, _TCHAR* argv[])`
	`{ int b= 0; // b - глобальная`
	`// b== 0;`
	`f(b);`
	`// b== 0;`
	`}`

*команда **return** служит для возвращения значения из подпрограммы в головную программу*	

`int f() // нет параметров`
`{`
`return 123;`
`// возвращаемое значение`
`}`
	`void _tmain(int argc, _TCHAR* argv[])`
	`{ int b= 0; // b - глобальная`
	`// b== 0;`
	`b= f();`
	`// b== 123;`
	`}`

*«**по ссылке**» можно передавать данные в подпрограмму из головной программы, а можно наоборот – возвращать данные из подпрограммы в головную программу*	

`void f(int &a)`
`{ // a - ссылка на глобальную b`
`// a== 0;`
`a= 123;`
`// a== 123;`
`}`
	`void _tmain(int argc, _TCHAR* argv[])`
	`{ int b= 0; // b - глобальная`
	`// b== 0;`
	`f(b);`
	`// b== 123;`
	`}`

*«**по указателю**» можно передавать данные в подпрограмму из головной программы, а можно наоборот – возвращать данные из подпрограммы в головную программу*	

`void f(int *a)`
`{//a - указатель на глобальную b`
`// a- адрес глобальной b;`
`*a= 123; // содержимое`
`// *a== *b== 123;`
`}`
	`void _tmain(int argc, _TCHAR* argv[])`
	`{ int b= 0; // b - глобальная`
	`// передача адреса b`
	`f(&b);`
	`// b== 123;`
	`}`

4.2. Перегрузка функций

Разные функции, обычно имеют разные имена, но функциям, выполняющим сходные действия над объектами различных типов, иногда лучше дать возможность иметь одинаковые имена. Если типы их параметров различны, то компилятор всегда может различить их и выбрать для вызова нужную функцию.

Например, в стандартной библиотеке <*math.h*> имеется шесть функций возведения в степень:

```
double pow(double, double);
double pow(double, int);
float pow(float, float);
float pow(float, int);
long double pow(long double, long double);
long double pow(long double, int);
```

Листинг 44 иллюстрирует перезагрузку различных функций, отличающихся списком параметров. По результатам работы программы (*Рис. 26*) можно судить о том, какие из функций вызывались и почему:

Листинг 44

```
void test(int X)
{       printf("вызов функции с целым аргументом X= %d\n", X); }

void test(float X)
{       printf("вызов функции с вещественным аргументом X= %f\n", X); }

void test(float *X)
{       printf("вызов функции с вещественнозначным указателем X= %p\n", X);
}

void test(int X, float Y)
{       printf("вызов функции с двумя аргументами X= %d Y= %f\n", X, Y); }

void test(double X)
{       printf("вызов функции с double аргументом X= %f\n", X); }

void _tmain(int argc, _TCHAR* argv[])
{       setlocale(LC_ALL, "Russian"); // подпрограмма русского шрифта

        int a=3;                test(a);
        float b= 3.234;         test(b);
        double d= 0.001;        test(d);
                                test(&b);
                                test(a, b);
                                test(3.5657675);

        cin.get(); // ожидание нажатия клавиши
}
```

```
вызов функции с целым аргументом X= 3
вызов функции с вещественным аргументом X= 3,234000
вызов функции с double аргументом X= 0,001000
вызов функции с вещественнозначным указателем X= 0038FD48
вызов функции с двумя аргументами X= 3 Y= 3,234000
вызов функции с double аргументом X= 3,565768
```

Рис. 26. Результаты выполнения программы

К понятию перегрузки функций языка C++ относится и метод маскировки (или маскирования) – это способ перегрузки функции (или метода и оператора класса) при котором полностью все описание новой функции совпадает с описанием функции описанной ранее, совпадают в том числе имена и списки параметров. В этом случае компилятор не может найти различий в вызове этих двух функций и считает актуальным последнее задание функции. Говорят, что одна функция маскирует другую.

Например, можно следующим вызовом (*Листинг 45*) замаскировать стандартную функцию библиотеки <*math.h*> возведения в степень *double pow(double, double):*

Листинг 45

```
double pow(double X, double Y);
{
        return X+Y; // измененное тело функции
}
```

Теперь, если мы вызовем функцию *pow(4, 2),* то она вернет результат не 16.0, а сумму введенных значений аргументов – число 6.0.

4.3. Функции библиотеки <math.h>

В состав стандартного пакета *Visual C++* входит огромное количество математических функций. Прототипы математических функций содержатся в файле <*math.h*> (*Таблица 11*), за исключением прототипов *_clear87, _control87, _fpreset, status87,* определенных в файле <*float.h*>.

Функции библиотеки <*float.h*> реализуются арифметическим сопроцессором, который как раз и был предназначен для выполнения операций с числами в формате с плавающей точкой (вещественные числа) и длинными целыми числами или его 64-битным аналогом. Арифметический сопроцессор (*FPU, Floating Point Unit*) значительно ускоряет вычисления, связанные с вещественными числами. Он используется при вычислениях значений таких функций, как синус, косинус, тангенс, логарифмы и т.д.

Таблица 11. Математические функции

Функция	Описание
abs	абсолютное значение (модуль) целого числа
acos, acosl	арккосинус
asin, asinl	арксинус
atan, atanl	арктангенс x
atan2, atan2l	арктангенс x/y
cabs, cabsl	абсолютное значение комплексного числа
ceil, ceill	округление вверх, наименьшее целое, не меньшее x
cos, cosl	косинус
cosh, coshl	косинус гиперболический
exp, expl	экспонента
fabs, fabs	абсолютное значение вещественного числа (double)
floor, floorl	округление вниз, наибольшее целое, не большее x
fmod, fmodl	остаток от деления, аналог операции %
frexp, frexpl	разделяет число на мантиссу и степень
hypot, hypotl	гипотенуза
labs	абсолютное значение длинного целого (long)
ldexp, ldexpl	произведение числа на два в степени e, вычисление $x*2^e$
log, logl	логарифм натуральный

Функция	Описание
log10, log10l	логарифм десятичный
modf, modfl	разделяет на целую и на дробную часть
poly, polyl	полином
pow, powl	степень
pow10, pow10l	степень десяти
sin, sinl	синус
sinh, sinhl	синус гиперболический
sqrt, sqrtl	квадратный корень
tan, tanl	тангенс
tanh, tanhl	тангенс гиперболический
matherr	управление реакцией на ошибки при выполнении функций математической библиотеки
_clear87	получение значения и инициализация состояния сопроцессора
_control87	получение старого значения слова состояния для функций арифметики с плавающей точкой и установка нового состояния
_status87	получение значения слова состояния с плавающей точкой

Вещественные функции, как правило, работают с двойной точностью (тип *double*). Многие функции имеют версии, работающие с учетверенной точностью (тип *long double*). Имена таких функций имеют суффикс *"l"* в конце (*atan* и *atanl, fmod* и *fmodl* и т. д.). Действие модификатора *long* в применении к *double* зависит от архитектуры ЭВМ.

В библиотеке определен также ряд констант, таких как *M_PI* (число π), *M_E* (основание натурального логарифма *e*) и др.

Функция *matherr*, которую пользователь может определить в своей программе, вызывается любой библиотечной математической функцией при возникновении ошибки. Эта функция определена в библиотеке, но может быть переопределена для установки различных *пользовательских* процедур обработки ошибок.

4.4. Отладка программ. Трассировка программного кода. Окно watch

```
Настоящие программисты не
исправляют чужих ошибок -
они убивают их собственными.
```

В нижней части рабочего окна *Visual C++* имеется несколько вкладок, отражающих процесс выполнения программы. На *Рис. 6* можно видеть окно вывода «*Output*», в котором отражается процесс компиляции и выполнения программы, а так же наличие в ней ошибок.

Для просмотра информации об обнаруженных компилятором ошибках необходимо перейти во вкладку «*Error List*» (*Рис. 7*). В этом окне выводятся сообщения трех видов: ошибки (*error*), предупреждения (*warning*) и сообщения (*messages*). По каждой ошибке указывается имя файла и номер строки, в которой она обнаружена, код ошибки и текстовое объяснение этой ошибки.

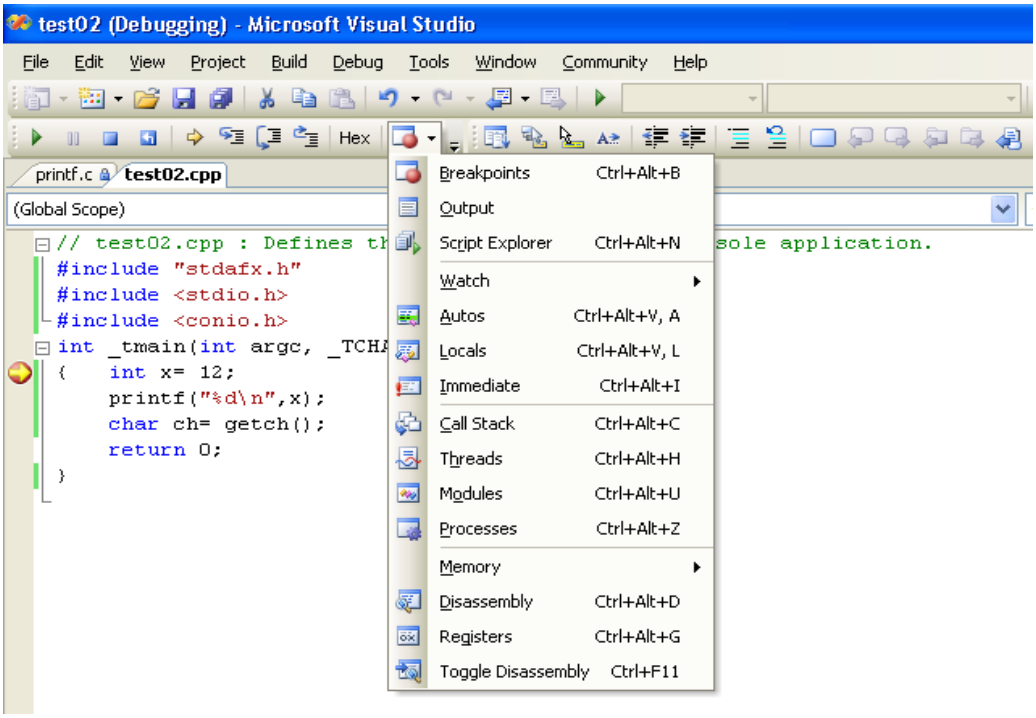

Рис. 27. Пошаговая трассировка программного кода. Точка останова

Для обнаружения и исправления ошибок (*debugging*) в коде программы используется такой инструмент как пошаговая трассировка (*trace*) – это просмотр выполнения программы по шагам – от оператора к оператору.

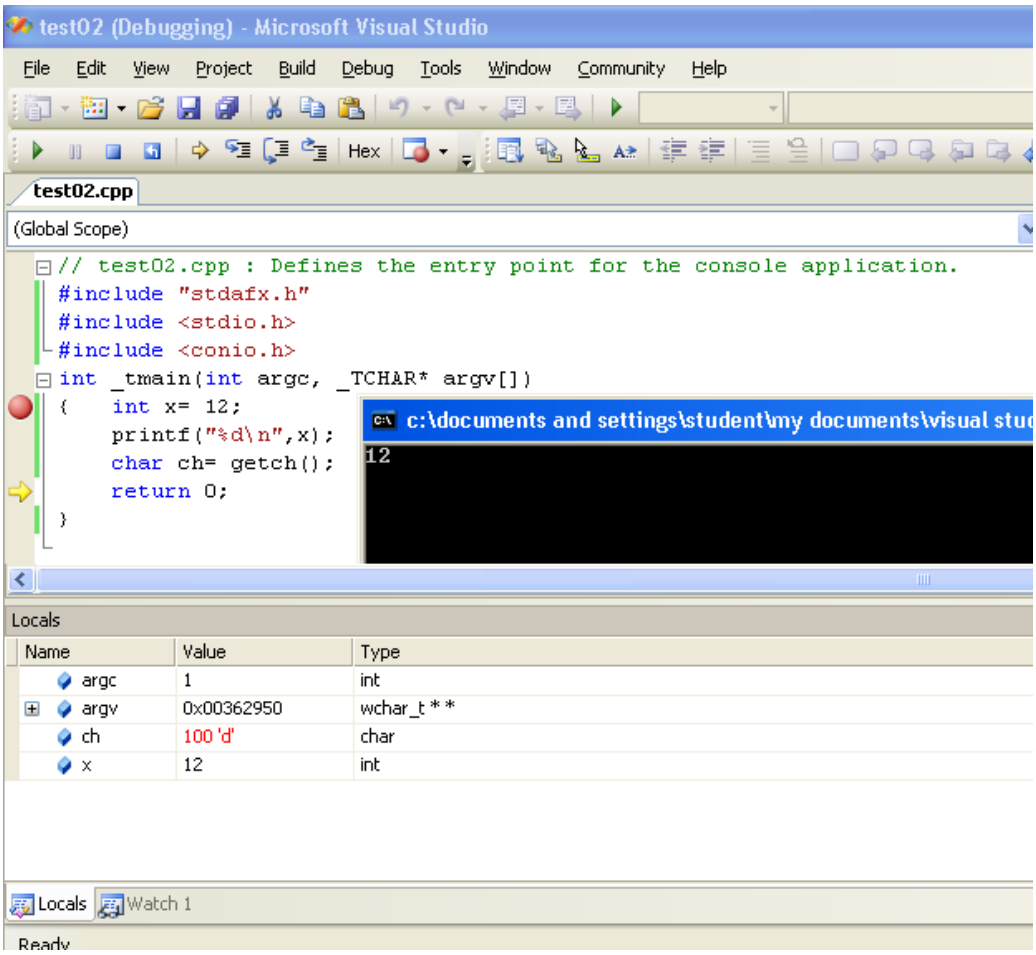

Рис. 28. Пошаговая трассировка. Просмотр значений переменных

Все команды, необходимые для пошаговой трассировки сгруппированы в основном меню *Visual C++* в разделе «*Debug*» - отладка.

Базовым инструментом отладки и трассировки является понятие *точка останова* – «*Breakpoint*» – это место программного кода, на котором выполнение программы будет приостановлено. Расставив в нужных местах программы точки останова (на *Рис. 27* точка останова отображена красным кружком) можно пошагово проследить выполнение операторов программы при помощи команд *StepInto* <*F11*>, *StepOver* <*F10*> и *StepOut* <*Shift+F11*>: на *Рис. 27* выполняемый в данный момент оператор обозначен желтой стрелкой.

Для просмотра текущих значений переменных в процессе отладки используется окно *watch* <*Ctrl+Alt+Q*> со вкладками *Watch* и *Locals* – в нижней части окна (*Рис. 28*). В этих вкладках указываются имена, типы и текущие значения переменных.

На рисунке (*Рис. 28*) приведен процесс пошаговой трассировки с визуализацией текущих значений переменных. Красным кружком обозначена точка останова. Желтая стрелка указывает, какой оператор будет выполняться следующим. Красным цветом в окне просмотра значений выделена переменная (*ch*), значение которой изменено на предыдущем шаге трассировки.

4.5. Лабораторная работа № 4. Подпрограммы.

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цели работы

Научиться создавать подпрограммы и вызывать их из тела основной функции. Освоить применение оператора *return*, возвращающего значение функции. Научиться передавать параметры в функцию и корректно возвращать значения – результаты вычислений из функции. Освоить передачу параметров в функцию *по значению*, *по ссылке* и *по указателю*. Научиться создавать *прототипы функций*, освоить *перегрузку функций*. Изучить функции библиотеки *math.h*. Научиться пользоваться средствами пошаговой трассировки кода и просмотра текущих значений переменных.

Задание на лабораторную работу

1. Написать подпрограмму в соответствии со своим вариантом индивидуального задания. Реализовать вызов функции из главной программы и вывод результатов вычисления на экран. Отладить и протестировать программу. В процессе отладки пользоваться средствами пошаговой трассировки кода и окнами просмотра текущих значений переменных (*Watch* и *Locals*).
2. Написать прототип функции. Передачу параметров в функцию реализовать тремя изученными методами: *по значению, по ссылке* и *по указателю*. Вывести в окно *Watch* адреса указателей, ссылок и значений переменных, используемых для обращения к параметрам функции.
3. Продемонстрировать в работе использование оператора *return*, использование прототипа функции и перегрузку функций.
4. В отчете отразить процесс трассировки кода, результаты работы программы, листинг программного кода с комментариями, блок-схему программы.

Варианты индивидуальных заданий

1. Простое число делится нацело только на 1 и на само себя. Напишите функцию bool f(int n), которая возвращает true, если n – простое число и false в противном случае. Составьте таблицу и подсчитайте количество простых чисел от 2 до 1000.
2. Напишите функцию int f(int M, int N), которая вычисляет и возвращает сумму всех нечетных целых чисел в пределах от M до N включительно. Используйте оператор for.

3. Напишите функцию int f(int N, int n), которая вычисляет и возвращает наименьшее из чисел, больших или равных N, которое делится нацело на n. Используйте оператор while.

4. Определите функцию double f(double x, double y), которая вычисляет и возвращает длину гипотенузы прямоугольного треугольника, две другие стороны x и y которого известны.

5. Напишите функцию double f(double x1, double y1, double x2, double y2), которая вычисляет расстояние между двумя точками (xl, yl) и (x2, y2).

6. Напишите функцию f(double& a, double& b, double c, double q), которая возвращает катеты прямоугольного треугольника, гипотенуза которого равна c, а острый угол q (градусов).

7. Напишите функцию int fact(int m), которая вычисляет и возвращает факториал положительного числа m, т.е. произведение всех натуральных чисел, меньших или равных m.

8. Напишите функцию f(int& m4, int& m3, int& m2, int& m1, int& m0, int N), которая возвращает все цифры пятизначного натурального числа N.

9. Напишите функцию f(int& h, int& m, int& s, int sec), которая принимает количество секунд, прошедших с начала дня, и возвращает три целых переменных (часы, минуты и секунды).

10. Напишите функцию int f(int x), которая получает целое значение (не больше пяти знаков) и возвращает число с обратным порядком цифр. Например, принимается число 7631, возвращается число 1367.

11. Напишите функцию f(double* a, double* b, double c, double q), которая возвращает катеты прямоугольного треугольника, гипотенуза которого равна c, а острый угол q (градусов).

12. Напишите функцию swap(int& a, int& b, int& c), которая изменяет значения параметров по правилу $a \to b \to c \to a$.

13. Напишите функцию int f(int& a, int& b, int x, int y), которая присваивает объекту a значение, равное x*y, а объекту b значение x/y. Если значение y было равно нулю, функция должна возвратить 0, иначе 1.

14. Напишите функцию int f(int x, int y), которая возвращает 0, если значения x и y оба равны нулю, 12/x, если y равен 0, 12/y, если x равен 0, иначе 144/(x*y).

15. Напишите функцию double f(double x, double y, double z), которая возвращает m*n/k, где k есть меньшее из чисел x, y, z, а m и n есть среднее и большее из этих чисел. Предполагается, что значения параметров больше нуля.

16. Напишите функцию double f(double x, double y, double z), которая возвращает наибольшее из значений модулей попарных разностей: |x-y|, |y-z|, |z-x|.

17. Напишите функцию bool f(int x, int y, int z), которая возвращает true, если $x^2 + y^2 = z^2$, иначе возвращает false.

18. Наибольший общий делитель (НОД) двух целых чисел – это наибольшее целое, на которое без остатка делится на каждое из двух чисел. Напишите функцию int NOD (int m, int n), которая возвращает НОД двух целых чисел m и n.

19. Наименьшее общее кратное (НОК) двух целых чисел – это наименьшее целое, которое без остатка делится на каждое из двух чисел. Напишите функцию int NOK(int m, int n), которая возвращает НОК двух целых чисел m и n.

20. Говорят, что целое число является совершенным числом, если его сомножители, включая 1 (но не само число) в сумме дают это число. Например, 6 – совершенное число, так как 6=1+2+3. Напишите функцию bool f(int n), которая определяет, является

ли ее параметр n совершенным числом. Используйте эту функцию в программе, которая определяет и печатает все совершенные числа в диапазоне от 1 до 1000.

21. Напишите функцию bool f(int x, int y), которая возвращает true, если x делится нацело на y, или наоборот, y делится нацело на x, иначе возвращает false. Предполагается, что значения параметров больше нуля.

22. Положительные числа x, y, z могут быть сторонами треугольника, если большее из них меньше суммы двух других. Напишите функцию bool f(int x, int y, int z), которая возвращает true, если числа x, y, z могут быть сторонами треугольника, иначе возвращает false.

23. Напишите функцию int f(int h, int m, int s), которая принимает три целых аргумента (часы h, минуты m и секунды s) и возвращает количество секунд, прошедших с начала дня.

24. Напишите функцию int f(int m, int d), которая принимает два целых аргумента (месяц m и день d) и возвращает количество дней, прошедших с начала года. Считаем, что год не високосный.

Контрольные вопросы:

1. Как происходит вызов подпрограммы и возвращение из нее в языке C++?

2. Почему нельзя изменить переменные вызывающей функции, передаваемые в вызываемую подпрограмму по значению?

3. Что такое перегрузка функций? Составьте пример перегруженной функции.

4. Как реализована передача значений переменных вызывающей функции в тело вызываемой подпрограммы по указателю? Изменяется ли значение указателя при этом?

5. Что такое точка возврата из функции, где она находится?

6. Как передать фактические параметры в подпрограмму по ссылке? Изменятся ли их значения в вызывающей программе?

7. Для чего нужна команда *return*? Какой тип данных должна иметь подпрограмма, не возвращающая никакого значения?

8. В чем отличие использования символа & в вызове подпрограммы и в ее описании?

5. Массивы

```
Программисты, рассматривают
фотографию девушки:
 - Она у тебя первая?
 - Не, нулевая.
```

Элементы *массива* в языке *C++* всегда нумеруются не с единицы, а *с нуля*. Тип данных *масси*в - это набор данных *одного и того же типа*, собранных под одним именем. *Элемент массива* определяется именем массива и порядковым номером (*индексом*). [5, 6] Основная форма объявления массива следующая:

```
тип имя_массива[размер1]; // одномерный массив == вектор
тип имя_массива[размер1][размер2]; // двумерный массив == матрица
тип имя_массива[размер1][размер2]...[размер N]; // N-мерный массив
```

Имя массива – это указатель, содержащий адрес, начиная с которого хранится массив. Здесь «*тип*» - это *базовый тип* элементов массива, «*размер*» - количество элементов одномерного массива. Двумерный массив – это массив одномерных массивов и т.д. Пример:

```
char id[8];      // 8 байт для 8-элементного массива символов
float price[3]; // вектор из 3х вещественных чисел
int m[4][3] = {{1,2,3},{4,5,6},{7,8,9},{0,1,2}}; // матрица целых чисел
```

Доступ к элементам массива выполняется при помощи операции *квадратные скобки* [ ]. Нумерация элементов всегда начинается *с нуля*. То есть первый элемент массива – это всегда элемент с нулевым номером. Например, массив *int a[100]* содержит следующие элементы: *a[0], a[1], a[2], ... , a[99]*. Легко подсчитать, сколько байт памяти потребуется под одномерный массив, зная, что размер базового типа может быть получен при помощи функции *sizeof(тип)*:

```
количество байт = размер_базового_типа * количество_элементов_в_массиве
количество байт = sizeof(тип) * длина_массива
```

В языке *C++* под массив всегда выделяется непрерывное место в оперативной памяти. Выход массива за свои (определенные описанием) пределы компилятором не проверяется. Это следует помнить. То есть, например, если массив имеет 100 элементов и описан как *a[100]*, то обращение к элементу *a[200]* компилятор языка C++ не считает ошибкой. Выход за пределы памяти, отведенной под массив – *контроль диапазона* – полностью отдается программисту.

Листинг 46

```
#define N 10 // макрос размерность массива

void _tmain(int argc, _TCHAR* argv[])
{       srand(time(NULL)); // инициализация генератора случайных чисел

        int A[N];

        for (int i=0; i<N; i++) // заполнение массива случайными числами
                A[i]= rand()%100;

        for (int i=0; i<N; i++) // вывод массива чисел с адресами ячеек
                printf("A[%2d]= %2d (%p)\n", i, A[i], &(A[i]));

        printf("\nA= %p   &(A[0])= %p", A, &(A[0]));

        cin.get();
}
```

Рис. 29. Вывод на экран массива чисел с адресами ячеек

Элементы массива хранятся в памяти последовательно (*Рис. 29*). Например, для массива *int A[10]* начальный (нулевой) элемент хранится по адресу 002CFBA8, то второй будет храниться по адресу 002CFBAC, третий – по адресу 002CFBB0 и т.д., поскольку размер ячейки памяти под целое число – 4 байта (*Рис. 30*):

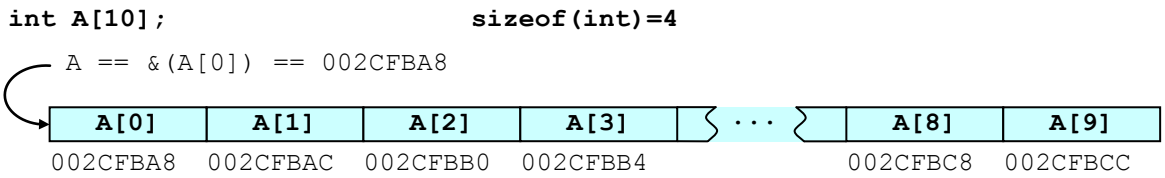

Рис. 30. Размещение в памяти элементов массива

Имя массива – это адрес, начиная с которого последовательно хранятся все ячейки массива (см. *Листинг 46, Рис. 29*).

5.1. Указатели и массивы в C++

В языке Си существует два способа обращаться к ячейкам памяти (переменным) – *по имени* и *по адресу*. Ячейки массива расположены в памяти одним монолитным блоком (*Рис. 30*) и имеют одно общее имя. Но они имеют и адреса, отстоящие друг от друга на размер ячейки массива (*Рис. 29*). Поэтому доступ к членам массива можно получать как через *имя массива и индекс*, так и через *адрес и смещение*. Имя массива является указателем на начальный (нулевой) элемент массива, а индекс ячейки – смещением относительно начального адреса.

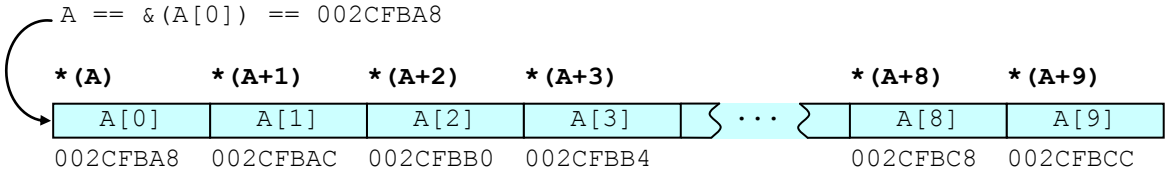

Рис. 31. Доступ к элементам массива по адресу и смещениию

Таким образом, выражение типа *A[i]* интерпретируется компилятором как *адрес[смещение]*, а на этапе компиляции все записи такого вида переводятся в формат **(адрес+смещение)*, только потом выполняются.

В примере ниже (*Листинг 47*) показано, что доступ к элементам массива вида *A[i]*, **(i+A)* и **(A+i)* будут интерпретированы одинаково. Напомним, что запись *(A+i)* в соответствии с правилами сложения адресов (*см. Гл.0*), означает увеличение адреса *A* не на *i* байт, а на *i* ячеек соответствующего размера, в нашем случае – на *i*sizeof(int)* байт.

Листинг 47

```
#define N 10 // макрос размерность массива
void _tmain(int argc, _TCHAR* argv[])
```

```
{       srand(time(NULL));
        int A[N]; // статический массив
        for (int i=0; i<N; i++) // заполнение массива случайными числами
                *(A+i)= rand()%100;
printf("массив A= %p адрес начального элемента &(A[0])= %p", A, &(A[0]));

        for (int i=0; i<N; i++)
        {       // вывод на экран массива в формате имя[индекс]
                printf("A[%2d]= %2d (%p)\t", i, A[i], &(A[i]));
                // вывод на экран массива в формате *(адрес+смещение)
                printf("*(A+%2d)= %2d (%p)\n", i, *(A+i), A+i);   }
                cin.get();
}
```

Результаты работы программы (*Листинг 47*) приведены на рисунке ниже (*Рис. 32*), из него можно видеть, что значения, хранящиеся в ячейках массива можно вызывать *имя[индекс]* и через **(адрес+смещение)*. Для нашего массива *A* ячейки имеют следующие названия: или *A[i]*, или **(A+i)*. Кроме того, *адреса ячеек массива* можно получить по запросу *&(A[i])* или *(A+i)* – это одно и то же.

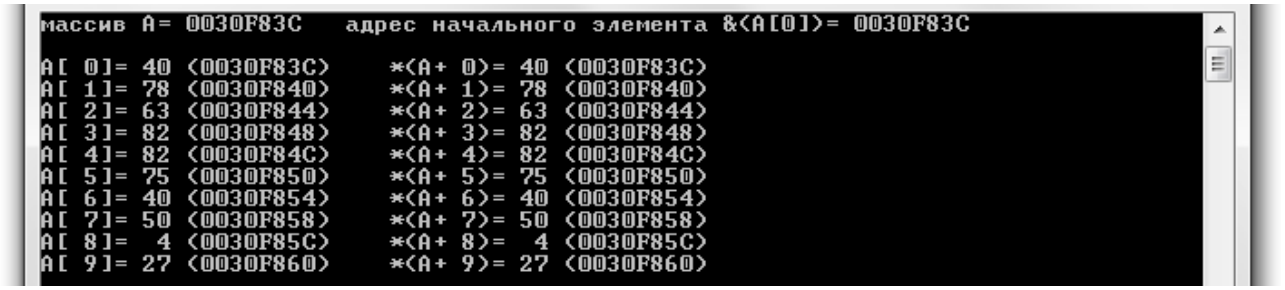

Рис. 32. Доступ к элементам массива по адресу и смещению

5.2. Динамические одномерные массивы

Мы научились обращаться к элементам массива, используя метод **(адрес+смещение)*, также мы понимаем, что адрес массива хранится в указателе с его именем. Зная это можно создавать *динамические массивы*.

По аналогии со статическими и динамическими переменными (*пп. 3.2, 3.3*), массивы также могут являться статическими и динамическими. Динамические массивы не имеют имени, и для работы с ними требуется хранить где-то адрес выделенного оператором *new* или функциями *calloc( ), malloc( )* и *realloc( )* блока памяти. Освобождается память массива при помощи оператора *delete[]* или функции *free()*:

```
double *A; // указатель для хранения адреса динамического массива
A= new double[10]; // выделение памяти под массив
...             // оператор new возвращает адрес выделенного блока памяти
delete[] A; // освобождение памяти

double *B; // указатель для хранения адреса динамического массива
B= (double*)malloc(10*sizeof(double)); // выделение памяти под массив
...
free(B); // освобождение памяти
```

Функция *malloc( )* возвращает нетипизированный адрес выделенной под массив блок памяти в виде *void**, так что его перед сохранением в указателе *B* необходимо принудительно типизировать *B= (double*)...*

Возвращаемый функцией выделения памяти указатель можно использовать для контроля корректного выделения памяти, как показано в примере ниже (*Листинг 48*) – если диспетчер ОЗУ операционной системы по какой-то причине не смог выделить

память по запросу функции *calloc( )*, то будет возвращен пустой указатель *NULL*, что и проверяется в приведенном примере:

Листинг 48

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

#define N 200 // макрос размерность массива

void _tmain(int argc, _TCHAR* argv[])
{       setlocale(LC_ALL, "Russian");

        double *MASSIV; //указатель для хранения адреса массива
        if ((MASSIV= (double*)calloc(N, sizeof(double))) == NULL)
        {       cout << "память под массив выделить не удалось" << endl;
                cin.get(); // ожидание нажатия клавиши
                return; }

        for (int i=0; i<N; i++)
                MASSIV[i] = i+i*0.0001;

        for (int i=0; i<N; i++)
                printf("A[%3d]= %8.4f (%p)\n", i, MASSIV[i], (MASSIV+i));

        cin.get();
        printf("\nsizeof(A)= %d    sizeof(calloc(A))= %d\n",
                sizeof(MASSIV), N*sizeof(double));

        free(MASSIV); // освобождение памяти
}
```

Рис. 33. Результаты выполнения программы

Как и в статическом случае, обращаться к содержимому и адресам ячеек массива можно через *имя[индекс]* и через **(адрес+смещение)*. Для нашего массива *MASSIV* так: *MASSIV[i]* или **(MASSIV+i)*, а *адреса ячеек: &(MASSIV[i])* или *MASSIV+i*.

В приведенном выше примере (*Листинг 48, Рис. 33*) создается массив, ячейки которого являются числами типа *double*, поэтому выделяется массив памяти размером *N*sizeof(double)*, т.е. непрерывный блок памяти 1600 байт. Этим же объясняется тот факт, что адреса ячеек (см. *Рис. 33*) отстоят друг от друга на 8 байт.

Напомним, что подпрограмма *calloc( )* не только выделяет память и возвращает указатель на нее, но и заполняет ячейки созданного массива нулями, что иногда бывает удобно.

Копирование массивов

Как уже многократно говорилось: *имя массива – это адрес,* начиная с которого хранится массив. Поэтому невозможно присвоить один массив другому одним оператором присваивания – будет скопирован только адрес (*Листинг 49*). Массивы должны копироваться *поэлементно*, например, при помощи цикла:

Листинг 49

```
float prev[20000], *current;
for (int i=0; i<20000; i++)
        prev[i]= i*0.000001;
// current = prev; // неправильно – копируется только адрес, не элементы
// правильно так:
current= new float[20000]; // выделение памяти под массив
int i = 0;
while (i<20000)
{       current[i] = prev[i]; // поэлементное копирование
        i++;
}
```

5.3. Передача массива в функцию

Язык *C++* не допускает копирования всего массива *«по значению»* для передачи его в функцию. Это объясняется ограниченным размером *стека вызова подпрограмм*, а ведь массивы могут быть огромными. Однако можно передавать элемент массива или начальный адрес массива. При передаче начального адреса массива в функцию она может иметь непосредственный доступ к элементам исходного массива *«по адресу»*, ведь имя массива это…

Листинг 50

```
void String30Copy (char[], char[]); // прототип функции

void _tmain(int argc, _TCHAR* argv[])
{       char current[30], target[30];
        StringCopy (current, target); // вызов функции
}

void String30Copy (char str1[], char str2[]) // реализация функции
{       for(int i= 0; i<30; i++)
                str1[i] = str2[i];
}
```

При передаче массива, как параметра в функцию, на самом деле передаётся значение имени массива – указатель на первый элемент массива (*Рис. 30*). Во внутреннем пространстве имен функции *String30Copy( )* создаются две локальных переменных *char *str1* и *char *str2*, в них при вызове функции копируются значения фактических параметров – указателей *current* и *target*, интерпретируемые как адреса первых элементов соответствующих массивов.

5.4. Переименование типов (typedef)

Для того чтобы сделать программу более ясной, можно задать типу новое имя с помощью ключевого слова *typedef*. Введенное таким образом имя можно использовать таким же образом, как и имена стандартных типов.

```
typedef unsigned int UINT;
typedef char Msg[100];

UINT i, j; // две переменных фактически типа unsigned int
Msg str[10]; // двумерный массив из 10 строк по 100 символов
```

Переименование типов *typedef* используется исключительно для удобства программиста, так как отладочная среда перед компиляцией программного кода всё равно переобозначает заданные оператором *typedef* новые типы обратно их исходными смыслами.

Макроподстановка #define

Точно также компилятор поступает с константами, задаваемыми при помощи директивы #define – на этапе препоцессирования заменяет *макросы* их содержанием, а уж после этого переходит к компиляции.

```
#define rndm(a,b) (rand()%(b-a))+a // макроподстановка
#define N 20000 // макрос-константа
```

Макроподстановки (или *макроопределения, макросы*) очень удобно применять для задания размерности массива, если нам потребуется изменить его размер, то достаточно изменить число в макроопределении, а изменять размерность массива во всех циклах не придется. Сравните с точки зрения применения макроопределений следующие примеры – удачные реализации *Листинг 47*, *Листинг 48* и неудачные – *Листинг 49*, *Листинг 50*.

5.5. Лабораторная работа № 5. Одномерные массивы

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цели работы

Научиться создавать массивы, выделять память под элементы массива и инициировать их значениями. Научиться обращаться к элементам массивов как при помощи оператора *имя[индекс]*, так и посредством методики **(адрес+смещение)*. Освоить понятия: *адрес массива, адрес элемента массива, смещение, индекс*. Освоить пошаговую трассировку программы с массивами, научиться отражать в окне *watch* элементы массива и их адреса. Научиться передавать массив в функцию.

Задание на лабораторную работу

1. Написать подпрограмму, выполняющую расчеты в соответствии со своим вариантом индивидуального задания. Выполнение задания реализовать с использованием массива соответствующего типа и размера.
2. Реализовать вызов функции из главной программы и вывод результатов вычисления на экран. Отладить и протестировать программу. В процессе отладки пользоваться средствами пошаговой трассировки кода, продемонстрировать в окне *Watch* элементы массива и их адреса.
3. Массив описать и инициировать в теле основной программы, передать его в функцию для расчетов. Вывести на экран адрес массива и его элементы.
4. Продемонстрировать в программе обращение к элементам массивов при помощи оператора *имя[индекс]*, так и посредством методики **(адрес+смещение)*.
5. В отчете отразить процесс трассировки кода, результаты работы программы, листинг программного кода с комментариями, блок-схему программы.

Варианты индивидуальных заданий

1. Напишите функцию, которая суммирует последовательность целых чисел размерности *N*, динамически выделяемых оператором *new*.
2. Создайте функцию, которая находит наименьшее из чисел вещественного массива размерности *N*, выделяемого оператором *new*. Причем, если минимумов несколько, то определять требуется последний из них.
3. Напишите функцию, которая по заданному массиву вещественных чисел x_i размерности N строит массив отклонений от заданного числа X по формуле: $\sqrt{\left|x_i^2 - X^2\right|}$.

4. Создайте функцию, которая находит наибольший элемент и его номер в массиве *int * arr* размерности *N*, выделяемого оператором *new*, причем, если максимумов несколько, то определять требуется первый из них.

5. Напишите функцию, определяющую произведение всех элементов массива типа *double* заданного размера *N*, выделяемого оператором *new*. Массив проинициализировать набором *N* случайных чисел в диапазоне 2..50.

6. Напишите функцию, строящую по заданному массиву целых чисел другой массив натуральных логарифмов этих чисел.

7. Создайте функцию, которая находит и возвращает первое встретившееся наименьшее и наибольшее значение из массива вещественных чисел заданного размера *N*. Задайте значения элементов с помощью генератора случайных чисел из диапазона 30..100.

8. Напишите функцию, которая находит сумму и среднее арифметическое элементов массива *int * arr* заданного размера *N*. Задайте значения элементов с помощью генератора случайных чисел.

9. Напишите функцию, вычисляющую среднеквадратичное отклонение элементов массива x_i типа *double* размерности *N* по формуле: $\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x-\overline{x}\right)^2}$, где $\overline{x}=\frac{1}{N}\sum_{i=1}^{N}x_i$.

10. Создайте функцию, которая находит и возвращает последнее наименьшее и наибольшее значение из набора чисел типа *double* заданного размера *N*.

11. Напишите функцию, которая сортирует по возрастанию массив элементов целого типа заданного размера *N*. Память под массив выделять при помощи оператора *new*, инициировать массив случайными числами.

12. Создайте функцию, которая сортирует массив символов по убыванию. Память под массив выделять при помощи оператора *new*, инициировать массив случайными буквами (а не другими символами).

13. Напишите функцию, которая вычисляет и возвращает количество пар равных друг другу элементов массива целых чисел заданного размера *N*, не только соседних и расположенных в любом месте массива.

14. Заданы три массива вещественных чисел размерности *N*, содержащих координаты x_i, y_i, z_i точек в пространстве. Напишите функцию, вычисляющую расстояния от всех *N* точек массивов до заданной точки $A\left(\overline{x}, \overline{y}, \overline{z}\right)$, напомним, что $d=\sqrt{\left(x_1-x_2\right)^2+\left(y_1-y_2\right)^2+\left(z_1-z_2\right)^2}$.

15. Создайте функцию, которая вычисляет и возвращает количество элементов массива элементов типа *long* заданного размера *N*, равных *m*.

16. Напишите функцию, вычисляющую индексы последнего максимума и первого минимума в массиве вещественных чисел размерности *M*. Массив задавать динамически, заполнять случайными числами.

17. Создайте функцию, по имеющемуся массиву *ASCII*-символов размера *N* создающую массив кодов этих символов. Исходный массив задавать динамически, заполнять случайными символами.

18. Напишите функцию, создающую по одному заданному массиву вещественных чисел размерности *N* два массива – массив целых частей заданных чисел и массив их дробных частей.

19. Создайте функцию, вычисляющую массив остатков от деления нацело на заданное число *A* элементов заданного массива целых чисел размерности *M*.

20. Заданы два массива вещественных чисел размерности N – координаты x_i, y_i точек на плоскости. Напишите функцию, вычисляющую расстояния от точек массивов до заданной точки $A(\bar{x}, \bar{y})$, напомним, что $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

21. Напишите функцию, которая по заданному массиву вещественных чисел размерности M формирует массив модулей разности элементов заданного массива и заданного числа A.

22. Создайте функцию, вычисляющую количество вхождений символа «*A*» в заданный массив символов длины *N*. Исходный массив задавать динамически, заполнять случайными буквами.

23. Напишите функцию, вычисляющую номера первого максимума и последнего минимума в массиве целых чисел заданной размерности. Массив задавать динамически, заполнять случайными числами.

24. По заданному массиву абсцисс (чисел типа *double*) вычислите значения ординат функции $F(x_i) = \sin\left(x_i - \frac{1}{N}\sum_{i=1}^{N} x_i\right)$, здесь N – размерность массива.

Контрольные вопросы:

1. Что будет выведено на экран подпрограммой *printf("sizeof(A)= %d", sizeof(A))*, если *A* – одномерный статический массив *double A[100]*?

2. Как скопировать один массив в другой?

3. Что такое динамический массив, как выделять под него память?

4. Какие операторы и функции применяются для освобождения динамической памяти?

5. Как располагаются в памяти элементы массива?

6. Что будет выведено на экран подпрограммой *printf("sizeof(A)= %d", sizeof(A))*, если *A* – одномерный динамический массив *double *A= new double[100]*?

7. Что такое статический массив, какие есть два основных способа обращения к элементам массива?

8. Можно ли хранить в ячейках одномерного массива переменные разных типов?

9. Какие операторы и функции используются для выделения памяти под динамический массив?

10. Как определить адреса статического и динамического массивов, где они хранятся?

11. Являются ли эти записи идентичными для одномерного массива *A*: *&(A[i])* или *A+i*?

5.6. Двумерные массивы

В различных программных приложениях довольно часто приходится производить работу с двумерными массивами – *матрицами*. Необходимость в этом часто возникает в задачах, связанных с математическими вычислениями, преобразования объектов мультимедиа, потоковой обработкой данных, шифровкой и т.п.

5.6.1. Статические двумерные массивы

Рассмотрим, как задается и располагается в памяти компьютера двумерный массив.

Листинг 51

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

#define N 5 // макросы размерность массива
#define M 3

void Set_Array_NxM_rand(double Q[N][M]); // прототипы функций
void Print_Array_NxM(double Q[N][M], char*);

void _tmain(int argc, _TCHAR* argv[])
{       setlocale(LC_ALL, "Russian");

        double F[N][M]; // статический двумерный массив

        Set_Array_NxM_rand(F); // вызов подпрограмм
        Print_Array_NxM(F, "F");

        cin.get();
}

// ---------------------------------------------------------------------
void Set_Array_NxM_rand(double Q[N][M])
{       srand(time(NULL));
        for (int i=0; i<N; i++)
        {       for (int j=0; j<M; j++)
                {       Q[i][j]= rand()/10000.0; }
        }
}

// ---------------------------------------------------------------------
void Print_Array_NxM(double Q[N][M], char* S)
{       printf("матрица %s:\n",S);
        for (int i=0; i<N; i++)
        {       for (int j=0; j<M; j++)
                printf("%s[%d,%d]= %6.4f(%p) ",S,i j,Q[i][j],&Q[i][j]);
                printf("\n");
        }
}
```

```
матрица F:
F[0,0]= 2,7175(002DFAE8) F[0,1]= 1,0055(002DFAF0) F[0,2]= 1,9851(002DFAF8)
F[1,0]= 0,5572(002DFB00) F[1,1]= 2,0543(002DFB08) F[1,2]= 2,3243(002DFB10)
F[2,0]= 2,9406(002DFB18) F[2,1]= 0,4721(002DFB20) F[2,2]= 0,3973(002DFB28)
F[3,0]= 0,2892(002DFB30) F[3,1]= 2,7549(002DFB38) F[3,2]= 0,2473(002DFB40)
F[4,0]= 0,1211(002DFB48) F[4,1]= 0,7513(002DFB50) F[4,2]= 2,2958(002DFB58)
```

Рис. 34. Адресация элементов двумерной статической матрицы

Двумерный массив располагается в памяти построчно, например элементы массива *F[5][3]* хранятся в следующем порядке: *F[0][0]*, *F[0][1]*, *F[0][2]*, *F[1][0]*,

F[1][1], *F[1][2]*, *F[2][0]*, *F[2][1]*, ... , *F[4][1]*, *F[4][2]*, в чем можно убедиться, рассмотрев *Рис. 34.*

При описании списка параметров подпрограммы, в которую необходимо передать статический массив, необходимо указывать размерность этого массива, как в примере выше (*Листинг 51*).

Нужно понимать, что двумерный массив – это «*массив массивов*», т.е. элементами двумерного массива являются одномерные массивы и т.д. Это свойство позволяет понять, как правильно организовать заполнение массива начальными значениями:

```
int MyMas[3][4] = {     { 11, 12, 13, 14 },   // MyMas[0]
                        { 21, 22, 23, 24 },   // MyMas[1]
                        { 31, 32, 33, 34 }    // MyMas[2]
                  };
```

Для того чтобы рассмотреть, какие адреса соответствуют и элементам двумерного массива, добавим приведенный ниже код (*Листинг 52*) в тело программы, работающей со статическим двумерным массивом *double F[N][M]* (*Листинг 51*).

Листинг 52

```
printf("\nадреса строк матрицы %s:\n","F");
for (int i=0; i<N; i++)
       printf("F[%d]: %p %p\n", i, F[i], &F[i]);
printf("\nадрес матрицы F: %p %p %p %p\n", F, F[0], &F[0], &F[0][0]);
```

```
адреса строк матрицы F:
F[0]: 002DFAE8 002DFAE8
F[1]: 002DFB00 002DFB00
F[2]: 002DFB18 002DFB18
F[3]: 002DFB30 002DFB30
F[4]: 002DFB48 002DFB48

адрес матрицы F: 002DFAE8 002DFAE8 002DFAE8 002DFAE8
```

Рис. 35. Адресация строк двумерной статической матрицы

Сравнив полученные результаты (*Рис. 35*) с предыдущими (*Рис. 34*), можно видеть, что *&F[i]* – адреса строк *F[i]* двумерной матрицы *F* совпадают с адресами начальных элементов этих строк *&F[i][0]*. Кроме того, адрес *F* самой матрицы совпадает с адресом *&F[0]* начальной строки *F[0]* и адресом начального элемента *&F[0][0]*.

Понимание принципов построения и размещения в памяти компьютера двумерных статических массивов позволяет перейти к построению аналогичных *динамических* конструкций.

5.6.2. *Двумерные динамические массивы. Массивы указателей*

Гибкость семейства языков программирования, относимых к *C++*, позволяет организовать хранение многомерных массивов данных многими способами.

Рассмотрим способ, основанный на выделении динамической памяти под матрицу в виде указателя, хранящего адрес динамического массива указателей, каждый из которых содержит адрес одномерного динамического массива – строк (или столбцов) матрицы.

Этот способ выделения памяти под двумерный $N \times M$ массив состоит из двух этапов – сперва формируется массив из N указателей на строки (или столбцы) матрицы, а затем в цикле создаются одномерные массивы для хранения M переменных заданного типа (*Рис. 36*), память под которые так же выделяется динамически:

Листинг 53

```
//выделение динамической памяти под массив указателей
matr= new Тип* [N];
for (int i=0; i<N; i++)
{       //выделение динамической памяти для массивов значений
        matr[i] = new Тип [M];
}
```

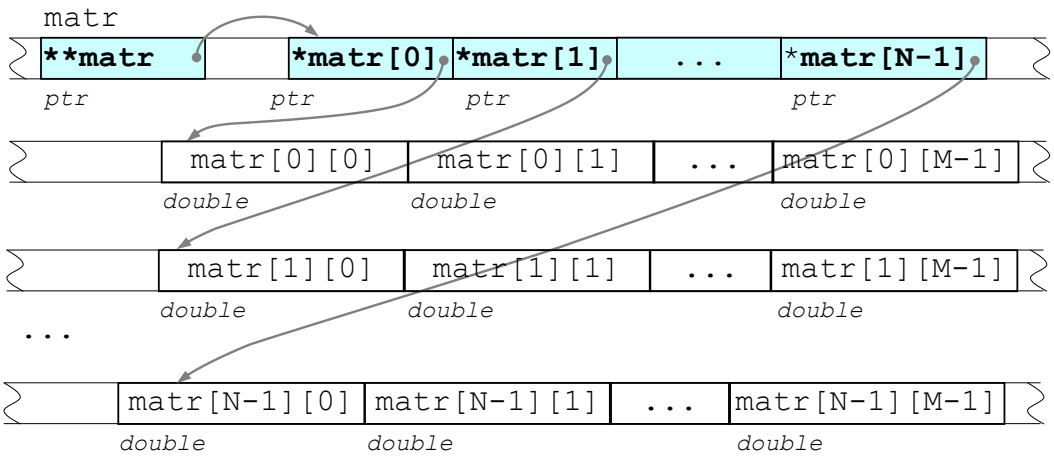

Рис. 36. Расположение двумерного динамического массива в памяти

На рисунке (*Рис. 36*) цветом выделен массив указателей на строки матрицы *matr*, стрелками показано, что в нем хранятся адреса строк матрицы. В отличие от статического двумерного массива, выделение памяти под строки матрицы происходи не одномоментно, а последовательно в итерациях цикла (*Листинг 53*), отдельно под каждую строку из «кучи». Поэтому строки расположены не обязательно одна за другой – одномерные массивы строк могут располагаться в разных областях памяти.

Листинг 54

```
double** dd; // указатель для хранения адреса массива указателей
const int M= 4; // размерность матрицы N*M
const int N= 3;

dd = new double*[N]; // выделение памяти под массив указателей
for (int i=0; i<N; i++)
        dd[i] = new double[M]; // выделение памяти под массивы значений
матрицы
```

При таком способе выделения памяти переменная *dd* имеет тип *double*** и фактически представляет собой *«указатель на указатель на ячейку типа double»*. Команда *dd= new double*[N]* создает массив указателей, каждая ячейка которого представляет собой указатель *double**, хранящий адрес одного из *N* одномерных массивов *dd[i]*, предназначенных для хранения собственно значений матрицы. На начальном этапе массив указателей не заполнен, поэтому дальше в цикле по счетчику *i* от 0 до *N* выделяется память под одномерные массивы рациональных значений *dd[i]* и адреса этих массивов сохраняются в соответствующих ячейках массива указателей *dd*.

```
int n, m; // n и m – количество строк и столбцов матрицы
float** matr; // указатель для массива указателей
matr= (float**) malloc(n*sizeof(float *));
// выделение динамической памяти под массив указателей
for (int i=0; i<n; i++)
        // выделение динамической памяти для массива значений
        matr[i]= (float*) malloc(m*sizeof(float));
```

Так как функции *malloc()* и *calloc()* возвращают *нетипизированный* указатель *void **, то необходимо выполнять его явное преобразование в указатель объявленного типа.

Адресация элементов динамических массивов такого вида осуществляется с помощью индексированного имени, точно также как и для статической матрицы при помощи оператора «*квадратные скобки*»:

```
ИмяМассива[ЗначениеИндекса][ЗначениеИндекса];
matr[i][j];
```

Рассмотрим присвоение значений двумерному массиву – матрице *dd*.

```
// Матрица (двумерный массив)
cout << "адрес двумерного массива " << dd << " содержимое матрицы:\n";
for (int i=0; i<N; i++, cout << endl) // Двумерный массив
        for (int j=0; j<M; j++)
        {       dd[i][j] = (double)(i + j);
                cout<< "dd[" << i <<"]["<< j <<"]="<< dd[i][j] << endl; }
```

Если теперь удалить из памяти массив *dd*, например, командой *delete []dd*, то будут потеряны адреса всех одномерных массивов-строк, и мы не сможем работать с ними и даже удалить их не сможем – *утечка памяти!* Поэтому удаление из памяти двумерного массива осуществляется в порядке, обратном его созданию, то есть сначала освобождается память, выделенная под одномерные массивы с данными *dd[i]*, а только затем память, выделенная под одномерный массив указателей *dd*.

```
//освободить память, выделенную для N массивов значений
for (int i=0; i<N; i++)
        delete dd[i];
//освободить память, выделенную под массив указателей
delete []dd;
```

Квадратные скобки *[ ]* в последней строке примера означают, что освобождается память, занятая всеми элементами массива, а не только первым.

Необходимо четко представлять себе, что удаление памяти, выделенной под *массив указателей* до того, как будет освобождена динамическая память *массивов значений*, приведет к ошибке. Так как адреса массивов значений *dd[i]* будут утрачены, не будет возможности обратиться к массивам *dd[i]* для освобождения занятой ими памяти. Эта «потерянная» часть динамической памяти будет не доступна для программы пользователя и не сможет быть перераспределена диспетчером памяти операционной системы другим программам, так как она выделялась под пользовательское приложение. Это основная ошибка при работе с динамической памятью.

Участок памяти, выделенный ранее операцией при помощи библиотечных функций *malloc()* и *calloc()* освобождается при помощи библиотечной функции *free()*:

```
//освободить память, выделенную для массива значений
for (int i=0; i<n; i++)
        free(matr[i]);
//освободить память, выделенную под массив указателей
free (matr);
```

Рассмотрим пример (*Листинг 55*) конструирования в памяти двумерного динамического массива, заполнения его значениями, вывода на экран и корректного удаления. Видно, что матрица размещена в памяти аналогично статической, но имеются отличия.

Листинг 55

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

#define n 5 // размерность массива
#define m 3

long double **Set_Matrix_NxM(int N, int M)
{    srand(time(NULL));
     // выделение памяти для массива N указателей (long double*)
     long double** Q= (long double**) malloc(N*sizeof(long double*));
     // выделение памяти для массивов M значений (long double)
     for (int i=0; i<N; i++)
          Q[i]= (long double*) malloc(M*sizeof(long double));
     for (int i=0; i<N; i++) // заполнение массива
          for (int j=0; j<M; j++)
               Q[i][j]= rand()/10000.0;
     return Q;
}

void Print_Array_NxM(long double** Q, char* S, int N, int M)
{    printf("матрица %s:\n",S);
     for (int i=0; i<N; i++)
     {    for (int j=0; j<M; j++)
          printf("%s[%d,%d]= %6.4f (%p)  ", S, i, j, Q[i][j], &Q[i][j]);
          printf("\n");
     }
}

void Free_Array_NxM(long double** Q, int N, int M)
{    for (int i=0; i<N; i++) // удаление строк
          free (Q[i]);
     free (Q); // удаление массива указателей
}

void _tmain(int argc, _TCHAR* argv[])
{    setlocale(LC_ALL, "Russian");

     long double** R; // указатель для массива указателей
     R= Set_Matrix_NxM(n, m);
     R[3][1]= 3.33333333; // изменить один элемент матрицы
     Print_Array_NxM(R, "R", n, m);

     printf("адреса строк матрицы %s:\n","R");
     for (int i=0; i<n; i++)
          printf("%s[%d]: %p %p\n", "R", i, R[i], &R[i]);

   printf("адрес матрицы %s: %p %p %p %p\n","R", R, R[0], &R[0], &R[0][0]);
     Free_Array_NxM(R, n, m);
     cin.get();
}
```

Из рисунка (*Рис. 37*), отображающего результаты работы программы (*Листинг 55*) можно видеть, что адрес, хранящийся в указателе *R[i]* совпадает с начальным элементов соответствующей строки *&R[i][0]*, но адрес ячейки массива указателей *&R[i]* – другой. Точно также как адреса *R* и *&R[0]* не совпадают с адресами *R[0]* и *&R[0][0]*.

Сравните размещение элементов и строк двумерной динамической матрицы (*Рис. 37*) и двумерного статического массива (*Рис. 34*, *Рис. 35*).

```
матрица R:
R[0,0]= 0,0444 (00824318)  R[0,1]= 0,3704 (00824320)  R[0,2]= 2,8696 (00824328)
R[1,0]= 1,6361 (00824370)  R[1,1]= 0,3543 (00824378)  R[1,2]= 2,2942 (00824380)
R[2,0]= 1,9589 (00828E50)  R[2,1]= 1,9530 (00828E58)  R[2,2]= 2,9185 (00828E60)
R[3,0]= 1,8043 (00828EA8)  R[3,1]= 3,3333 (00828EB0)  R[3,2]= 0,1591 (00828EB8)
R[4,0]= 1,5967 (00828F00)  R[4,1]= 1,9506 (00828F08)  R[4,2]= 1,5560 (00828F10)

адреса строк матрицы R:
R[0]: 00824318 008242C8
R[1]: 00824370 008242CC
R[2]: 00828E50 008242D0
R[3]: 00828EA8 008242D4
R[4]: 00828F00 008242D8

адрес матрицы R: 008242C8 00824318 008242C8 00824318
```

Рис. 37. Адресация элементов двумерной динамической матрицы

5.6.3. *Двумерные динамические массивы, представленные одномерными массивами*

Поскольку обращение к элементам массива эквивалентно работе с указателями, то для работы с матрицами можно использовать обычные указатели и хранить многомерный массив *NxM* в памяти построчно в виде одномерного массива *N*M* элементов. После описания указателя, необходимо будет выделить динамическую память (в куче) для хранения *N*M* элементов (*Рис. 38*), например, при помощи оператора *new*, или посредством функций *calloc( )* или *malloc( )*.

```
float *A;
int n, m;
A= (float *) calloc(N*M, sizeof(float));
```

Описанным ниже образом для выделения памяти под многомерный массив можно использовать функцию *malloc( )* или оператор *new*:

```
A=(float *) malloc (N*M*sizeof(float));
A=new float(N*M);
```

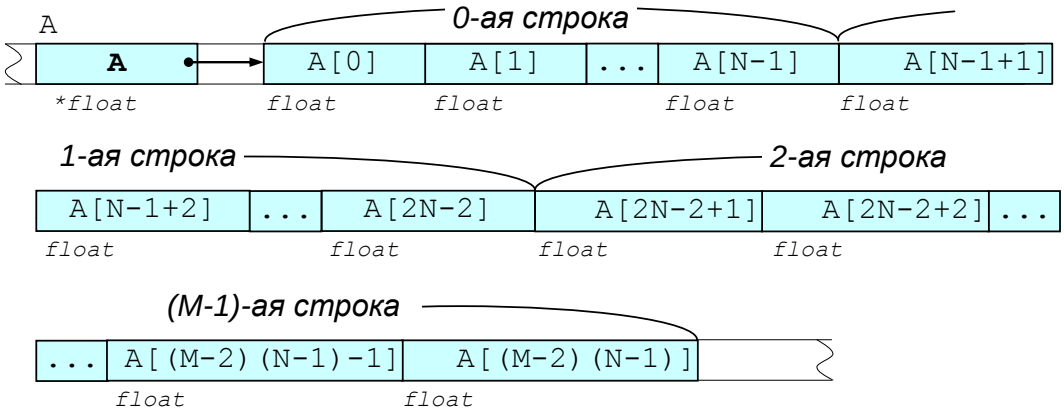

Рис. 38. Распределение в памяти элементов двумерного массива «в строку»

Все элементы двумерного массива (матрицы) хранятся в одномерном массиве размером *N*M* элементов построчно: сначала в этом массиве расположена 0-я строка матрицы, затем 1-я и т.д. Поэтому для обращения к элементу $A_{i,j}$ необходимо по номеру строки *i* и номеру столбца *j* матрицы вычислить номер этого элемента *k* в одномерном динамическом массиве. Учитывая, что в массиве элементы нумеруются с нуля $\boldsymbol{k = i \cdot M + j}$, обращение к элементу *A[i][j]* будет таким ***(A+i*m+j)**. За все надо платить: простота создания и удаления динамической матрицы такого вида (*Листинг 56*, *Рис. 39*) компенсируется неудобством доступа к ее элементам.

В качестве примера работы с динамическими матрицами рассмотрим следующую задачу (*Листинг 56*): Заданы две матрицы вещественных чисел *A(N, M)* и *B(N, M)*. Вычислить сумму матриц – матрицу *C = A+B*.

Листинг 56

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

void _tmain(int argc, _TCHAR* argv[])
{     setlocale (LC_ALL, "RUS");
      int i, j, N, M;
      double *a,*b,*c; // указатели на динамические массивы
      cout<<"N = "; cin>>N; // ввод размерности матрицы
      cout<<"M = "; cin>>M;

      a=new double[N*M]; // выделение памяти для матриц
      b=new double[N*M];
      c=new double[N*M];

      // ввод матрицы A
      cout<<"введите матрицу A"<<endl;
      for (i=0; i<N; i++)
            for (j=0; j<M; j++)
                  cin>> *(a+i*M+j);

      //ввод матрицы B
      cout<<"введите матрицу B"<<endl;
      for (i=0; i<N; i++)
            for (j=0; j<M; j++)
                  cin>> *(b+i*M+j);

      //вычисление матрицы C=A+B
      for (i=0; i<N; i++)
            for (j=0; j<M; j++)
                  *(c+i*M+j)= *(a+i*M+j)+*(b+i*M+j);

      //вывод матрицы C
      cout<<"матрица C:"<<endl;
      for (i=0; i<N; cout<<endl, i++)
            for (j=0; j<M; j++)
                  cout<<*(c+i*M+j)<<"\t";

      delete []a; //освобождение памяти
      delete []b;
      delete []c;
      system("pause");
}
```

```
N = 4
M = 3
введите матрицу A
1 2 3
4 5 6
7 8 9
0 1 2
введите матрицу B
1 1 1
2 2 2
3 3 -3
4 4 4
матрица C:
2       3       4
6       7       8
10      11      6
4       5       6
```

Рис. 39. Результаты работы программы

Как распределены в памяти элементы матрицы, представленной в виде одномерного массива, можно видеть, если вывести адреса матрицы *C* на экран (*Рис. 40*):

Рис. 40. Распределение в памяти элементов двумерного массива «в строку»

5.7. Практическая работа № 6. Двумерные массивы

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Освоить способ динамического захвата и освобождения памяти под двумерные массивы данных (матрицы) – при помощи оператора *new* и посредством функций *malloc( )* и *calloc( )*. Повторить понятия, операции и закрепить умения и навыки матричной алгебры. Уяснить практическую разницу и сходство в языке *C++* понятий *«массив»* и *«указатель»*. Повысить навыки отладки программного кода на *C++*, трассировки программы и просмотра значений переменных в окне *Watch*.

Задание на практическую работу

1. Написать, отладить и протестировать программу, выполняющую следующие операции с одномерными и многомерными массивами данных (в соответствии со своим вариантом, см. *Таблица 12*)

2. Во втором столбце таблицы индивидуальных заданий (*Таблица 12*) – способ выделения памяти под матрицу: 1 - статический двумерный массив (как в *п. 5.6.1*), 2 - хранение матрицы в памяти построчно в виде одномерного массива (как в *п. 5.6.3*); 3 - как массив указателей, содержащий адреса строк матрицы (как в *п. 5.6.2*).

3. Операции выделения памяти под матрицы и вектора, вывод на экран, операции матричной арифметики, освобождение памяти и т.п. организовать в виде подпрограмм (функций).

4. При выполнении операции *обращения к элементам матрицы* обеспечить защиту от выхода за границы диапазона строк и столбцов.

5. При выполнении операций матричной алгебры размерность вектора и второй матрицы задать самостоятельно таким образом, чтобы была возможность эти операции производить. Организовать в подпрограммах проверку корректности размерностей матриц: при произведении – проверять сцепленность матриц, при сложении – совпадение размерностей.

6. При демонстрации преподавателю работающей программы необходимо провести пошаговую трассировку указанного преподавателем участка программы с комментариями производимых программой действий. При трассировке адреса динамических переменных и их значения необходимо выводить в окне *Watch*.

7. По выполненной лабораторной работе подготовить отчет, включающий блок-схемы алгоритмов процедур и рисунок, отражающий распределение памяти под матрицу. Отчет в обязательном порядке снабдить комментариями.

8. В отчете привести проверку правильности произведенных вычислений, сделанную в программном пакете MatCAD, или другом математическом приложении.

9. Повышенный балл выставляется за выполнение программы с контролем адресов и выделяемой под динамические объекты памяти.

Варианты индивидуальных заданий к лабораторной работе содержит *Таблица 12*.

Контрольные вопросы:

1. Как размещается в памяти статический двумерный массив?

2 Задана матрица $N \times M$, при такой размерности N – это число строк или столбцов?

3. Как выделяется память под матрицу, хранящуюся в виде одномерного массива?

4. Как используется вложенный цикл при работе с матрицами?

5. В чем состоит способ выделения памяти под матрицу - *как массив указателей*, содержащий адреса строк матрицы.

6. Как организовать двумерный динамический массив символов?

7. Как обращаться к элементам с координатами i и j матрицы, хранящейся в виде одномерного массива?

8. Чем фактически являются строки любой – статической или динамической матрицы?

9. Как передается в подпрограмму статический двумерный массив?

10. Как задать статическую матрицу начальными значениями – константами?

11. Какие виды выделения памяти под двумерные динамические массивы вы знаете?

12. Как обращаться к элементам двумерного статического массива при помощи оператора «*квадратные скобки*» и по технологии «*адрес+смещение*».

13. Можно ли размещать в ячейках двумерного динамического массива адреса?

14. Как понимать следующую запись *&R[0][0]*? Чему равно это значение?

Таблица 12. Варианты индивидуальных заданий к практической работе

№	способ выделения памяти[2]	тип данных элементов матрицы	размерность массива A [NxM]	операции с вектором x	операции с матрицами A и B	вычислить
1.	1	double	[6,6]	3x*A	A+2*B	сумму элементов основной диагонали
2.	2	unsigned long	[6,5]	A*2x	A-4*B	сумму элементов последнего столбца
3.	3	int	[5,5]	x*3A	2*(A+B)	произведение элементов основной диагонали
4.	2	unsigned int	[4,5]	4A*x	3*(A-B)	произведение элементов последнего столбца
5.	1	long	[3,5]	4x*A	2*A+3*B	сумму элементов основной диагонали
6.	3	unsigned long	[2,5]	A*5x	3*A-2*B	произведение элементов первой строки
7.	1	float	[6,4]	x*2A	B-3*A	сумму элементов первой строки
8.	2	double	[5,4]	3A*x	B+4*A	минимальный элемент последней строки
9.	3	int	[4,4]	4x*A	3*B-A	удвоенную сумму элементов основной диагонали
10.	2	unsigned int	[3,4]	A*6x	5*B+2*A	максимум сумм элементов строк
11.	1	float	[2,4]	x*5A	3*(B+A)	максимальный элемент первой строки
12.	3	double	[6,3]	3A*x	4*(B-A)	минимум сумм элементов строк
13.	1	int	[5,3]	4x*A	3*B-2*A	максимальный элемент последней строки
14.	2	unsigned int	[4,3]	A*4x	5*B-4*A	максимум сумм элементов столбцов
15.	3	float	[3,3]	x*3A	A+3*B	утроенное произведение элементов основной диагонали
16.	2	double	[2,3]	2A*x	A-5*B	минимальный элемент первой строки
17.	1	long	[6,2]	6x*A	4*(A+B)	минимум сумм элементов в столбцах
18.	3	unsigned long	[5,2]	A*5x	2*(A-B)	максимум произведений элементов по столбцам
19.	1	float	[4,2]	x*4A	4*A+3*B	минимальный элемент первого столбца
20.	2	double	[3,2]	3A*x	3*A-4*B	максимум произведений элементов по строкам
21.	3	int	[2,2]	2x*A	B-5*A	сумму элементов дополнительной диагонали
22.	2	unsigned int	[3,3]	A*3x	B+2*A	минимальный элемент последнего столбца
23.	1	float	[3,4]	x*4A	4*B-A	минимум произведений элементов в столбцах
24.	3	double	[3,5]	5A*x	3*B+2*A	максимальный элемент последнего столбца

[2] *способ выделения памяти под матрицу: 1 - статический двумерный массив (п. 5.6.1), 2 - хранение матрицы в памяти построчно в виде одномерного массива (п. 5.6.3); 3 - как массив указателей, содержащий адреса строк матрицы (п. 5.6.2).*

6. Работа со строками

```
- Эй, очкарик! Как найти библиотеку?
- C:WINDOWS\System32\SYSTEM.dll
```

Строки в языке *C++* позволяют работать с символьными данными и текстом [*7, 8*].

6.1. Строки символов

Строку символов можно хранить в виде массива элементов типа *char*. Переменная типа *char* хранит в себе 1 символ, точнее – *ASCII-код* символа. Размер массива символов для хранения такой строки должен быть на 1 больше, т.к. последний элемент массива должен содержать символ '**\0**' (пустая переменная без значения), который обозначает *символ конца строки.* Пример:

```
char name[50];
std::cin >> name;
std::cout << "Hello " << name << endl;
```

При использовании строк такого типа необходимо помнить, что компилятор языка *C++* работает с массивами как с указателями: присваивает и сравнивает лишь *адреса массивов*, но не содержимое, например в результате работы данного фрагмента программы:

Листинг 57

```
char name1[5]= {'к','у','-','к','у'};
char name2[5]= {'к','у','-','к','у'};
if (name1==name2)
{     cout << "строки равны" << endl; }
else
{     cout << "строки НЕ равны" << endl; }
```

на экран будет выведено: *"строки НЕ равны"*, так как полностью идентичные строки *name1* и *name2* имеют разные адреса в памяти.

Поэтому программисту необходимо брать на себя написание функций поэлементного (в цикле) сравнения строк, присваивания значений и объединения строк, поиска в строках символов и т.п. Кроме того, сложностью при работе со строками такой архитектуры является необходимость постоянно контролировать размер массива и невозможность его изменять.

6.2. Строка – массив символов

В листинге, приведенном ниже, задан статический массив элементов *str1* типа *char*.

Листинг 58

```
#include "stdafx.h"
#include <conio.h>
#include <iostream>
using namespace std;
//--------------------------------------------------------
void  tmain(int argc,  TCHAR* argv[])
{     // задать статический массив символов – строку
      char* str1 = "1234567890qwertyuiop";
      cout << str1 << endl;
      cout << *str1 << endl;
}
```

Выделение памяти под статический массив элементов типа *char* происходит в момент задания: определяется его длина *L*, выделяется непрерывная область памяти

размера *L*+1, в первые *L* ячеек массива помещаются символы "*1234567890qwertyuiop*", а в последнюю ячейку – символ '\0' (конец строки), необходимый только для вывода элементов массива.
Имя *str1* представляет собой указатель на массив символов, как показано на *Рис. 41*.

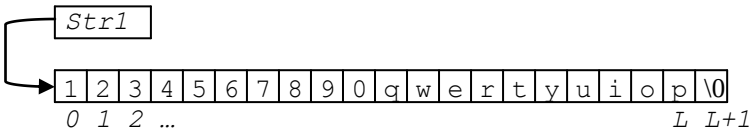

Рис. 41. Строка – это символьный массив

При выводе оператор << класса *std::cout* будет выводить все символы массива, начиная от указателя *str1* до символа '\0', сколько бы их не было. Если символ '\0' разместить в середину строки *str1*, то при выводе массива на экран отразятся не все символы строки, а только от начала до '\0'. Если напротив символ '\0' в конце строки *str1* удалить, то вывод символов не прекратится после окончания строки *str1*, и будут последовательно выводиться на экран в виде символов данные из ячеек, лежащих после строки *str1*, пока не встретится какой-нибудь другой символ '\0'.

Строка – это массив символов, следовательно, указатель *str1* – это адрес первой ячейки массива и содержимое указателя *str1* это содержимое ячейки *str[0]*.

```
char *pstr;
pstr= (char*)malloc(10*sizeof(char));
pstr[3]='\0';
cout << pstr << endl;
```

В данном примере под строку *pstr* при помощи функции *malloc( )* выделяется динамическая память – 10 ячеек размера *sizeof(char)*. Однако, поскольку в третью ячейку массива помещается символ конца строки, оператором << на экран будет выведено только 4 символа из массива *pstr*.

Листинг 59

```
char *pstr;
pstr= (char*)malloc(10*sizeof(char));
for (int i=0; i<9; i++)
      pstr[i]= 'A'+i;
pstr[9]= '\0';
cout << pstr << endl;
```

С содержимым динамической строки можно обращаться как с обыкновенным массивом. В приведенном выше примере (*Листинг 59*) в ячейки массива *pstr* в цикле помещаются коды девяти символов, начиная с 'A', в последнюю ячейку помещается символ конца строки – поскольку строка создается не автоматически, а «вручную», программист сам должен контролировать размер массива символов и размещение символа '\0' в конце строки.

Листинг 60

```
char* str1 = "1234567890qwertyuiop";
char* str2 = str1; // скопирован только указатель
cout << str1 << endl;
cout << str2 << endl;
//----------------------------------------
char* str3;
str3= (char*)malloc(21*sizeof(char));
strcpy(str3,str1); // корректное копирование
cout << str3 << endl;
```

Копирование строк

При копировании строк нужно ясно понимать, что строка – это не только указатель, но и область памяти (динамическая или статическая) в которой лежит сам текст строки. Из примера, приведенного выше (*Листинг 60*), можно увидеть правильное и неправильное копирование строк – более детально это пример разобран ниже, см. *Рис. 42*.

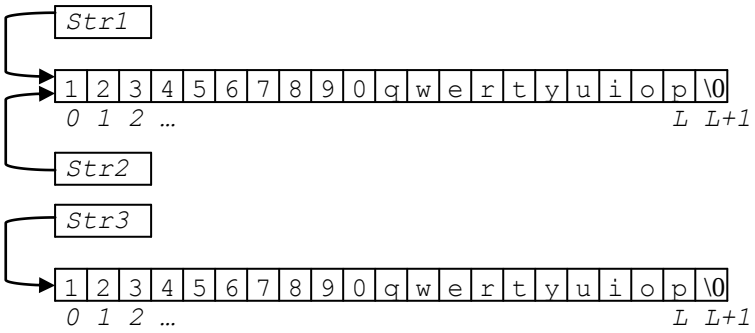

Рис. 42. Копирование строки и указателя на строку

Здесь копирование *str2 = str1* не является правильным, так как копируется только указатель на строку, а массив данных не дублируется. Правильное копирование строк – это выделение памяти и поэлементное копирование одного массива в другой – вручную или с использованием подпрограммы *strcpy(str3, str1)* из библиотеки <*string.h*> (*Таблица 13*).

Оператор << выведет во всех трех случаях одинаковую строку, но в первом и третьем случае (*str1* и *str3*) – это вывод самостоятельных массивов, а при выводе *str2* – это вывод массива *str1*, на который указывает адрес *str2*.

Сравнение строк

Таким же образом происходит и сравнение строк. В примере ниже (*Листинг 61*) строка *str3* является полной точной копией строки *str1*, однако, простое сравнение (*str3==str1*) даст отрицательный результат. В этом случае сравниваются адреса массивов, а не содержимое строк, а адреса-то у них разные!

Листинг 61

```
char* str1 = "1234567890qwertyuiop";
char* str3;
strcpy(str3,str1);
if (str3==str1)
      cout << str3 << " строки равны " << str1 << endl;
else
      cout << str3 << " строки не равны " << str1 << endl;
```

Посимвольно сравнить содержимое строк можно при помощи встроенной функции *strcmp( )*, как в листинге, приведенном ниже (*Листинг 62*).

Важно не перепутать: эта функция возвращает ***0*** (*false*) если содержимое строк идентично, и возвращает целое число – номер позиции в которой лежит первый несовпадающий символ – первое встреченное отличие одной строки от другой (в некоторых реализациях ***-1***).

Листинг 62

```
if (strcmp(str3,str1)==0)
      cout << str3 << " строки равны " << str1 << endl;
else
      cout << str3 << " строки не равны " << str1 << endl;
cout << strcmp(str3,str1) << endl;
```

Длину строки можно найти с помощью функции *strlen( )* (*Таблица 13*).

```
cout << strlen(str1) << endl;
```

Определите самостоятельно, в чем отличие подпрограммы *strlen( )* и функции *sizeof( )*.

Подпрограмма *strstr( )* (*Таблица 13*) предназначена для поиска подстроки в строке, как показано в примере ниже (*Листинг 63*) и на *Рис. 43*. Эта функция возвращает адрес той ячейки массива, с которой начинается подстрока и возвращает пустой указатель *NULL*, если такой подстроки нет.

Листинг 63

```
char *pos1= strstr(str1,"90");
if (pos1==NULL)
      cout << "подстрока не найдена";
else
      cout << pos1 << endl;
pos1[6]='A';
cout << pos1 << endl;
cout << str1 << endl;
```

В этом примере можно видеть, что присваивание символа 'A' в шестую позицию подстроки *pos1* изменит также и 14-ю позицию исходного массива *str1*, так как *pos1* – это не отдельный массив, а только указатель на восьмую ячейку строки *str1* (*Рис. 43*).

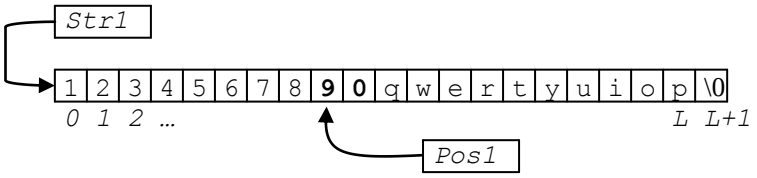

Рис. 43. Поиск и использование подстроки в строке

Для выделения найденной подстроки в виде отдельного массива можно воспользоваться функцией копирования *strncpy( )*:

```
char *pos2= (char*)calloc(4,sizeof(char));
strncpy(pos2,pos1,3);
```

Таблица 13 иллюстрирует краткий список с встроенных функций языка C++, предназначенных для работы со строками, здесь же дается их краткое описание. *Таблица 14* содержит функции, предназначенные для перевода числовых констант различного типа в виде строки символов. Более полную информацию можно посмотреть в справочной системе и в литературе [1-8].

6.3. Библиотека <string.h>

К счастью, имеется возможность использовать готовые подпрограммы для работы со строковыми переменными. Прототипы функций работы со строками содержатся в файле *string.h* (*Таблица 13*). Все функции работают со строками, завершающимися *нулевым байтом* `'\0'`. Для функций, возвращающих указатели, в случае ошибки возвращается *NULL*. Типизированный модификатор *size_t* определен как тип *unsigned*.

Таблица 13 Функции для работы со строками

Функция	Описание
strcat	*char *strcat(char *dest, const char *src);* конкатенация (склеивание) строки src с dest

Функция	Описание
strncat	*char *strncat(char *dest, const char *src, size_t maxlen);* добавить не более n символов из src в dest
strchr	*char *strchr(const char *s, int c);* найти первое вхождение символа c в строку s и вернуть указатель на найденное
strrchr	*char *strrchr(const char *s, int c);* найти последнее вхождение символа c в строку s и вернуть указатель на найденное
strcmp	*int strcmp(const char *s1, const char *s2);* сравнить две строки. Возвращаемое значение меньше 0, если s1 лексикографически (алфавитно) предшествует s2. Возвращается 0, если s1=s2, и возвращается число больше нуля, если s1 больше s2.
strncmp	*int strncmp(const char *s1, const char *s2, size_t n);* сравнить две строки, учитывая не более n первых символов
stricmp	*int stricmp(const char *s1, const char *s2);* сравнить две строки, считая латинские символы нижнего и верхнего регистров эквивалентными
strnicmp	*int strnicmp(const char *s1, const char *s2, size_t n);* сравнить две строки по первым n символам, считая латинские символы нижнего и верхнего регистров эквивалентными
strcpy	*char *strcpy(char *dest, const char *src);* копировать строку src в dest, включая завершающий нулевой байт
strncpy	*char *strncpy(char *dest, const char *src, size_t n);* копировать не более n символов из src в dest
strdup	*char *strdup(const char *s);* дублирование строки с выделением памяти
strlen	*size_t strlen(const char *s);* возвращает длину строки в символах
strlwr	*char *strlwr(char *s);* преобразовать строку в нижний регистр (строчные буквы)
strupr	*char *strupr(char *s);* преобразовать строку в верхний регистр (заглавные буквы)
strrev	*char *strrev(char *s);* инвертировать (перевернуть) строку
strnset	*char *strnset(char *s, int ch, size_t n);* установить n символов строки s в заданное значение ch
strset	*char *strset(char *s, int ch);* установить все символы строки s в заданное значение ch
strspn	*size_t strspn(const char *s1, const char *s2);* ищет начальный сегмент s1, целиком состоящий из символов s2. Вернет номер позиции, с которой строки различаются.
strcspn	*size_t strspn(const char *s1, const char *s2);* ищет начальный сегмент s1, целиком состоящий из символов, НЕ входящих в s2. Вернет номер позиции, с которой строки различаются.
strstr	*char *strstr(const char *s1, const char *s2);* найти первую подстановку строки s2 в s1 и вернуть указатель на найденное
strtok	*char *strtok(char *s1, const char *s2);* найти следующий разделитель из набора s2 в строке s1
strerror	*char *strerror(int errnum);* сформировать в строке сообщение об ошибке,

Функция	Описание
	состоящее из двух частей: системной диагностики и необязательного добавочного пользовательского сообщения

При вводе строк с клавиатуры и выводе на экран можно пользоваться стандартными средствами форматного (*п. 1.6*) и потокового (*п. 1.7*) ввода-вывода. Альтернативой операторам *извлечения из потока (>>)* и *вставки в поток (<<)* могут выступать функции *gets( )* и *puts( )* потокового ввода-вывода (*Листинг 64*).

Отличия состоят в том, что оператор *std::cin >> st* вводит с клавиатуры только одно слово из строки, а функция *gets(st)* вводит всю строку, даже если она содержит пробелы, табуляцию и другие служебные символы.

Листинг 64

```
#include "stdafx.h"
#include <iostream>
void _tmain(int argc, char* argv[])
{     char st[21]; // массив под 20 символов + 1 символ конца строки '\0'
      gets(st); // ввод строки с клавиатуры
      puts(st); // вывод строки на экран
      cin.get();
}
```

Пример: дублирование строки с выделением памяти:

```
char *dup_str, *string = "abcde";
dup_str = strdup(string);
```

Пример (*Листинг 65*) иллюстрирует действие функции *strtok( )*, позволяющей разбивать строку на лексемы:

Листинг 65

```
char input[16] = "abc,d";
char *p;
/* strtok() помещает нулевой байт вместо разделителя лексем, если поиск был
успешен */
p = strtok(input, ",");
printf("%s\n", p); //будет выведено "abc";
/* второй вызов использует NULL как первый параметр и возвращает указатель
на следующую лексему */
p = strtok(NULL, ",");
printf("%s\n", p); //будет выведено "d"
```

Разумеется, число выделяемых лексем и набор разделителей могут быть любыми. В коде, приведенном ниже (*Листинг 66*), лексемы могут разделяться пробелом, запятой или горизонтальной табуляцией, а прием и синтаксический разбор строк завершается при вводе пустой строки.

Листинг 66

```
#include <string.h>
#include <stdio.h>
#define BLANK_STRING ""

void _tmain(int argc, char* argv[]))
{     setlocale (LC_ALL, "RUS");
      char *token, buf[81],*separators = "\t,. ";
      int i;
      puts("Вводите строки\n\Завершение - нажатие ENTER.\n");
      while(strcmp(gets(buf),BLANK_STRING)!=0)
      {     i = 0;
```

```
            token = strtok(buf, separators);
            while(token != NULL)
            {     printf("Лексема %d - %s\n", i, token);
                  token = strtok(NULL, separators);
                  i++;
            }
      }
}
```

```
What is you name, Susy?
Лексема 0 - What
Лексема 1 - is
Лексема 2 - you
Лексема 3 - name
Лексема 4 - Susy
```

Рис. 44. Разделение строки на лексемы

6.4. Функции преобразования типов

Описанные ниже функции (*Таблица 14*) объявлены в стандартной библиотеке *stdlib.h*. Прототип функции *atof* содержится, кроме того, в файле *math.h*.

Таблица 14 Функции преобразования типов

Функция	Описание
atof	*double atof(const char *s);* преобразование строки, в представляемое ей число типа double. На переполнение возвращает плюс или минус HUGE_VAL (константа из библиотеки)
atoi	*int atoi(const char *s);* преобразование строки в число типа int (целое). При неудачном преобразовании вернет 0
atol	*long atol(const char *s);* преобразование строки в число типа long (длинное целое)
ecvt	*char *ecvt(double value, int ndig, int *dec, int *sign);* преобразование числа типа double в строку. ndig - требуемая длина строки, dec возвращает положение десятичной точки от 1-й цифры числа, sign возвращает знак
fcvt	*char *fcvt(double value, int ndig, int *dec, int *sign);* преобразование числа типа double в строку. В отличие от ecvt, dec возвращает количество цифр после десятичной точки. Если это количество превышает ndig, происходит округление до ndig знаков.
gcvt	*char *gcvt(double value, int ndig, char *buf);* преобразование числа типа double в строку buf. Параметр ndig определяет требуемое число цифр в записи числа.
itoa	*char *itoa (int value, char *string, int radix);* преобразование числа типа int в строку, записанную в системе счисления с основанием radix (от 2 до 36 включительно)
ltoa	*char *ltoa (long value, char *string, int radix);* преобразование числа типа long в строку
ultoa	*char *ultoa (unsigned long value, char *string, int radix);* преобразование числа типа unsigned long в строку

Рассмотрим функции *sizeof( )* – возвращает размер объекта в байтах и *strlen( )* – определяет длину строки числом символов. Рассмотрим отличия этих функций на примере:

Листинг 67

```
#include "stdafx.h"
#include <iostream>
using namespace std;
#define string2 "This is test string."
```

```
int _tmain(int argc, char* argv[])
{     setlocale (LC_ALL, "Russian");

      char string1[50];

printf("Введите ваше имя: "); scanf("%s", string1);
printf("Приятно познакомиться, уважаемый %s.\n" , string1);
printf("\nВведенная строка %d символов и занимает %d ячеек памяти.\n",
      strlen(string1), sizeof(string1));

printf("Тестовая строка: %s\n" , string2);
printf("Тестовая строка %d символов и занимает %d ячеек памяти.\n",
      strlen(string2), sizeof(string2));

      system("pause");
      return 0;
}
```

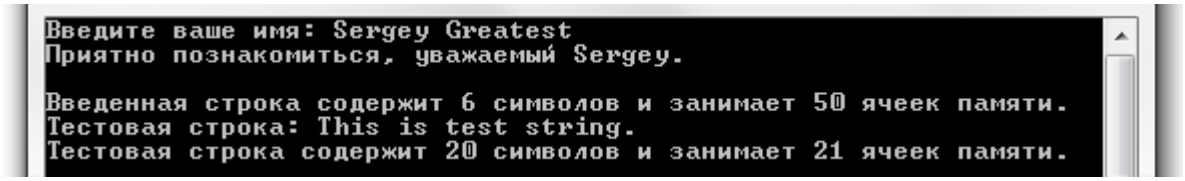

Рис. 45. Отличие функций sizeof и strlen

Из примера (*Листинг 67*) и рисунка (*Рис. 45*) видно, что функция *sizeof( )* возвращает размер объекта в байтах: даже если строка *string1* заполнена не полностью, под нее выделено 50 байт, а *strlen( )* возвращает количество элементов (*символов*) до первого встретившегося символа "\0" – конца строки.

Вы заметили, что функция *scanf( )* считала только первое слово, а не всю строку?

6.5. Практическая работа № 7. Строки.

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Научиться работать со строками. Освоить способ динамического захвата и освобождения памяти под символьные массивы. Уяснить разницу между понятиями «размер массива символов» и «длина строки» при помощи функций *sizeof* и *strlen*. Освоить функции библиотеки «string.h». Научится переводить числовые данные в строковые и обратно.

Задание на практическую работу

1. Написать, отладить и протестировать программу, выполняющую операции с массивами символов в соответствии со своим вариантом (*Таблица 15*), используя при этом функции библиотеки «string.h».
2. Продемонстрировать выделение памяти под статические и динамические строковые переменные, а также корректное удаление динамических массивов символьного типа.
3. В тестовых строках обязательно должны содержаться те символы и подстроки, работа с которыми предусмотрена в задании. Переменные X, Y и Z – вещественного типа.
4. Производимые над строками операции оформить в виде подпрограмм. В теле подпрограмм допускается использование функций стандартных библиотек работы со строками «string.h» и т.п.
5. При демонстрации преподавателю работающей программы необходимо провести пошаговую трассировку указанного преподавателем участка программы. При

трассировке адреса и значения строковых переменных необходимо выводить в окне *Watch*.

6. Результат работы программы выводить на экран и сохранять в виде скриншотов для отчета. По выполненной лабораторной работе подготовить отчет с результатами выполнения работы. Отчет без комментариев не принимается.

Таблица 15 Варианты индивидуальных заданий

№	Операции со строками	Операции с данными
1.	Найти в строке все подстроки, начинающиеся с большой буквы и определить их позиции, последнюю такую подстроку заменить на слово «Кукушечка».	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 15 позициях: X= sin(3*Y)+16.5.
2.	Найти в строке все цифры, определить их позиции и заменить их на текстовые представления: 1 – на «один», 2 – на «два» и т.д. Подсчитать количество замен.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 5 позициях после точки: X= Y-3.0/Z.
3.	Каждый символ «@» в строке заменить на его номер позиции в строке. Подсчитать количество замен.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 3 позициях до и 4 после точки: X= Y+3*Z.
4.	Определить порядковый номер последней буквы «а» в строке. Поиск осуществлять с учетом регистра. Заменить ее на «А».	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 12 позициях: X= cos(Y-3*Z).
5.	В строке найти количество встречающихся рядом символов «соседних по алфавиту» - «а» и «б», «Ю» и «Я» и т.п. Дописать в конец строки слово «Ботаник».	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 3 позициях после точки: X= ln(Y+0.2*Z).
6.	Вывести на экран «Да» или «Нет» в зависимости от того, найдена ли в строке числовая подстрока. Заменить все числа на нули.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 4 позициях до и 4 после точки: X= Y*exp(-3*Z).
7.	В части строки, от начала и до первого символа «@» определить все вхождения символа «А» и заменить их на подстроку «А-а-а-а!». Подсчитать число замен.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 10 позициях: X= Y+3*exp(Z).
8.	В строке заменить все символы «$» на подстроки «1 замена $», «2 замена $» и т.д. в конце строки дописать «Долой буржуйские $!»	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 6 позициях после точки: X= Y+3*tg(Z).
9.	Дописать в начало строки количество вхождений подстроки «Опера» от начала исходной строки до первого вхождения символа «$».	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 2 позициях до и 8 после точки: X= pow(Y,-3*Z).
10.	В части строки, начинающейся символом «#» до конца строки определить все вхождения символа «%» и заменить их на «процент». Подсчитать число замен.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 11 позициях: X= fabs(-Y-3*Z).

№	Операции со строками	Операции с данными
11.	Удалить из введенной строки все символы, отличные от прописных (заглавных) латинских букв и пробелов. Подсчитать число замен.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 9 позициях после точки: X= sqrt(Y^2+Z^2).
12.	После каждого вхождения символа «$» в строке вставлять подстроку «Поставьте мне зачет, пожалуйста!» Подсчитать число вставок.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 4 позициях до и 6 после точки: X= floor(Y-Z).
13.	Во введенной строке заменить каждую точку и запятую на точку с запятой, подсчитать число замен и дописать его в конец строки.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 12 позициях: X= fmod(17,floor(3*Z)).
14.	После каждого слова из введенного предложения вставить число букв в этом слове. Подсчитать количество вставок.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 7 позициях после точки: X= Y^{-2*Z}.
15.	Перед каждым вхождением символа «&» в строке вставить подстроку «Вот он, этот символ, нашелся:». Подсчитать число вставок.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 4 позициях до и 5 после точки: X= lg(Y+3*Z).
16.	Дописать в конец строки список всех знаков препинания, встретившихся в строке и номера их позиций.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 13 позициях: X= pow(13*Y, -13*Z).
17.	Удалить из строки все вхождения подстроки «5» или «4» заменить их на подстроки «Неудовлетворительно». Последнее вхождение заменить на «Ладно, тройка!»	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 8 позициях после точки: X= sin(Y^2)+3*cos(Z).
18.	Все слова введенного предложения взять в кавычки. Подсчитать число слов и дописать в начало строки текст «Было произведено N вставок».	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 3 позициях до и 5 после точки: X= log10(Y+3*Z)
19.	В подстроке, начинающейся символом «&» до конца строки определить все вхождения символов-цифр и заменить их на «100500». Подсчитать число замен.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 14 позициях: X= 2*log(Y,3*Z) – 2*ln(3*Z).
20.	В подстроке между первым и последним вхождением символа «#» заменить все строчные (маленькие) буквы на прописные (заглавные) и наоборот.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 4 позициях после точки: X= Y+exp(2*Z).
21.	Дописать в начало строки количество вхождений подстроки «Аля-улю» в исходной строке.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 6 позициях до и 2 после точки: X= |Y+3.14/Z|.
22.	Все слова во введенном предложении переставить в обратном порядке, порядок букв в словах не менять. Дописать в конец строки подстроку «Да пребудет с вами сила!».	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 9 позициях: X= exp(Y)-ln(4*Z).

№	Операции со строками	Операции с данными
23.	В части строки, от начала и до последнего символа «%» определить количество вхождений подстроки «Ёпрст» и дописать в конец строки это количество.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 7 позициях после точки: X= floor(Y+Z^3).
24.	После каждой последовательности одинаковых символов в строке вставить в круглых скобках количество повторяющихся символов.	Задать переменные, преобразовать выражение в строку, значения переменных выводить в 9 позициях до и 9 после точки: X= sqrt(Y+3*Z^2).

Контрольные вопросы:

1. Что представляет собой символьная строка?
2. Как используются стандартные функции форматного и потокового ввода-вывода в работе со строками?
3. Размер символьного массива и длина строки – это одно и то же?
4. Как обозначается конец строки, для чего он нужен?
5. Какие операторы и функции служат для освобождения памяти, занятой динамическим символьным массивом?
6. Какие стандартные библиотеки для работы со строками вы знаете? Можно ли программировать работу со строками без них?
7. Как производится присваивание (копирование) строковых переменных?
8. Может ли строка символов быть статической или динамической?
9. Как определить количество символов в массиве? В строке?
10. Какие символы строки будут выводиться на экран стандартными подпрограммами ввода-вывода, а какие – нет?
11. Можно ли обращаться к элементам строки при помощи оператора «*квадратные скобки*», а по технологии «*адрес + смещение*»?
12. Какие операторы и функции служат для выделения памяти под динамический символьный массив?
13. Что такое адрес строки? Как сравнить две строковых переменных?
14. Какие функции служат для перевода числа в символьную строку, какие – для перевода символьной строки в число? Что возвращается, если перевод сделать не удалось?

7. Работа с файлами

```
Жили-были у программиста два сына.
И звали их: НовоеИмя1 и НовоеИмя2.
```

Большинство компьютерных программ работают с файлами, и поэтому возникает необходимость создавать, удалять, записывать читать, открывать файлы [3, 4]. *Файл* – это именованный набор байтов, который может быть сохранен на некотором накопителе. Под файлом понимается некоторая последовательность байтов, которая имеет своё, уникальное имя, например *файл.txt*. Полное имя файла – это полный путь к директории файла с указанием имени файла, например: *D:\\docs\\file.txt.*

Во всех файлах на компьютере хранятся только *двоичные данные*. Видеоинформация, аудиоданные, текст, графика, программы – все это лишь нолики и единички, хранящиеся в дисковой или оперативной памяти в определенном формате. Как компьютерные программы получают информацию о том, что можно с тем или иным файлом делать? Какой файл требуется открывать в графическом редакторе, а какой – показывать в виде фильма? Большинство файлов имеет *расширение,* указывающее, как будут интерпретированы хранящиеся в нем двоичные данные.

Программирование на C++ позволяет получить доступ к файлам с любым расширением – работать с данными любого вида. Однако, на данном уровне обучения мы ограничимся работой с текстовыми файлами **.txt.*

С точки зрения программирования, работа с файлами складывается из трех шагов.

1. *Открытие файла*. Получение пользовательской программой у операционной системы прав для работы с заданным *по имени* файлом. Данная операция проделывается, для исключения конфликтов, при которых несколько программ одновременно пытаются записывать информацию в один и тот же файл. Правда, считывать данные из файла допустимо одновременно множеством программ, поэтому в операции открытия файла обычно уточняется, что файл открывается *"для чтения"* (считывание информации, которая не меняется) либо *"для записи"* (данные в файле модифицируются). Операция открытия файла возвращает некий идентификатор (как правило, целое число), которое однозначно определяет в программе в дальнейшем нужный открытый файл. Этот идентификатор запоминается в переменной; обычно такая переменная называется *файловой переменной*.

2. *Работа с файлом.* Считывание данных из файла, либо запись в него.

3. *Закрытие файла*. После этой операции файл снова доступен другим программам для обработки.

7.1. Файловые операции библиотеки <stdio>

Стандартные функции работы с файлами, существующие практически во всех реализациях языка C. Одна из библиотек, предлагающих подпрограммы работы с файлами - *stdio.h*.

Функция открытия файла называется *fopen( ).* Она возвращает указатель на объект стандартного (определенного в данной библиотеке) типа *FILE*. Функция *fopen( )* имеет два аргумента – *путь_к_файлу* (строка) и *параметры_открытия_файла*. Параметр открытия файла на запись в текстовом виде записывается строкой *"wt"*. Буква *w* означает *write*, буква *t* – *text*. Если указанный файл в каталоге существует, он будет перезаписан (все старое содержимое в нем уничтожится), если он не существует, то будет создан исходно пустым.

```
FILE *F, *fo; // указатель на файловую переменную
F = fopen("test.txt","wt"); // открыть файл для записи
```

После открытия файла в файловую переменную *F* занесется некоторое число. Если таким числом будет ноль, считается, что файл открыть не удалось. В языке *C* нередки записи вида:

```
if((fo=fopen("c:\\tmp\\test.txt","wt")) == 0 )
{  cout << "файл открыть не удалось" << endl; } // сообщение об ошибке
```

где одновременно открывается файл и проверяется, успешно ли это сделано.

Закрывается файл с помощью функции *fclose( )*:

```
fclose(F);
```

После закрытия файла к файловой переменной обращаться уже нельзя.

Запись в файл, как и при выводе на экран возможна при помощи потоковых и форматных подпрограмм и операторов.

Форматная запись текстовой строки в файл выполняется функцией *fprintf( )*:

```
fprintf(имя_файловой_переменной, формат, список_выводимых_переменных);
```

Здесь, как и в *п. 1.6*, *формат* – это текстовая строка, задающая формат записываемого в файл текста. Все, что содержится в этой строке, пишется в файл один-в-один. Чтобы вывести значение некоторой переменной, надо использовать элемент формата, который начинается с символа %. Так, формат *%d* задает вывод целочисленного значения, формат *%s* - вывод строки и т.д. (*Таблица 3*).

```
int n = 10;
char str[20] = "значение переменной n равно ";
fprintf( F, "Вывод: %s %d", str, n );
```

В файл запишется строка "*Вывод: значение переменной n равно 10*". Вместо формата *%s* подставлено значение переменной *str*, а вместо формата *%d* - значение переменной *n*.

Для вывода дробных чисел используют формат *%f*, где перед *f* ставят число всех выводимых символов и число символов после запятой, разделенных точкой. Например: *"%5.2f"* означает вывод числа из пяти позиций, причем две из них будут отведены для дробного значения. Если оно не уместится в этих позициях, то вылезет за границу, а если займет меньше, то дополнится пробелами.

Листинг 68

```
FILE * fo;
fo = fopen("test.txt","wt");
int i;
for( i=0; i<100; i++ ) // записать в файл 100 строк с числами
{       fprintf( fo, "%d\n", i ); }
fclose(fo);
```

Для ввода данных (текстовой строки) используют функцию *fscanf( )*:

```
fscanf( файловая_переменная, формат_ввода, список_адресов_переменных )
```

При вводе *список_адресов_переменных* отличается от списка имен переменных. Если в *C++* указывается имя переменной, то подразумевается значение этой переменной. Но при вводе нужно не значение, а местонахождение области в памяти, отведенной для заданной переменной – чтобы в эту область поместить новое значение,

считанное из файла. Операция *& «взятие адреса переменной»,* примененная к имени переменной, возвращает адрес этой переменной в памяти. Тогда операцию ввода числа из текстового файла *F* в целочисленную переменную *n* можно записать так:

```
fscanf( F, "%d", &n );
```

Полезная функция *feof(файловая_переменная)* возвращает 1 (истинное значение), если файл, открытый для считывания, закончился. Она возвращает 0, если из файла еще можно продолжать ввод.

Листинг 69

```
FILE * fi;
fi = fopen("test.txt","rt"); // rt означает открытие файла для чтения
int n;
while( !feof(fi) ) // считывание из файла всех строк с числами
{       fscanf( fi, "%d", &n ); }
fclose(fi);
```

7.2. Работа с файлами посредством библиотеки <fstream>

Работа с файлами организована аналогично работе с другими потоками ввода-вывода. В *C++* так удобно созданы подпрограммы, что для программиста по большей части не видно разницы между программированием процесса ввода данных из файла или с клавиатуры, процесса вывода данных в файл или на экран.

Потоковый способ работы с файлами организован в виде *иерархии классов* – подробно работу с *классами* мы будем разбирать позднее в *главах 10-11*, сейчас же важно уловить основные принципы работы с объектами файловых классов.

Для работы с файлами необходимо подключить заголовочный файл <*fstream*> библиотеки потоковой работы с файлами. В библиотеке <*fstream*> определены *два класса:* <*ifstream*> для файлового ввода и <*ofstream*> – для файлового вывода.

Для потоковой работы с файлами необходимо проделать следующие шаги:
1. Создать объект класса *ofstream* (или *ifstream*);
2. Связать объект класса с файлом операционной системы (открыть файл);
3. Записать информацию в файл/ или прочитать из него;
4. Закрыть файл.

```
ofstream fout; // создаём объект класса ofstream для записи в файл
fout.open("cpp_test.txt"); // связываем объект с именем файла на диске
fout.open("cpp_test.txt", ios_base::out); // или так
```

Здесь операция *«точка»* обеспечивает доступ к методу *open( )* класса, в круглых скобках указывается имя файла (более подробно о ней в *главах 8, 10-11*). Указанный файл будет создан в текущем каталоге (там же где лежит и основная программа). Если файл с таким именем существует, то существующий файл будет заменён новым.

Параметр *ios_base::out* относится к флагам открытия файла, они представлены в сводной таблице флагов ниже (*Таблица 16*).

```
fout << "Работа с файлами в C++"; // запись строки в файл оператором <<
fout.close(); // закрываем файл
```

Здесь для записи строки в файл используется оператор << - *передача в поток* объекта *fout*. После чего файл закрывается при помощи метода *close( )*.

В одной строке можно создать объект и сразу же связать его с файлом при помощи *конструктора* (более подробно в *главе 10*). Делается это так:

```
ofstream fout("cpp_test.txt", ios_base::app);
// создаём объект класса ofstream и сразу же связываем его с файлом
```

Чтение данных из файла производится аналогично записи в файл, только используются при этом объекты класса вывода *ifstream*. Параметр режима открытия файла (*Таблица 16*) можно не указывать – он устанавливается по умолчанию.

Листинг 70

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

int _tmain(int argc, char* argv[ ])
{    setlocale(LC_ALL, "rus"); // корректное отображение Кириллицы
     char buff[50]; // буфер для хранения считываемого из файла текста

     ifstream fin("cpp_tst.txt"); // открыли файл для чтения
     if(fin.is_open()) // а он открылся?
     {    fin >> buff; // считать первое слово из файла
          cout << buff << endl; // напечатать это слово
          fin.getline(buff, 50); // считать строку из файла
          fin.close(); // закрыть файл
          cout << buff << endl; // напечатать строку
     }
     else
          cout << "файл открыть не удалось" << endl;

     cin.get();
     return 0;
}
```

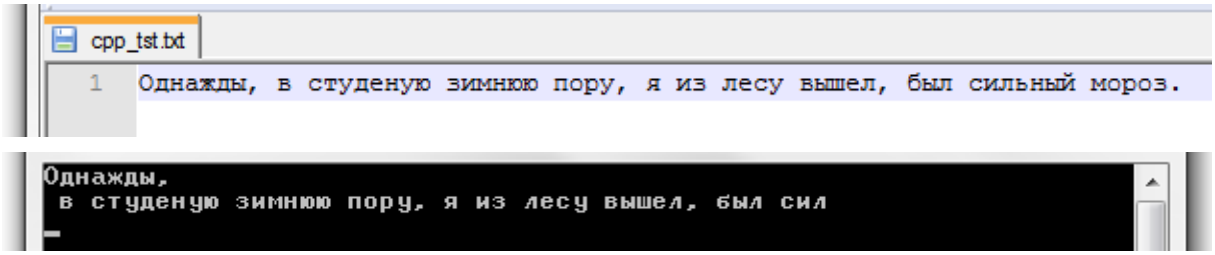

Рис. 46. Содержимое файла cpp_tst.txt (а) и результаты работы программы (б)

В программе (*Листинг 70*) показаны два способа чтения из файла, первый – используя операцию >> *передачи в поток*, второй – используя *метод*-функцию *getline( )*. В первом случае считывается только одно слово, а во втором случае считывается строка, в данном случае – длинной не более 50 символов. Если в файле осталось меньше 50 символов, то считываются символы до последнего включительно, если символов больше – как в примере (*Рис. 46*), то будет считано ровно 50, т.е. считываемая строка обрезается. Обратите внимание на то, что вторая операция считывания продолжилась после первого слова, а не с начала строки, так как первое слово уже было прочитано ранее.

В библиотеке *fstream* предусмотрена функция *is_open( )*, которая возвращает целые значения: 1 – если файл был успешно открыт, 0 – если файл открыт не был (*Листинг 70*).

Режимы открытия файлов

Режимы открытия файлов устанавливают характер использования файлов. Для установки режима в классе *ios_base* предусмотрены константы - флаги, которые определяют режим открытия файлов (*см. Таблица 16*).

Таблица 16 Флаги открытия файлов

Константа	Описание
ios_base::in	открыть файл для чтения
ios_base::out	открыть файл для записи
ios_base::ate	при открытии переместить указатель в конец файла
ios_base::app	открыть файл для записи в конец файла
ios_base::trunc	удалить содержимое файла, если он существует
ios_base::binary	открытие файла в двоичном режиме

Режимы открытия файлов можно устанавливать непосредственно при создании объекта или при вызове функции *open( )*.

```
ofstream fout("cppstudio.txt", ios_base::app);
// открываем файл для добавления информации к концу файла
fout.open("cppstudio.txt", ios_base::out);
// открываем файл для перезаписи файла
```

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции *«или»* |, например: *ios_base::out* | *ios_base::trunc* – открытие файла для записи, предварительно очистив его.

В примере (*Листинг 71*) приведена функция производящая подсчет числа строк в текстовом файле

Листинг 71

```
int string_count(const char *file_name)
{       ifstream fin(file_name, ios_base::in);
        if (fin.is_open()) // если файл открыт
        {       int i= 0;
                char *s= new char[255]; // строка для чтения
                while (!fin.eof()) // пока не конец файла
                {       fin.getline(s, 254);
                        i++; }
                fin.close();
                delete []s;
                return i;
        } return 0;
}
```

7.3. Лабораторная работа № 8. Работа с файлами.

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Научиться работать с файловыми переменными посредствам подпрограмм библиотек *fstream.h* и *stdio.h*. Закрепить навыки работы со строками. Научиться открывать файлы для записи и чтения, создавать и уничтожать файлы, определять конец файла.

Задание на лабораторную работу

1. Написать, отладить и протестировать программу, выполняющую операции с текстовыми файлами в соответствии со своим вариантом (*Таблица 17*). Продемонстрировать использование при этом файловых функций из библиотек «*fstream.h*» и «*stdio.h*».

2. Подобрать текст размером не менее 30 строк, в тексте обязательно должны содержаться те символы и подстроки, работа с которыми предусмотрена в индивидуальном задании.

3. Результат работы программы выводить на экран и обязательно сохранять в файл результатов.

4. Производимые над строками текстового файла операции оформить в виде подпрограмм. В теле подпрограмм допускается использование функций стандартных библиотек работы со строками «*string.h*» и т.п.

5. По выполненной лабораторной работе подготовить отчет с результатами работы программы. В отчет вставить листинг программного кода, исходный файл и файл результатов. Отчет без комментариев не принимается.

Таблица 17 Варианты индивидуальных заданий

№	Задание
1.	Определить количество вхождений символов «Я» и «я» в тексте, заменить эти символы на «Ты» и «ты» соответственно.
2.	Препроцессинг китайского: Записать строки текста по столбцам сверху вниз. Можно использовать двумерные массивы.
3.	Подсчитать количество вхождений всех символов в текст, выводя каждый символ, встретившийся в тексте, и количество его вхождений в текст.
4.	Отсортировать символы в строках «по возрастанию», а после отсортировать строки в тексте «по убыванию». Строки сравнивать по длине.
5.	Отсортировать символы в строках «по убыванию». Удалить из строк все символы отличные от букв и цифр.
6.	В конец каждой строки текстового файла дописать количество символов этой строки. Заменить все заглавные буквы строчными.
7.	Заменить все символы текста соответствующими символами кода Морзе. Код считать из файла. Результат вывести в файл.
8.	Прочесть из файла числовой массив, в который занести количество букв и цифр в каждой строке. Результат вывести в файл.
9.	Заменить все буквы в строках текста, прочитанного из файла, их кодами, при этом знаки препинания и цифры не заменять.
10.	Расположить строки текста, прочитанного из файла, «по возрастанию». Удалить из текста цифры. Результат вывести в файл.
11.	Осуществить перевод всех символов текста в верхний регистр, а цифры и знаки препинания удалить. Результат вывести в файл.
12.	Препроцессинг иврита: Записать символы строк текста, прочитанного из файла, в обратном порядке, заменяя все вхождения строк «да» на строки «таки да».
13.	Оставить в тексте только цифры, заменив все буквы символами «*», а знаки препинания – символами «@». Результат вывести в файл.
14.	Из заданного текстового файла сгенерировать три: в одном – только буквы, в другом – цифры, в третьем – все остальные символы.
15.	Подсчитать количество пробелов в каждой строке и всего в тексте. После этого удалить пробелы. Результаты вывести в файл.

№	Задание
16.	Каждое слово в тексте записать в обратном порядке. Последовательность слов не менять. Результат вывести в файл.
17.	Препроцессинг женской логики: Заменить в тексте все вхождения слов «да» и «нет» на «может быть». Все имена (с заглавной буквы) на слово «Зая».
18.	В конец каждой строки дописать, содержит ли строка символы, отличные от латинских букв и сколько таковых. Результат вывести в файл.
19.	Буквы, расположенные в четных позициях, перевести в прописные (заглавные), а те, которые на нечетных – в строчные (маленькие).
20.	Все цифры в тексте заменить их строчным написанием: 1 - «один», 2 – «два» и т.д. Результат вывести в файл.
21.	Подсчитать количество слов в каждой строке и всего в тексте. Результат вывести в файл. Количество слов для каждой строки записывать в начало строки.
22.	В конец каждой строки дописать позицию первого вхождения символа «А» или «а» в эту строку. Результат вывести в файл.
23.	Препроцессинг телеграфа: перевести все символы в верхний регистр и все знаки препинания в тексте заменить на соответствующие обозначения: символ «.» - на «ТЧК», символ «,» - на «ЗПТ» и т.д.
24.	Заменить все пробелы в тексте, прочитанном из файла символами «*подчеркивание*», удалить все знаки препинания. Подсчитать количество замен.

Контрольные вопросы:

1. Что такое текстовый файл? Опишите последовательность записи строки в файл потоковым способом.

2. Что представляют собой файловые переменные типа *FILE* из библиотеки <*stdio.h*> как с ними работать? Какое им присваивается значение?

3. С точки зрения механизма размещения двоичных данных в памяти компьютера имеются ли особенности хранения файлов различного назначения?

4. Опишите логику последовательности чтения отдельного слова из файла для файловых переменных (*FILE*) из библиотеки <*stdio.h*>.

5. Что представляют собой объекты файлового ввода-вывода <*ifstream*> и <*ofstream*> из библиотеки <*iostream*>? Как с ними работать?

6. Опишите последовательность записи числа в файл форматным способом, используя переменные типа *FILE* из библиотеки <*stdio.h*>.

7. В чем разница использования метода getline() и оператора >> при потоковом вводе данных из файла – подпрограмм класса <*ifstream*>.

8. По какому признаку можно определить, как будут интерпретированы операционной системой данные, хранящиеся в файле?

9. Как указать для чего – для чтения или записи открывается файл? Объяснение привести для переменных типа *FILE* из библиотеки <*stdio.h*> и объектов потоковых классов.

8. Структуры языка C++

```
- Здравствуйте, Катю можно?
- Она в архиве.
- Разархивируйте ее, пожалуйста,
  она мне срочно нужна!
```

Все современные данные в любых информационных системах представляются в структурированном виде. Мы изучили стандартные типы данных (*п. 1.4*), после этого мы познакомились с массивами – конструкцией из множества *однотипных* ячеек памяти (*гл. 5*), строки, изученные в *гл. 6* – это частный случай массива. Настало время познакомиться с типом данных, позволяющим в одной переменной объединять ячейки памяти *разного типа*.

8.1. Структуры (struct)

В отличие от массива, все элементы которого однотипны, такой *тип данных* как *структура* может содержать элементы разных типов. Элементы структуры называются *полями структуры* и могут иметь любой тип, кроме типа этой же структуры (но могут быть указателями на него).

Программист сам конструирует, какие поля должны входить в структуру, задаваемую им. Описывается структурный тип одним из следующих способов:

```
struct имя_структуры
{       тип_1 поле_1:
        тип_2 поле_2;
                ...
        тип_n поле_n;
};
```

```
struct
{       тип_1 поле_1:
        тип_2 поле_2;
                ...
        тип_n поле_n;
} список_описателей;
```

Если отсутствует *имя_структуры*, то должен быть указан *список_описателей переменных, указателей или массивов*. В этом случае описание структуры служит определением элементов этого списка:

Листинг 72

```
struct // безымянная структура
{       char f1o[30]; // поля структуры
        int date, code;
        double salary;
} staff[100], *ps; // определен массив структур и указатель на структуру
```

В примере (*Листинг 72*) структурный тип не имеет имени, однако определены две переменные вновь созданного типа: *staff[100]* – массив из 100 объектов структурного типа и **ps* – типизированный указатель на ячейку структурного типа.

Для инициализации структуры, значения ее элементов перечисляют в фигурных скобках в порядке их описания:

Листинг 73

```
struct
{       char fio[30]; // первое поле – символьная строка
        int date, code; // второе и третье поля – целые числа
        double salary; // четвертое поле – вещественное число
} worker = {"Абрамович В. В.", 31, 215, 3400.55 };
```

В примере (*Листинг 73*) задан безымянный структурный тип данных, определена статическая переменная этого типа с именем *worker* и полям переменной присвоены начальные значения. Первое поле с именем *fio* – это символьная строка из

30 элементов, второе поле *date* – целое число, третье поле *code* – тоже целое число, четвертое поле *salary* – вещественной число с двойной точностью.

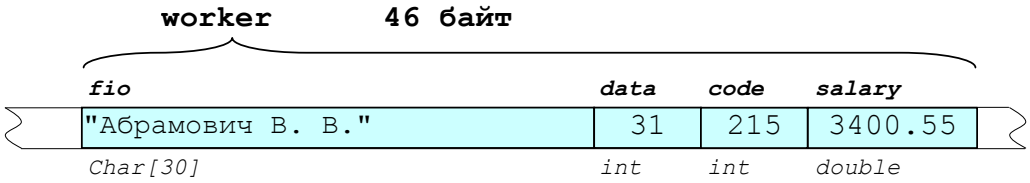

Рис. 47. Размещение переменной worker структурного типа в памяти

Как показано на рисунке (*Рис. 47*), поля структуры расположены по порядку один за другим. Адрес первого поля структурной переменной совпадает с адресом самой переменной. Размер структуры складывается из размеров ее полей.

Чаще структурному типу присваивается имя (*см. Листинг 74*), тогда описание структуры определяет *новый тип*, имя которого можно использовать в дальнейшем наряду со стандартными типами, например:

Листинг 74

```
struct WorkerType // описание нового типа WorkerType
{       char fio[30];
        int date, code;
        double salary;
}; // описание заканчивается точкой с запятой

WorkerType staff[100], *ps; // определение переменных типа WorkerType
```

Доступ к полям структуры выполняется для статических переменных с помощью *оператора выбора* «**.**» (точка) при обращении к полю через имя структуры. Если же мы имеем дело с указателем на структуру, то для доступа используется *оператор выбора* «**->**» (стрелочка), например:

Листинг 75

```
WorkerType worker, staff[100], *ps;
strcpy(worker.fio, "Медведевский Д. А."); // копирование строки в поле fio
staff[8].code = 216; // обращение к полю статической переменной

ps= &worker; // указателю ps присвоен адрес статической переменной
ps->salary = 0.12; // обращение к полям через указатель
cout << ps->fio << endl;

WorkerType* nps= new WorkerType; // nps – адрес динамической переменной
strcpy(nps->fio, "Чебурашкин К. Г."); // к полям динамической переменной
nps->date= 0;
nps->code= worker.code;
nps->salary= 123.456;
```

При начальной инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива:

Листинг 76

```
struct complex
{       float real, im;
} compl[2][3] =
{ { {0, 0}, {0, 1}, {0, 2} },   // строка 1, то есть массив compl[0]
  { {1, 0}, {1, 1}, {1, 2} } }; // строка 2, то есть массив compl[1]
```

Для переменных одного и того же структурного типа определена операция *присваивания,* при этом происходит поэлементное копирование. Структуру можно передавать в функцию и возвращать в качестве значения функции.

Если элементом структуры является другая структура, то доступ к ее элементам выполняется через две операции выбора:

Листинг 77
```
struct A {int a; double x;};
struct B {A a; double x;} x[2];
x[0].a.a = 1;
x[1].x = 0.1;
```

Как видно из примера (*Листинг 77*), поля разных структур могут иметь одинаковые имена, поскольку у них разная область видимости.

8.2. Битовые поля

Битовые поля – это особый вид полей *структуры*, которые используются для плотной упаковки данных, например, логических флагов со значениями «*false/true*». В памяти компьютера минимальная адресуемая ячейка памяти – 1 байт, а для хранения флага достаточно одного бита. При описании битового поля после имени через двоеточие указывается *величина поля в битах* (целая положительная константа), см. *Листинг 78.*

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Размер любой переменной, в том числе и структуры, включающей в себя битовые поля, выравнивается по конечному числу байт – в нашем примере (*Рис. 48*) переменная *MyOption* дополнена до 2 байт.

Листинг 78 (часть 1)
```
#include "stdafx.h"
#include <iostream>
using namespace std;

struct Options // описание структуры с битовыми полями
{       bool centerX:1;
        bool centerY:1;
        unsigned int shadow:5;
        unsigned int palette:4;
};

// ---------------------------------------------------------------
// передача переменной типа Option в подпрограмму "по указателю"

void showOption(Options *Opt)
{
      cout << "MyOption.centerX= " << Opt->centerX << endl;
      cout << "MyOption.centerY= " << Opt->centerY << endl;
      cout << "MyOption.shadow= " << Opt->shadow << endl;
      cout << "MyOption.palette= " << Opt->palette << endl << endl;
}
```

Доступ к полю осуществляется обычным способом – по имени. Адрес поля получить нельзя (поскольку у битов адреса не существует), однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры.

Листинг 78 (часть 2)
```
int _tmain(int argc, char* argv[])
{       setlocale (LC_ALL, "Russian");

Options MyOption;

MyOption.centerX= false;
MyOption.centerY= !MyOption.centerX;
```

```
MyOption.shadow= 6;
MyOption.palette= 3;

showOption(&MyOption);

MyOption.shadow= 32; // переполнение разряда поля shadow (5 бит)

showOption(&MyOption);

      cin.get();
      return 0;
}
```

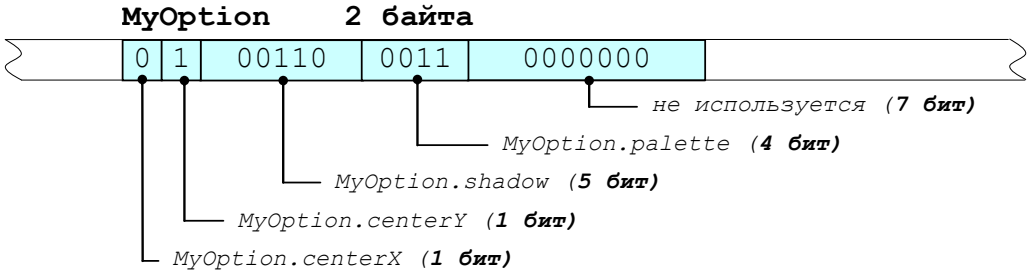

Рис. 48. Размещение переменной MyOption структурного типа Options в памяти

Важно знать, что компилятор не отслеживает выхода за границы битового поля – ответственность за это полностью лежит на программисте. Например, в задаче (*Листинг 78*) поле *shadow* структуры *Options* объявлено как *unsigned int*, а мы знаем, что переменные этого типа изменяются в диапазоне от 0 до 4294967295. Совершенно ясно, что в пятибитное поле *shadow* можно положить число не больше $11111_2=31_{10}$, однако компилятор *не посчитает за ошибку* запись такого вида:

```
MyOption.shadow= 32; // переполнение 5-битного поля!
```

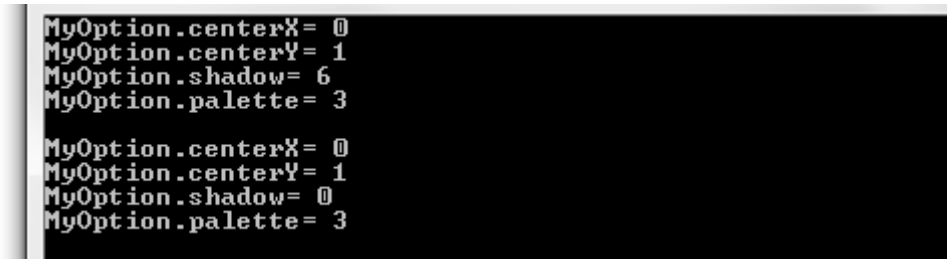

Рис. 49. Переполнение разряда 5-битного поля MyOption.shadow

Следует учитывать, что операции с отдельными битами реализуются гораздо менее эффективно, чем с байтами и словами, так как компилятор должен генерировать специальные коды, и экономия памяти под переменные оборачивается увеличением объема кода программы.

8.3. Объединения (union)

Объединение (union) представляет собой частный случай структуры, все поля которой располагаются *по одному и тому же адресу*. Формат описания такой же, как у структуры, только вместо ключевого слова *struct* используется слово *union*. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в переменной типа объединение хранится только одно значение, и ответственность за его правильное использование лежит на программисте.

Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно не требуется:

Листинг 79

```
int _tmain(int argc, char* argv[])
{       enum paytype { CARD, CHECK };
        paytype ptype;

        union payment
        {       char card[25];
                long check;
        } info;

        switch (ptype) // присваивание значений info и ptype
        {       case CARD: cout << "Оплата по карте: " << info.card; break;
                case CHECK: cout << "Оплата чеком: " << info.check; break;
        }

return 0;
}
```

Объединение часто используют в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Имя объединения можно не указывать, что позволяет обращаться к его полям непосредственно.

Если все поля объединения хранятся по одному адресу, то как же данные, хранящиеся в полях объединения, не смешиваются? А они смешиваются! При изменении значения какого-то поля у переменной типа *union* изменения вносятся и во все другие поля – ведь это же одна и та же ячейка памяти!

Основываясь на этом свойстве, объединения часто применяются для разной интерпретации одного и того же битового представления (но, как правило, и в этом случае лучше использовать явные операции преобразования типов). В качестве примера (*Листинг 80*) рассмотрим работу со структурой, содержащей битовые поля:

Листинг 80 (часть 1)

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

union uByteType
{       unsigned char byte; // обращение к байту целиком
        struct  { // битовые поля для обращения к данным побитово
                bool b0:1;
                bool b1:1;
                bool b2:1;
                bool b3:1;
                bool b4:1;
                bool b5:1;
                bool b6:1;
                bool b7:1;
                        } bit; // переменная типа битовое поле
};
```

Нетривиальная конструкция *uByteType* представляет собой объединение в одной ячейке двух типов данных (полей) – однобайтовой переменной (*unsigned char*) *byte* и другой структуры (*битового поля*) *bit*.

```
void show(uByteType *B)
{   printf("\nB.byte= %d B.bit= (%d%d%d%d%d%d%d%d)", B->byte, B->bit.b7,
    B->bit.b6,B->bit.b5,B->bit.b4,B->bit.b3,B->bit.b2,B->bit.b1,B->bit.b0);
    printf("\n&B= %p &B.byte= %p &B.bit.b0= %p", B, &(B->byte), &(B->bit));
}
```

На примере программы *void show(uByteType *B)* удобно рассмотреть гибкость

полученной конструкции. Когда нам удобно обращаться к переменной типа *uByteType* как к десятичному числу, мы будем писать *B->byte* и данная однобайтовая переменная **B* будет интерпретироваться как *unsigned char*. Если же нам требуется получить доступ к конкретному биту *B->bit* этой переменной, то мы воспользуемся побитовым обращением *B->bit.b7, ... ,B->bit.b0*.

Листинг 80 (часть 2)

```
int _tmain(int argc, char* argv[])
{       setlocale (LC_ALL, "Russian");

        uByteType B;

        B.byte= 204;
        show(&B);

        B.bit.b3=0;
        show(&B);

        B.byte= 128;
        show(&B);

      cin.get();
      return 0;
}
```

```
B.byte= 204 B.bit= (11001100)
&B= 0026FDB3 &B.byte= 0026FDB3 &B.bit.b0= 0026FDB3
B.byte= 196 B.bit= (11000100)
&B= 0026FDB3 &B.byte= 0026FDB3 &B.bit.b0= 0026FDB3
B.byte= 128 B.bit= (10000000)
&B= 0026FDB3 &B.byte= 0026FDB3 &B.bit.b0= 0026FDB3_
```

Рис. 50. Размещение в памяти переменной типа union

Можно видеть (*Рис. 50*) что адреса переменной *B*, первого поля *B.byte* и второго поля *B.bit* совпадают, это подтверждает тот факт, что все поля объединения хранятся в одной адресной ячейке.

Рисунок (*Рис. 51*) иллюстрирует расположение ячейки *B* типа *uByteType* в памяти. В листинге (*Листинг 80, часть 2*) показывается, что можно обращаться ко всему байту целиком – когда мы пишем *B.byte=204*, или при изменении отдельного бита нашего байта *B*, мы пишем *B.bit.b3=0*. При этом изменяется весь байт.

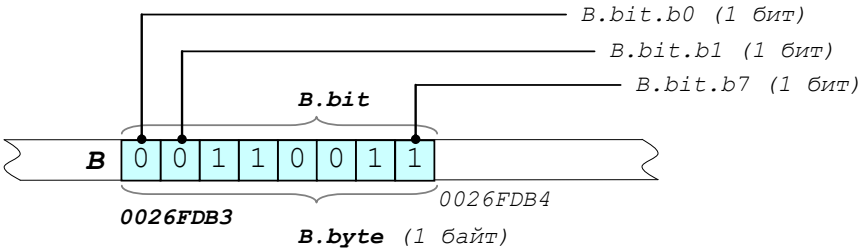

Рис. 51. Размещение в памяти переменной типа union

По сравнению со структурами и классами на объединения налагаются некоторые ограничения:

- объединение может инициализироваться только значением его первого элемента;
- объединение не может содержать битовые поля;
- объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания (*см. гл. 10*);
- объединение не может входить в иерархию классов (*см. гл. 11*).

8.4. Указатели на структуру

Если функции передается большая структура, то эффективнее передать указатель на эту структуру, в крайнем случае – ссылку на нее, но не копировать ее в стек целиком. Работа с указателем на структуру ничем не отличается от работы с указателями на обычные переменные.

Листинг 81

```
struct WorkerType
{       char fio[30];
        int date, code;
        double salary; };

WorkerType sw= {"Ельцинишвили Б. Н.", 67, 217, 0.05 };
WorkerType *ps= &sw; // указателю ps присвоить адрес переменной sw
cout << (*ps).fio << endl;

ps= new WorkerType; // создать динамический объект типа Worker
(*ps).fio= "Горбачавичус М. С.";
ps->code= 218; (*ps).salary= 0.01;
```

Поскольку *ps – указатель* на структуру типа *Worker*, то *содержимое указателя *ps* – это сама структура, а *(*ps).fio, (*ps).code* и *(*ps).salary* – поля структуры. Скобки *(*ps).fio* здесь необходимы, так как приоритет операции (.) выше приоритета операции (*). В случае отсутствия скобок **ps.fio* интерпретируется компилятором как **(ps.fio)*, а это в данном случае лишено смысла.

Операторы доступа к полям структуры (.) и (->) вместе с операторами вызова функции ( ) и индексами массива [ ] занимают самое высокое положение в иерархии приоритетов операций в языке С.

Указатели на структуру используются в следующих случаях:

- доступ к структурам, размещенным в динамической памяти;
- создание сложных структур данных – списков, деревьев;
- передача структур в качестве параметров в функции.

Рассмотрим пример (*Листинг 82*), в котором строится сложная структура данных *TypeWeather*, для хранения данных о погоде на определенный день *TypeDate*, направление ветра указывается при помощи перечисляемого типа *TypeWD*.

В листинге (*Листинг 82, часть 1*) описываются три подпрограммы для работы со структурой такого типа, реализация которых приводится ниже (*Листинг 82, часть 2-3*). Проанализировав главную программу *_tmain()*, можно видеть основной алгоритм работы. Сперва выясняется количество строк в файле данных (*weather.txt*) при помощи программы, аналогичной *(Листинг 71*), затем создается динамический массив *ArrayWeather* соответствующего размера для хранения переменных типа *TypeWeather*. После чего массив *ArrayWeather* заполняется данными из файла и выводится на экран.

Листинг 82 (часть 1)

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

enum TypeWD {N, NW, NE, W, E, S, SW, SE}; // напраление ветра
struct TypeDate { int Day, Month, Year; }; // дата

// ------------------------------------------------------------
struct TypeWeather // структура погода
{       TypeDate Date;                  // дата
        long double Temperature;        // температура
        long double Moisture;           // влажность
        TypeWD WindDirection;           // неправление ветра
```

```
        long double WindPower;          // сила ветра
};
// ---------------------------------------------------------------
int Read(const char*, TypeWeather*);
void ReadWeather(ifstream&, TypeWeather&);
string StringWeather(TypeWeather&);
// ---------------------------------------------------------------
int _tmain(int argc, char* argv[])
{       setlocale (LC_ALL, "Russian");

        int size= string_count("weather.txt");
        TypeWeather *ArrayWeather= new TypeWeather[size];

        int rsize= Read("weather.txt",ArrayWeather);
        while (rsize)
        { cout << StringWeather(ArrayWeather[--rsize]) << endl; }

      cin.get();
      return 0;
}
```

Заполнение массива *ArrayWeather* осуществляется при помощи функции *int Read(const char*, TypeWeather*)*, описанной в листинге (*Листинг 82, часть 2*). Можно заметить, что параметрами этой функции выступает константная строка – имя файла и указатель на *TypeWeather*, смысл которого – передача адреса массива *ArrayWeather* в функцию «*по указателю*».

Листинг 82 (часть 2)

```
int Read(const char *file_name, TypeWeather *ArrWeather)
{       ifstream fin(file_name, ios_base::in);
        if (fin.is_open())
        {       int i=0;
                while (!fin.eof())
                {       ReadWeather(fin, &(ArrWeather[i]));
                        i++; }
                fin.close();
                return i;
        } return 0;
}
```

Для чтения из файла подпрограмма *Read( )* вызывает последовательно, пока не дойдет до конца файла, подфункцию *void ReadWeather(ifstream&, TypeWeather&)*, описанную в листинге (*Листинг 82, часть 3*), которая в качестве аргументов получает ссылку на объект *fin* класса потокового ввода *ifstream* и указатель на очередную структуру из массива *ArrayWeather*. При вызове функции *ReadWeather( )* приходится передавать в неё адрес очередного элемента массива – *&(ArrWeather[i])).*

Листинг 82 (часть 3)

```
void ReadWeather(ifstream &fin, TypeWeather *x)
{     fin >> x->Date.Day;
      fin >> x->Date.Month;
      fin >> x->Date.Year;
      fin >> x->Temperature;
      fin >> x->Moisture;
      int i; fin >> i; x->WindDirection= (TypeWD)i;
      fin >> x->WindPower;
}
// ---------------------------------------------------------------
string StringWeather(TypeWeather &x)
{ string st= "прогноз на " + to_string(_Longlong(x.Date.Day)) + "." +
to_string(_Longlong(x.Date.Month))+"."+to_string(_Longlong(x.Date.Year));
      st+= "\nтемпература " + to_string(x.Temperature) + " градусов, ";
```

```
      st+= "влажность " + to_string(x.Moisture) + " процентов,\n";
      switch(x.WindDirection)
      {     case N: st+= "ветер северный "; break;
            case NE: st+= "ветер северо-восточный "; break;
            case NW: st+= "ветер северо-западный "; break;
            case E: st+= "ветер восточный "; break;
            case W: st+= "ветер западный "; break;
            case SE: st+= "ветер юго-восточный "; break;
            case SW: st+= "ветер юго-западный "; break;
            case S: st+= "ветер южный "; break;
      };
      st+= to_string(x.WindPower) + " метров в секунду\n";
      return st;
}
```

Вывод на экран элементов массива *ArrayWeather* – структур типа *TypeWeather*, осуществляется при помощи функции *string StringWeather(TypeWeather&)*, возвращающей строку для вывода (*см. Листинг 82, часть 3*). В этой подпрограмме использован способ передачи объекта структурного типа в функцию *«по ссылке»*.

8.5. Структура, включающая в свой состав динамический массив

Поскольку структура может включать в себя указатели, значит, эти указатели могут хранить в себе адреса динамических объектов – переменных различного типа, массивов, строк, матриц. Рассмотрим, как это делается.

Создадим новый тип данных – структуру для работы со строками переменной длины. Для хранения динамической строки *d* используется указатель *char**, статическая строка *s* задается сразу в виде статического массива из 100 элементов типа *char*.

Листинг 83

```
struct STR112
{       int a;
        char s[100]; // статическая строка
        // заготовка для хранения динамической строки - указатель на char
        char* d;
        char c;
};
```

При описании конструируемого типа данных *struct STR112* память под переменные не выделяется и никакими значениями не заполняется: создается тип данных, но не переменная этого типа (*Листинг 83*).

Приведенный ниже листинг (*Листинг 84*) иллюстрирует создание статической переменной *x* типа *STR112*. Память под статическую строку *x.s* выделяется сразу при описании переменной. Память под динамическую строку еще не выделена. Указатель *x.d*, для определенности, обнуляется. Символ '\0', интерпретируемый компилятором как "конец строки", поставлен в первую позицию статической строки, таким образом, строка *x.s* пустая (хотя она по-прежнему занимает 100 байт).

Листинг 84

```
STR112 x; // задана статическая переменная типа STR12
x.a= 1234; // заполнение полей переменной x значениями
x.c= 'A';
x.d= NULL; // память под динамическую строку не выделена
x.s[0]= '\0'; // символ "конец строки" поставлен в начало строки
```

На иллюстрации (*Рис. 52*) схематически представлено размещение переменной *x* структурного типа в оперативной памяти. Важно понимать, что поля структуры размещаются в памяти последовательно, без пропусков. Можно видеть, что адрес переменной *x* совпадает с адресом первого поля структуры – с полем *x.a*. Второе поле

структуры смещено на 4 байта, так как *sizeof(int)=4*. Второе поле нашей структуры – это статическая строка *x.s*, она имеет 100 элементов и занимает *100×sizeof(char)=100* байт.

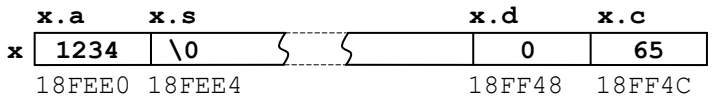

Рис. 52. Структура, содержащая в себе статическую строку

Третье поле – указатель *x.d*, который в дальнейшем будет указывать на динамическую строку, а пока он обнулен. Он расположен непосредственно после строки *x.s* и занимает 4 байта. Последнее поле *x.c* расположено сразу же после указателя *x.d*, занимает 1 байт и содержит код введенного символа. Таким образом, вся переменная *x* структурного типа должна бы занимать в памяти 109 байт, но идеология 64-разрядных данных требует, чтобы размер переменных выравнивался до числа, кратного 8 байтам. Так что переменная ***x*** будет занимать в памяти 112 байт.

Листинг ниже (*Листинг 85*) иллюстрирует заполнение статической строки *x.s* при помощи функции *strcpy()*, выделение памяти под безымянную динамическую строку, адрес которой будет храниться в указателе *x.d* и заполнение динамической строки *x.d* при помощи функции *strcpy()*.

Листинг 85

```
strcpy(x.s,"привет!"); // заполнение статической строки
// выделение памяти под динамическую строку в «куче»
x.d= (char*)malloc(7*sizeof(char));
strcpy(x.d,"привет!"); // заполнение динамической строки
```

Обращение к динамической строке, созданной при помощи функции *malloc()* возможно только по адресу, так как имени у нее нет. В дальнейшем мы будем говорить «динамическая строка *x.d*», но нужно четко понимать, что *x.d* – это статическая переменная-указатель, хранящая адрес динамически созданной строки. При выделении памяти под динамическую строку при помощи функции *malloc()* выполняется подряд три операции:

– вычисление количества байт для динамической строки *7*sizeof(char)* = 7 байт;

– выделение 7 байт под динамическую строку при помощи функции *malloc()* и возвращение нетипизированного адреса этой строки – *void**;

– принудительная типизация адреса из *void** в типизированный адрес – *char**.

Полученный адрес сохраняется в переменной-указателе *x.d*, как показано на *Рис. 53*.

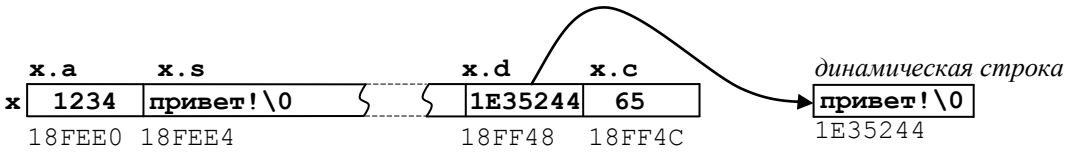

Рис. 53. Структура, содержащая в себе статическую и динамическую строки

Напишем подпрограмму для выделения памяти под динамическую строку структуры *STR112* и заполнения ее полей (*Листинг 86*). Поскольку в теле подпрограммы необходимо изменять внешнюю переменную и т.к. размер структуры значителен, формальные параметры в функцию лучше передавать *«по указателю»*. В остальном, всё аналогично примеру выше. Напомним, что обращение к полям указателя на структуру осуществляется оператором **->** «стрелочка».

Листинг 86

```
// подпрограмма для заполнения структуры STR112
void set_STR112 (STR112 *y) // передача параметров "по указателю"
{       y->a= 0;
        y->c='A';
        strcpy(y->s,"static string s"); // копирование данных в строку
        // выделение памяти под динамическую строку
        y->d= (char*)malloc(31*sizeof(char));
        strcpy(y->d,"dynamic string d"); // копирование данных в строку
}
```

Для правильного понимания того, как размещены в памяти компьютера поля структуры *STR112* (*см. Рис. 53*) и какие значения они содержат, предлагаю рассмотреть подпрограмму *void print_STR112(STR112*)*, отображающую на экране поля исследуемой структуры, адреса всех ее полей и длины статической и динамической строк (*Листинг 87*):

Листинг 87

```
// подпрограмма вывода на экран структуры STR112
void print_STR112 (STR112 *z) // передача параметров "по указателю"
{       printf("\n"); // вывод на экран полей структуры
        printf("a= %d\n",z->a);
        printf("s= %s\n",z->s);
        printf("d= %s\n",z->d);
        printf("c= %c\n",z->c);
        // вывод адресов всех полей структуры
        printf("addresses\n");
        printf("addr_STR112= %p\n",z);
        printf("addr_a= %p\n", &(z->a));
        printf("addr_static_string_s= %p\n",&(z->s));
        printf("z->s= %p\n",z->s);
        printf("addr_dynamic_string_d= %p\n",&(z->d));
        printf("z->d= %p\n",z->d);
        printf("addr_c= %p\n", &(z->c));
        // размеры и длины строк
        printf("sizeof(s)= %d\n",sizeof(z->s));
        printf("sizeof(d)= %d\n",sizeof(z->d));
        printf("strlen(s)= %d\n",strlen(z->s));
        printf("strlen(d)= %d\n",strlen(z->d));
        printf("sizeof(STR112)= %d\n",sizeof(STR112));
}
```

Обратите внимание на разницу между показаниями подпрограмм *sizeof( )* и *strlen( )*. Пример ниже (*Листинг 88, часть 1*) иллюстрирует создание *статической* переменной *x* структурного типа *STR112*. Результаты выполнения этой программы приведены на *Рис. 54*.

Листинг 88 (часть 1)

```
int _tmain(int argc, char* argv[])
{       setlocale(LC_ALL, "Russian"); // подключение русского шрифта

        STR112 x; // задана статическая переменная типа STR12
        set_STR112(&x); // заполнение статической переменной
        print_STR112(&x); // вывод на экран статической переменной

        getch();
        return 0;
}
```

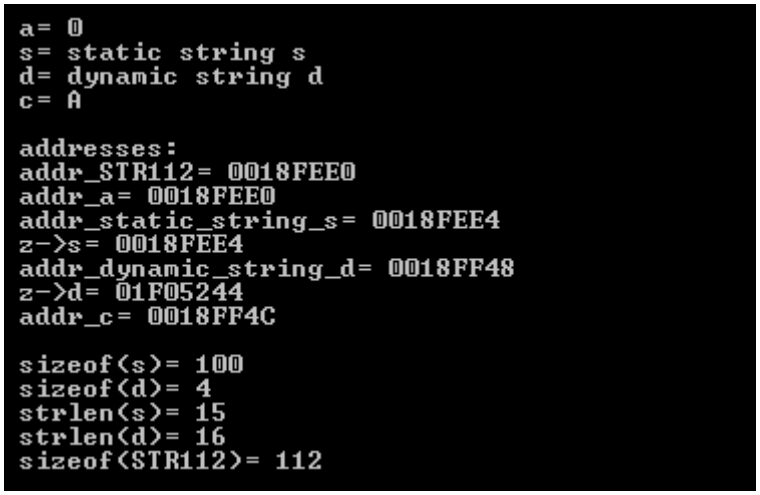

Рис. 54. Размещение в памяти статической переменной типа STR112

Дополнив эту программу строками из листинга ниже (*Листинг 88, часть 2*), проиллюстрируем работу с динамической переменной типа *STR112*. Видно, что указатель *y* удобно передавать в подпрограммы *print_STR112( )* и *set_STR112( )*, содержимое ячеек структуры и их размеры, естественно, не изменились (*см. Рис. 55*).

Листинг 88, часть 2

```
STR112 *y; // указатель на переменную типа STR112
// выделение памяти под динамическую структуру
y= (STR112*)malloc(sizeof(STR112));
set_STR112(y); // заполнение динамической переменной
print_STR112(y); // вывод на экран динамической переменной
```

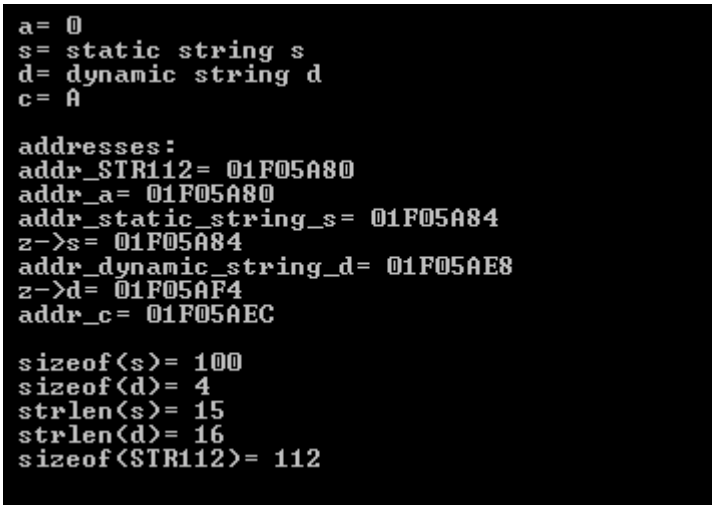

Рис. 55. Размещение в памяти динамической переменной типа STR112

По адресам, приведенным на *Рис. 55*, можно видеть, что в отличие от *Рис. 54*, динамическая переменная *y* размещается в динамической области памяти – *«куче»*. Однако принцип расположения полей не изменился, что естественно.

8.6. Лабораторная работа № 9. Структуры

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Освоить умение работать со структурами данных. Научиться создавать новый комбинированный тип данных – структуру. Научиться обращаться к статическим и

динамическим переменным этого типа и к массиву структур. Закрепить навыки работы с файлами и символьными строками.

Задание на лабораторную работу

1. Задать структуру для хранения данных в соответствии с индивидуальным заданием (*Таблица 18*, столбец «Данные»), и написать подпрограммы, выполняющую заданные действия над структурой данных (*Таблица 18*, столбец «Операции»).
2. Создать массив для хранения переменных (объектов) заданной структуры. Заполнить массив структур, прочитав данные из текстового файла (не менее 10 объектов).
3. Задать в программе динамическую переменную типа заданной структуры. Заполнить ее, вводя значения полей с клавиатуры, и поместить (вставить) в массив структур. При вводе осуществлять проверку правильности (разумности) вводимых значений.
4. Результаты работы программы записать в текстовый файл результатов.
5. Реализовать в программе простейшее интерактивное меню для организации последовательного выполнения задания и вывода результатов. В отчете привести блок-схему диалога программы с пользователем.
6. По выполненной лабораторной работе подготовить отчет с результатами работы программы. В отчет вставить листинг программного кода, файл данных и файл результатов. Отчет без комментариев не принимается.

Таблица 18 Варианты индивидуальных заданий

№	Данные	Операции
1.	Авиабилеты (фамилия, имя, отчество пассажира, авиакомпания, № рейса, аэропорт вылета, аэропорт назначения, дата, время).	Выбрать и вывести пассажиров указанного рейса. Выдать ближайший к заданному времени вылет. Выдать все рейсы с заданного аэропорта.
2.	Таксопарк: (фамилия, имя, отчество водителя, № машины, № заказа, время заказа, адрес, телефон, занятость).	Вывести список всех свободных машин. На заданное время вывести список всех занятых машин и время в пути. Определить, занят ли водитель.
3.	Магазин цветов: (название цветка, цвет, температура хранения, стоимость, количество в магазине).	Рассчитать стоимость покупки заданного количества выбранных цветов. Вывести список всех цветов с ценой и количеством. Определить требуемую температуру.
4.	ОК завода: (фамилия, имя, отчество сотрудника, дата рождения, № цеха, должность, оклад).	Вывести всех сотрудников заданного цеха. Рассчитать фонд оплаты труда всех сотрудников. Вывести всех работников пенсионного возраста.
5.	Библиотека: (№ в каталоге, название, Автор, количество экз., дата выдачи, стоимость).	Вывести все книги выбранного автора. Вывести количество книг и общую стоимость всех книг. Вывести все книги находящиеся «на руках» больше месяца.
6.	Деканат: (фамилия, имя, отчество студента, № группы, курс, количество задолженностей).	Выдать список всех студентов заданной группы. Выдать список всех должников данного курса. Рассчитать общее количество задолженностей по группам.

№	Данные	Операции
7.	Расписание: (дата, время начала и конца, название предмета, вид занятия, № аудитории, фамилия, имя, отчество преподавателя).	Выдать список всех занятий на дату. Рассчитать объем часов выбранного преподавателя в неделю. Вычислить количество лекций.
8.	Страны (площадь территории, название материка, количество населения, столица, язык).	Выдать список всех стран с населением больше 100 млн., Посчитать суммарную площадь стран на заданном материке, вывести все столицы.
9.	Больница (фамилия, имя, отчество пациента, диагноз, фамилия врача, стоимость лечения, № палаты).	Вывести всех пациентов с заданным диагнозом, рассчитать доход от каждого врача, вывести всех больных с заданной болезнью, лежащих в определенной палате.
10.	Билеты в кино: (название, жанр, дата и время сеанса, стоимость, ряд, место).	Выбрать все сеансы, заданного жанра. Выдать количество купленных билетов на данный фильм. Заполненность кинозала на определенный сеанс.
11.	Горы: (название, местоположение, вершина, высота, сложность маршрута, длительность подъема, стоимость).	Расположить маршруты по сложности и стоимости. Вычислить суммарную длительность и стоимость всех походов. Выбрать самый сложный и дорогой.
12.	Прогноз погоды: (местность, погода, температура, дата, вероятность прогноза).	Выбрать все регионы с температурой выше заданной. Вывести прогноз на выбранную дату. Вычислить суммарную вероятность прогноза.
13.	Психбольница: (фамилия, имя, отчество пациента, мания, дата поступления, затраты на лечение).	Подсчитать количество Наполеонов. Вывести затраты на содержание психов с даты поступления по указанную дату. Определить самого дешевого психа.
14.	Рецепт: (ингредиент, требуемое количество, имеющееся количество, стоимость, время хранения, дата покупки).	Выдать количество непросроченных ингредиентов на требуемую дату. Вычислить стоимость всех просроченных продуктов. Определить, достаточно ли имеющихся.
15.	Салон красоты: (фамилия, имя, отчество мастера, вид услуги, дата, время заказа, стоимость).	Выдать список всех клиентов на заданную дату. Рассчитать прибыль салона от определенного мастера. Определить количество каждых процедур в день.
16.	Коллекция марок: (название, страна, цена, год, погашенность, количество).	Рассчитать стоимость коллекции. Выдать все непогашенные марки выбранной страны. Выдать список всех марок заданной страны старше заданного года.
17.	Цирк: (название номера, фамилия, имя, отчество актера, длительность номера, жанр, рейтинг, требуемый реквизит).	Составить 3часовую программу из номеров наивысшего рейтинга, включить все жанры. Составить список реквизита к программе и вывести список незадействованных артистов.

№	Данные	Операции
18.	Зоомагазин: (название животного, количество, класс, вес, рост, стоимость, регион обитания, затраты на содержание).	Вычислить суммарную длину пресмыкающихся и массу млекопитающих. Вывести зверей по регионам обитания. Подсчитать относительный доход с учетом затрат.
19.	Созвездия: (название, количество звезд, координаты на небе, в каких полушариях).	Вывести все созвездия южного полушария. Определить относительное расстояние между наибольшим и наименьшим созвездиями. Сосчитать все звезды в небе.
20.	Фотосалон: (вид фото, тип бумаги, размер, дата заказа, срочность, стоимость).	Вычислить доход на указанную дату. Вывести список срочных заказов. Расставить все заказы по типу и размерам бумаги.
21.	Рыбацкая шхуна: (вид рыбы, масса и дата вылова, стоимость, время хранения.)	Рассчитать, к какой дате необходимо вернуться в порт. Отсортировать рыбу по стоимости. Подсчитать доход.
22.	Типы данных языка С: (название типа, размер в памяти, список операций, максимальное и минимальное значение, знаковый или нет)	Выписать типы данных по размеру ячеек памяти. Просчитать для каждого типа диапазон размеров. Выбрать среди беззнаковых наибольшее число.
23.	Стройка: (материал, стоимость, дата поставки, количество на складе, скорость расходования).	Выяснить, какой материал раньше кончится. Подсчитать общую стоимость материалов. Определить, какие товары кончатся к дате X.
24.	Туризм: (название курорта, средняя стоимость пребывания, заполненность, вместимость, уровень сервиса).	Выписать курорты по комфортности. Выбрать самый дорогой среди малозаполненных. Подсчитать относительный доход с каждого курорта.

Контрольные вопросы:

1. Возможно ли в полях структуры размещать данные различных типов?
2. Как выделяется память под динамическую переменную структурного типа, как освобождается?
3. Как размещаются в переменной структурного типа ячейки полей структуры?
4. Как осуществляется обращение к полям статической переменной структурного типа?
5. Что такое объединение (*union*), как размещаются поля в переменных такого типа данных?
6. Что такое указатель на структуру, как с ним работать, как обращаться к полям структурной переменной «по адресу?»
7. Можно ли указывать, сколько бит выделять под каждое поле структуры?
8. Как осуществляется обращение к полям динамической переменной структурного типа?
9. Как передается в подпрограмму статическая переменная структурного типа?

9. Операции с разрядами

```
Программистов на похоронах выносят
младшим байтом вперёд
```

В организации оперативной памяти всех вычислительных машин – начиная от IBM-совместимых и до современных вычислительных комплексов распределенных вычислений – заложен *принцип побайтной адресации*. Иначе говоря, обратиться по адресу можно лишь к байту, но не к каждому отдельно взятому биту. Этот принцип характерен для оперативной и дисковой памяти, для КЭШ всех уровней, BIOS, регистров процессора и т.д.

Но для некоторых программ необходима или, по крайней мере, полезна, возможность манипулировать отдельными *разрядами* (*битами*) в байте (*unsigned char*) или двухбайтном слове (*short int*). Например, часто состояние режимов устройств ввода-вывода устанавливаются байтом, в котором каждый разряд действует как признак «включено-выключено» или «вход-выход».

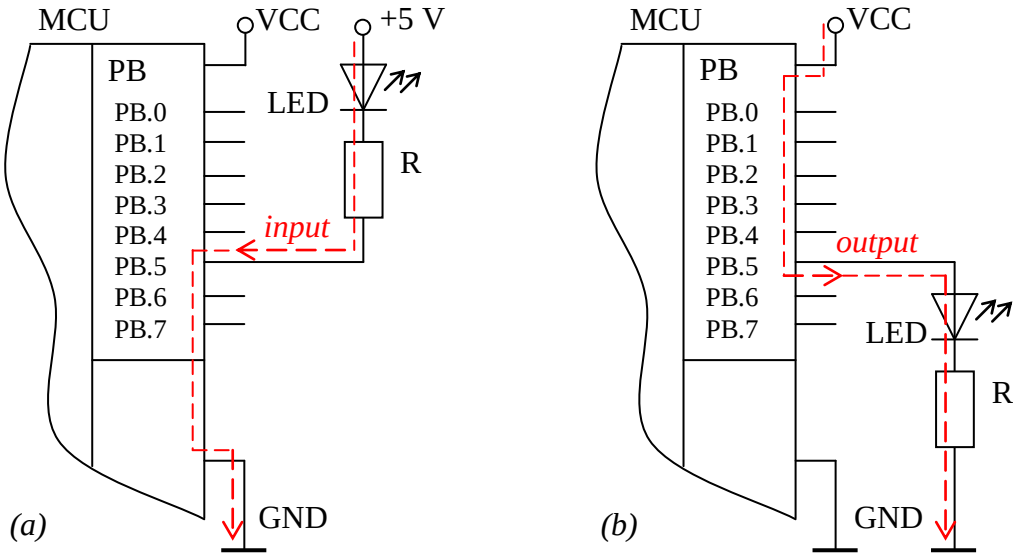

Рис. 56. Варианты схем принципиальных подключения светодиодного индикатора к микроконтроллеру

На *Рис. 56* приводятся варианты электрических принципиальных схем подключения светодиодного индикатора (*LED*) к пятому выходу порта *PB* некоторого микроконтроллера (*MCU*).

Для протекания тока так, как показано красной пунктирной стрелочкой, необходимо настроить пятый бит регистра *DDRB* на вход (*input*) или на выход (*output*) соответственно. И открыть его (включить) – изменить состояние пятого бита регистра *PORTB* с нолика («закрыто») на единичку («открыто»).

Листинг 89

```
void main()
{       DDRB = 0b11111111; //задаём все разряды порта B на вход
        PORTB = 0b11111111; //по умолчанию всё выключено
        while(1) // бесконечный цикл
        {       PORTB = ~PORTB; //переключаем состояние светодиода на
обратное
                delay_ms(100); //делаем задержку на 100 миллисекунд
        }
}
```

Листинг выше (*Листинг 89*) отражает программный код, задающий бесконечный цикл мигания светодиода. Для управления разрядами порта *PB* микроконтроллера используются восьмибитовые регистры *DDRB* и *PORTB*. Регистр *DDRB* определяет, на вход (*input*) или на выход (*output*) настроен порт, в листинге настройка порта сделана для схемы, приведенной на *Рис. 56 (а)* – «на вход». Резистор *R* выполняет функции токоограничения в соответствии с законом Ома.

Проанализировав приведенный выше листинг, можно видеть, что обращение к разрядам порта *PB* также подчинено *принципу побайтной адресации* – на каждой итерации цикла изменяются *все восемь* битов байта *PORTB*. Однако, для случая, когда необходимо изменить, например, *только пятый разряд* регистров *DDRB* и *PORTB*, не изменяя остальные разряды, не существует способа непосредственного обращения к нужному биту.

В языке *C++* есть несколько путей, позволяющих манипулировать разрядами. Рассмотрим два наиболее часто применяемых способа. Во-первых, набор из шести *поразрядных логических операций* (*см. п. 9.1, 9.2*), выполняющихся над битами байта. Во-вторых, конструируемая форма данных, объединяющая *битовое поле* (*field*) и *объединение* (*union*), дающая доступ к разрядам переменной (*см. п. 8.2, 8.3*).

В языке *C++* предусматриваются поразрядные логические операции и операции сдвига. В реальных программах используются целые переменные или константы, записанные в обычной форме. Например, вместо 00011001_2 можно использовать 25_{10} или 031_8, либо даже $0x19_{16}$. Ниже в примерах будут применяться 8-разрядные числа, в которых разряды пронумерованы справа налево (или слева направо от 7 до 0).

9.1. Поразрядные логические операции

Четыре операции производят действия над данными, относящимися к классу целых, включая *unsigned char*. Они называются "поразрядными", потому что выполняются отдельно над каждым разрядом *независимо* от разряда, находящегося слева или справа.

~ : Дополнение до единицы, или поразрядное отрицание

Эта унарная операция изменяет побитно каждую 1 на 0, а 0 на 1. Поэтому

$$\sim(10011010)_2 == (01100101)_2$$

&: Поразрядное И

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен 1, если только оба соответствующих разряда операндов равны 1. В терминах "истинно-ложно" результат получается истинным, если только каждый из двух одноразрядных операндов является истинным. Так,

$$(10010011)_2 \;\&\; (00000100)_2 == (00000000)_2$$
$$(10010011)_2 \;\&\; (00000001)_2 == (00000001)_2$$
$$(10010011)_2 \;\&\; (00111101)_2 == (00010001)_2,$$

потому что только четвертый и первый разряды обоих операндов содержат 1.

| : Поразрядное ИЛИ

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен 1, если любой из соответствующих разрядов операндов равен 1. В терминах "истинно-ложно" результат получается истинным, если один из двух (или оба) одноразрядных операндов является истинным. Так,

```
(10010011)2 | (11111111)2 == (11111111)2
(10010011)2 | (00000000)2 == (10010011)2
(10010011)2 | (00111101)2 == (10111111)2,
```

потому что все разряды, кроме шестого, в одном из двух операндов имеют значение 1.

^: Поразрядное исключающее ИЛИ

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен 1, если соответствующие разряды операндов не совпали и 0 – если совпали. В терминах "истинно-ложно" результат получается истинным, если один из двух (но не оба) одноразрядных операндов является истинным. Поэтому

```
(10010011)2 ^ (00000000)2 == (10010011)2
(10010011)2 ^ (11111111)2 == (01101100)2
(10010011)2 ^ (00111101)2 == (10101110)2,
```

заметим, что, когда нулевой разряд в обоих операндах имеет значение 1, нулевой разряд результата имеет значение 0.

Использование разрядной маски (MASK)

Описанные выше операции часто используются для работы с выбранными разрядами, причем другие разряды остаются неизменными. Для этого используется такое понятие как *маска* (*MASK*), представляющая собой байт, задающий те разряды, которые должны быть изменены и те – которые изменению не подлежат.

Например, предположим, что мы определили *MASK* в директиве *#define MASK* равным 2, т. е. двоичному значению 00000010, имеющему ненулевое значение только в первом разряде. Тогда оператор

Листинг 90

```
#define MASK 2 // 00000010
flags = flags & MASK;
```

установит все разряды байта *flags* (кроме первого) в 0, потому что любое значение разряда при выполнении операции & дает 0, если разряд второго операнда равен 0. Однако первый разряд останется неизменным.

Аналогично оператор

```
flags = flags | MASK;
```

установит первый разряд в 1 и оставит все остальные разряды неизменными.

Оператор «*исключающее или*» в комбинации с маской действует следующим образом:

```
flags = flags ^ MASK;
```

он инвертирует все разряды байта *flag*, заданные маской, и оставит все остальные биты неизменными.

Комбинируя описанные выше битовые (поразрядные) операции и маски различного вида можно работать непосредственно с теми разрядами, которые необходимы, не изменяя остальные биты в байте.

Например, в листинге ниже (*Листинг 91*) выполняется следующее действие: обнулить два бита – первый и второй, оставив без изменения остальные разряды в числе. Для этого задается маска, равная шести, позволяющая выделить два бита – первый и второй.

Листинг 91

```
#define MASK 6 // 00000110
x= 119; show_byte("\n                  x= ", x);
show_byte("(x | MASK) ^ MASK= ", (x | MASK) ^ MASK);
x= 204; show_byte("\n                  x= ", x);
show_byte("(x | MASK) ^ MASK= ", (x | MASK) ^ MASK);
x= 224; show_byte("\n                  x= ", x);
show_byte("(x | MASK) ^ MASK= ", (x | MASK) ^ MASK);
```

```
    MASK=  6   00000110

                  x=  119        01110111
(x | MASK) ^ MASK=  113        01110001

                  x=  204        11001100
(x | MASK) ^ MASK=  200        11001000

                  x=  224        11100000
(x | MASK) ^ MASK=  224        11100000
```

Рис. 57. Результат действия комбинированного оператора и маски на байт

9.2. Поразрядные операции сдвига

Эти операции сдвигают разряды влево или вправо. Мы снова запишем двоичные числа в явной форме, чтобы подробно показать механизм сдвига.

<< : Сдвиг влево

Эта операция сдвигает разряды левого операнда влево на число позиций, указанное правым операндом. Освобождающиеся позиции заполняются нулями, а разряды, сдвигаемые за левый предел левого операнда, теряются. Поэтому

```
10001010 << 2 == 00101000
```

где каждый разряд сдвинулся на две позиции влево, а справа добавилось два нуля.

>> : Сдвиг вправо

Эта операция сдвигает разряды левого операнда вправо на число позиций, указанное правым операндом. Разряды, сдвигаемые за правый предел левого операнда, теряются. Для чисел типа *unsigned* позиции, освобождающиеся слева, заполняются нулями. Для чисел со знаком в зависимости от ОС, освобождающиеся позиции могут заполняться нулями или значением знакового разряда (самого левого). Для значений без знака:

```
10001010 >> 2 == 00100010
```

где каждый разряд переместился на две позиции вправо.

Операции выполняют сдвиг влево или вправо, что также является эффективным способом умножения и деления на степени 2:

```
number << n; // умножает number на 2 в n-й степени
number >> n; // делит number на 2 в n-й степени, если число неотрицательное
```

что аналогично соответствующему алгоритму для десятичной системы счисления, обеспечивающему сдвиг десятичной точки при умножении или делении на 10.

В листинге, приведенном ниже (*Листинг 92*), показано, как при помощи операции сдвига организовать вывод *разрядов* – байтов, содержащих единицу только в одном соответствующем разряде (в черном прямоугольнике справа отражен результат вычислений).

Листинг 92

```
for( int i = 7; i >= 0; i--)
cout << (1 << i) << " ";
cout << endl;
```

```
128 64 32 16 8 4 2 1
```

Сравнивая побитно (*используя поразрядное И*) единицу соответствующего разряда и бит этого же разряда переменной *b* (на листинге ниже) можно вывести на экран значение переменной в двоичном формате (*Листинг 93*):

Листинг 93

```
void show_byte(const char* s, unsigned char b)
{
        printf("\n%s=   %d\t\t", s, b);
        for(int i= 7; i >= 0; i--)
                if( b&(1<<i) ) cout << "1";
                else cout << "0";
}
```

В этом листинге (*Листинг 93*) каждый разряд в числе *b* сравнивается побитно с изменяющейся в цикле маской (*1 << i*), представляющей собой единичку в соответствующем ***i**-том* разряде. Если сравнение истинно – в ***i**-том* разряде единица, то на экран выводится 1, в противном случае – выводится 0. Для двухбайтного числа этот же алгоритм приведен ниже (*Листинг 94*):

Листинг 94

```
short int b= 253; // 2 байта
for( int i = 15; i >= 0; i--)
        if( b & (1 << i) ) cout << "1";
        else cout << "0";
```

```
0000000011111101
```

9.3. Обращение к разрядам при помощи битовых полей

Второй способ манипуляции разрядами заключается в использовании *битовых полей* (*п. 8.2*). Следующее описание устанавливает новый тип данных – структуру *BitMap*, имеющую восемь 1-разрядных полей (битов). Задается однобайтная переменная *prnt*:

```
struct BitMap
{       bool b0:1; // обращение к соответствующему биту однобайтной
переменной
        bool b1:1;
        bool b2:1;
        bool b3:1;
        bool b4:1;
        bool b5:1;
        bool b6:1;
        bool b7:1; };

BitMap prnt;
```

Обращение к полям структуры *BitMap* можно использовать для присвоения значений отдельным битовым полям:

```
cout << "size= " << sizeof(prnt) << "byte" << endl;
```

```
size= 1byte
```

```
prnt.b0 = 0;
prnt.b3 = 1;
prnt.b4 = false;
prnt.b5 = true;

cout << prnt.b5 << endl;
```

```
1
```

Поскольку каждое поле состоит только из одного разряда, мы можем использовать для присваивания лишь значение 0 (*false*) или 1 (*true*). Переменная *prnt* запоминается в ячейке памяти, имеющей размер 1 байт (данные в памяти адресуются только побайтно).

Теперь для интерпретации данных зададим как в *п. 8.2 объединение* (*union*) *uBitMap* с двумя полями: поле *ch* типа *unsigned char* – для хранения, присваивания и вывода переменной целым байтом, и второе поле *bm* типа *BitMap* для обращения к каждому биту в отдельности.

```
union uBitMap
{       unsigned char ch;
        BitMap bm;
};
```

Напомним, что в объединении данные хранятся в одной и той же ячейке памяти, в данном примере – однобайтовой. Данный байт здесь может вызываться и интерпретироваться как в формате *unsigned char*, так и в формате *BitMap*.

```
uBitMap X;
cout << " sizeof(X)= " << sizeof(X) << "byte" << endl; // размер
X.ch= 128;

printf(" X= %c\n", X.ch); // вывод значения в формате char
printf(" X= %X\n", X.ch); // в шестнадцатеричном формате
printf(" X= %d\n", X.ch); // в десятичном формате
printf(" X= %o\n", X.ch); // в восьмеричном формате
// вывод в двоичном формате по одному биту:
cout << " X= " << X.bm.b0 << X.bm.b1 << X.bm.b2 << X.bm.b3
               << X.bm.b4 << X.bm.b5 << X.bm.b6 << X.bm.b7 << endl;
```

9.4. Практическая работа № 10. Поразрядные операции

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Освоить умение работать с разрядами в байте. Научиться изменять произвольный бит в байте, не меняя остальных. Выводить на экран двоичный код переменной произвольного типа. Повысить понимание преобразований двоичного формата.

Задание на практическую работу

1. Создать *динамическую* переменную и написать программу для работы с разрядами этой переменной в соответствии со своим вариантом индивидуального задания (*Таблица 19*) *двумя способами*:
   - Используя операции поразрядного сдвига (*п. 9.2*) и поразрядные логические операции (*п. 9.1*).
   - Используя структуру объединение (*union*) с использованием *битовых полей* (*п. 9.3*).
2. При программировании обоими способами выводить данные на экран в десятичном и двоичном форматах. Выводить на экран размер задаваемой переменной/структуры.
3. Корректно создать и уничтожить динамическую переменную.
4. По выполненной лабораторной работе подготовить отчет с результатами работы программы. В отчет вставить листинг программного кода, и результаты выполнения программы. Отчет без комментариев не принимается.

Таблица 19 Варианты индивидуальных заданий

1.	Запрограммировать изменение двухбайтного числа так, чтобы единица перемещалась от разряда к разряду слева направо.
2.	Запрограммировать изменение четырехбайтного числа так, чтобы байт последовательно заполнялся сначала единицами, а затем – нулями слева направо.
3.	Запрограммировать изменение четырехбайтного числа так, чтобы ноль перемещался от разряда к разряду слева направо.
4.	Запрограммировать изменение однобайтного числа так, чтобы четные разряды менялись местами с нечетными разрядами.
5.	Запрограммировать изменение четырехбайтного числа так, чтобы байт последовательно заполнялся сначала единицами, а затем – нулями справа налево.
6.	Запрограммировать изменение однобайтного числа так, чтобы ноль перемещался от разряда к разряду слева направо.
7.	Запрограммировать изменение четырехбайтного числа так, чтобы четные разряды менялись местами с нечетными разрядами.
8.	Запрограммировать изменение двухбайтного числа так, чтобы ноль перемещался от разряда к разряду слева направо.
9.	Запрограммировать изменение однобайтного числа так, чтобы единица перемещалась от разряда к разряду слева направо.
10.	Запрограммировать изменение двухбайтного числа так, чтобы ноль перемещался от разряда к разряду справа налево.
11.	Запрограммировать изменение двухбайтного числа так, чтобы четные разряды менялись местами с нечетными разрядами.
12.	Запрограммировать изменение четырехбайтного числа так, чтобы единица перемещалась от разряда к разряду справа налево.
13.	Запрограммировать изменение двухбайтного числа так, чтобы байт последовательно заполнялся сначала нулями, а затем – единицами справа налево.
14.	Запрограммировать изменение однобайтного числа так, чтобы ноль перемещался от разряда к разряду справа налево.
15.	Запрограммировать изменение четырехбайтного числа так, чтобы байт последовательно заполнялся сначала нулями, а затем – единицами слева направо.
16.	Запрограммировать изменение четырехбайтного числа так, чтобы единица перемещалась от разряда к разряду слева направо.
17.	Запрограммировать изменение двухбайтного числа так, чтобы байт последовательно заполнялся сначала единицами, а затем – нулями справа налево.
18.	Запрограммировать изменение однобайтного числа так, чтобы байт последовательно заполнялся сначала нулями, а затем – единицами справа налево.
19.	Запрограммировать изменение однобайтного числа так, чтобы единица перемещалась от разряда к разряду справа налево.
20.	Запрограммировать изменение двухбайтного числа так, чтобы байт последовательно заполнялся сначала единицами, а затем – нулями слева направо.
21.	Запрограммировать изменение двухбайтного числа так, чтобы единица перемещалась от разряда к разряду справа налево.

22.	Запрограммировать изменение четырехбайтного числа так, чтобы байт последовательно заполнялся сначала нулями, а затем – единицами справа налево.
23.	Запрограммировать изменение четырехбайтного числа так, чтобы ноль перемещался от разряда к разряду справа налево.
24.	Запрограммировать изменение двухбайтного числа так, чтобы байт последовательно заполнялся сначала нулями, а затем – единицами слева направо.

Контрольные вопросы:

1. Как запрограммировать побитное обращение к переменным целых типов.

2. Как работает двоичная побитная операция & «*и*», что получится в результате вычисления 23&126?

3. Как размещаются в памяти поля переменной типа *union*?

4. Как работает двоичная побитная операция | «*или*», что получится в результате вычисления 23|112?

5. Как работает двоичная побитная операция ^ «*исключающее или*», что получится в результате 103^112?

6. Объединение содержит три поля *unsigned short X*, *double Z* и *char Y*, какой размер будет занимать переменная этого типа?

7. Как работает побитная операция «*сдвиг*», что получится в результате 23>>2?

8. Что такое битовые поля и как с ними работать?

10. Классы

Ложась спать, программист ставит рядом на столик 2 стакана: один с водой – если захочет пить, второй пустой – если не захочет.

10.1. Введение в понятие класс

Классы и объекты в *C++* являются основными концепциями *объектно-ориентированного программирования – ООП.* [1, 2]

Классы в C++ это тип данных, описывающий *методы* (встроенные подпрограммы) и *свойства* (поля, тип ячеек памяти) объектов.

Объекты – переменные данного типа, *экземпляры* этого класса. В ООП существует три основных принципа построения классов:

1. *Инкапсуляция* – это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.
2. *Наследование* – это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.
3. *Полиморфизм* – свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Методика простейшего объявления классов приведена ниже:

```
class /*имя_класса*/
{
        private:
        /* список свойств и методов для использования внутри класса */
        public:
        /* список свойств и методов, доступных другим функциям программы */
        protected:
        /*список свойств и методов, доступных при наследовании*/
};
```

Объявление класса начинается с зарезервированного ключевого слова ***class***, после которого пишется *имя_класса*. В фигурных скобочках, объявляется тело класса. В теле класса объявляются три метки *спецификации доступа:* ***private***, ***public*** и ***protected***.

• Все методы и свойства класса, объявленные после спецификатора доступа *private*, будут доступны только внутри класса.

• Все методы и свойства класса, объявленные после спецификатора доступа *public*, будут доступны другим функциям и объектам в программе.

• Методы и свойства класса, объявленные после спецификатора доступа *protected*, будут доступны только внутри текущего класса и классов от него наследуемых. Спецификатор доступа, используемый при наследовании, будет рассмотрен в *главе 11*, где подробно будет рассмотрен механизм наследования.

Спецификаторы доступа и порядок их использования определяются программистом по необходимости. Рассмотрим простейший пример использования класса (*Листинг 95*). Похоже на структуру, не правда ли?

Листинг 95

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

class date // имя класса
{       public: // спецификатор доступа
```

```
        void message() // функция (метод) выводящая сообщение на экран
        { cout << "тестовое сообщение объекта класса date\n"; }
}; // конец объявления класса date

int main(int argc, char* argv[])
{
        date X; // объявление объекта
        X.message(); // вызов функции message() принадлежащей классу date
        system("pause");
        return 0;
}
```

В данном примере определен класс с именем *date*. , не содержащий никаких полей, но включающий в себя встроенную подпрограмму (метод) *void message( ).*

Объявлена переменная *X* типа *date* (объявлен объект *X* класса *date*). После того как объект класса объявлен, можно воспользоваться его методами. Обращение к свойствам и методам объектов класса организовано так же, как и для структур: через точку "." для статических переменных (объектов, экземпляров) и через стрелочку "->" для динамических.

Сама по себе подпрограмма *void message( )* не существует: к ней можно обратиться только из какой-нибудь переменной типа *date*.

В теле класса объявлен спецификатор доступа *public*, поэтому из главной функции *main( )* удалось получить доступ к методу *X.message( ).*

Set и Get методы классов

При создании *объектов класса* выделяется память для хранения *полей* (в разной литературе называемых так же *атрибутами* или *свойствами*). Если поля объекта имеют спецификатор доступа *private*, то доступ к ним могут получить только методы класса – внешний доступ к элементам данных запрещён.

Но работать-то с ними как-то необходимо. Поэтому принято объявлять в классах специальные методы – так называемые *set* и *get* функции, с помощью которых можно манипулировать элементами данных. *set-методы* служат для инициализации полей объекта, а *get-методы* позволяют выдать значения полей объекта.

Доработаем класс *date* так, чтобы в нём можно было хранить дату в формате *дд.мм.гг*, для изменения и просмотра даты реализуем соответственно *set* и *get* функции.

Листинг 96 (часть 1)

```
class date
{ private: // доступ ко всем последующим полям ограничен
     int day; // день
     int month; // месяц
     int year; // год
  public: // доступ ко всем последующим полям и методам разрешен
     void message() // метод, выводящий сообщение на экран
     { cout << "тестовое сообщение объекта класса date\n"; }

     void set(int n_day, int n_month, int n_year) // установка даты
     {    day = n_day; // инициализация поля day
          month = n_month; // инициализация поля month
          year = n_year; // инициализация поля year
     }

     void get() // отобразить текущую дату
     { cout << "date: " << day << "." << month << "." << year << endl; }

     int get_year() // возвращает значение свойства "год"
     { return year;     }
}; // конец описания класса
```

В этом примере (*Листинг 96, часть 1*) из-за спецификатора доступа *private* к переменным *day, month и year*, получают доступ только методы класса. Функции, не принадлежащие классу, не могут обращаться к этим переменным.

Листинг 96 (часть 2)

```
void _tmain(int argc, char* argv[])
{    setlocale(LC_ALL, "rus");

     int day, month, year; // вспомогательные переменные
     cout << "Введите текущий день месяц и год!\n";
     cout << "день: ";     cin >> day;
     cout << "месяц: ";    cin >> month;
     cout << "год: ";      cin >> year;

     date X; // объявление статического объекта класса date
     X.message(); // вызов метода message()
     X.set(day, month, year); // инициализация даты методом set()
     X.get(); // отобразить дату методом get()
// cout << "год= " << X.year; << endl; // ошибка доступа к полю year
     cout<< "год= " << X.get_year(); <<endl; // доступ по методу get_year()

     system("pause");
}
```

Элементы данных или методы класса, объявленные после спецификатора доступа *private*, до начала следующего спецификатора доступа называются *закрытыми полями класса* и *закрытыми методами класса*. Следуя принципу наименьших привилегий и принципу хорошего программирования, целесообразно объявлять элементы данных после спецификатора доступа *private*, методы класса делать открытыми, а для манипулирования элементами данных, объявлять функции *get и set*.

10.2. Конструктор и деструктор

При создании объектов (переменных) класса, можно сразу же проинициализировать их поля, и выполняет эту функцию конструктор.

Конструктор – специальная функция, которая выполняет начальную инициализацию элементов данных, причём имя конструктора обязательно должно совпадать с именем класса. Важным отличием конструктора от остальных функций является то, что он не возвращает значений вообще никаких, в том числе и *void*. В любом классе должен быть конструктор, даже если явным образом конструктор не объявлен (как в предыдущем классе), то компилятор предоставляет конструктор по умолчанию, без параметров.

Деструктор класса – это метод, выполняющий противоположную функцию – разрушения объекта класса, в нем обычно заложены алгоритмы освобождающие память, занятую под динамические поля класса. Имя деструктора должно совпадать с именем класса, но с добавлением символа тильда "~".

Поскольку структура создаваемого пользователем класса заранее не известна, автоматически генерировать конструкторы и деструкторы невозможно. Следовательно, задача написания и корректной работы конструктора и деструктора – это ответственность программиста. Доработаем класс *date* (*Листинг 96, часть 3*), добавив к нему конструктор *date( )* и деструктор *~date( )*.

Листинг 96 (часть 3)

```
class date
{ private: // закрытые поля класса
     int day, month, year;

  public: // открытые поля и методы
     date(int n_day, int n_month, int n_year ) // конструктор класса
     {    set(n_day, n_month, n_year); } // вызывает set-метод
```

```
    ~date() { }// деструктор класса - пустое тело деструктора
    // * * * // остальные методы класса date
}; // конец описания класса

//-------------------------------------------------------------
void _tmain(int argc, char* argv[])
{   date X(11,2,2013); // объявление статического объекта
                       // и инициализация элементов данных
    X.message(); // вызов функции message()
    X.get(); // отобразить дату

    date *Y; // указатель на объект класса date
    Y= new date(12,11,2013); // вызов конструктора для создания
                             // динамического объекта класса date
    int j= Y->get_year(); // вызов метода get_year()
    delete Y;     // вызов деструктора для удаления динамического объекта
    system("pause");
}
```

Конструктор здесь (*Листинг 96, часть 3*) имеет три параметра, через которые он получает информацию о дате. В теле конструктора вызывается *set-функция* для установки даты. При объявлении (и создании) объекта класса вызывается конструктор, которому в списке фактических параметров передаются три аргумента. Затем, в конструкторе выполняется начальная инициализация *закрытых полей класса*.

Вызов конструктора и деструктора для статических и динамических переменных

Конструктор класса вызывается в момент создания объекта (переменной) этого класса, а деструктор – в момент уничтожения объекта.

Таким образом, конструктор статического объекта вызывается *автоматически* в момент его описания (поскольку здесь же происходит и его создание), а деструктор – в конце области существования переменной – статического объекта класса (там, где автоматически происходит уничтожение статической переменной).

Создание и уничтожение динамического объекта явно задается и реализуется программистом – оператор *new* (*calloc, malloc*) создает объект, а оператор *delete* (*free*) его уничтожает. Поэтому в идеологии языка C++ заложен автоматический вызов конструктора в момент запуска оператора *new* (или функций *calloc, malloc*), а деструктор метода вызывается в момент начала работы оператора *delete* (или функции *free*).

В примере выше (*Листинг 96, часть 3*) задается динамический объект *Y* класса *date*. Для его создания используется оператор *new*, вызывающий конструктор *date(11, 2, 2013)* для инициализации полей динамического объекта *Y*. Соответственно, оператор *delete* вызывает деструктор *~date( )*, предназначенный для освобождения динамических полей объекта класса:

```
void _tmain(int argc, char* argv[])
{   date X(11,2,2013); // здесь вызывается конструктор для объекта X
    X.message();
    X.get();
    date *Y;
    Y= new date(12,11,2013); // здесь вызывается конструктор для объекта Y
    int j= Y->get_year();
    delete Y; // здесь вызывается деструктор для объекта Y
    system("pause");
// здесь вызывается деструктор для объекта X
}
```

Приведенный выше пример не отражает главного – для чего необходимы конструкторы и деструкторы? Ведь можно было задавать значения полей через метод *set*, а для чего нужен деструктор вообще не ясно – в данном примере тело его пусто.

Ответом на этот вопрос может стать пример со структурой *struct STR112*, содержащей динамическую строку (*Листинг 83* – *Листинг 88*), приведенный в *п. 8.5*. Можно вспомнить, что в том примере нам пришлось вручную создавать динамический массив, а его адрес хранить в одном из полей структуры *STR112*. Встает вопрос о том, корректно ли освобождается память из-под динамического массива после того, как переменная уничтожается.

Можно также вспомнить задачи корректного выделения памяти под динамические массивы и освобождения памяти, рассматриваемые в *п. 5.2*, *5.6*. Теперь, мы имеем замечательный инструмент – встроенные в тип данных подпрограммы, которые будут гарантировано автоматически вызваны при создании и при удалении объекта такого типа – это *конструкторы* и *деструкторы*.

Разберем пример (*Листинг 97*), в котором класс *VectorType* содержит два динамически создаваемых массива – одномерный динамический массив чисел, чей адрес хранится в поле *vect*, и динамическую строку, чей адрес хранится в поле *name*. В этом примере обратим внимание на корректное выделение памяти под динамические объекты в конструкторе и удаление оной в деструкторе.

Листинг 97 (часть 1)

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

class VectorType
{    private: // доступ ко всем последующим полям ограничен
          int count;          // размер массива
          char *name;         // имя массива
          long double *vect;  // указатель на массив переменной длины
     public: // доступ ко всем последующим полям и методам разрешен
          VectorType(); // конструктор по умолчанию
          VectorType(const char *n, int c); // конструктор с параметрами
          VectorType(const char *NameFile); // конструктор чтения
          ~VectorType(); // деструктор
          void Show(const char *annot); // метод визуализации
}; // конец описания класса

// ------------------------------------------------------------------
VectorType::VectorType() // конструктор по умолчанию
{    count= 5 + rand()%5; // случайный размер массива от 5 до 10
     name= new char[2];
     name[0]= 65 + rand()%22; // случайная заглавная буква
     name[1]='\0'; // конец строки
     vect= (long double*)calloc(count,sizeof(long double));
     for(int i=0; i<count; i++)
          vect[i]= rand()/1000.0; // случайные числа
     cout << "конструктор объекта "<< name << " отработал" << endl;
}

// ------------------------------------------------------------------
VectorType::~VectorType()// деструктор
{    cout << "деструктор объекта "<< name << " отработал" << endl;
     count= 0;
     free(vect); // освобождение памяти динамического массива чисел
     delete[]name; // освобождение памяти динамической строки
}
```

Заметьте, что в данном примере (*Листинг 97*) в описании класса *VectorType* присутствуют только прототипы методов этого класса, а реализация этих методов дается ниже. Для того чтобы сообщить компилятору о том, что методы *VectorType( )*, *~VectorType( )* и *Show( )* – это не просто подпрограммы, а методы определенного

класса, используется *оператор принадлежности* – :: «*двойное двоеточие*», обозначающий тот факт, что эти функции являются методами класса. Собственно, правильно писать названия этих методов полностью с указанием принадлежности к имени класса: *VectorType::VectorType( ), VectorType::~VectorType( )* и *void VectorType::Show( ).*

Обратите внимание, что поскольку память под динамические массивы *vect* и *name* выделяется в конструкторе по-разному (это сделано только ради иллюстрации), то и освобождается она в деструкторе соответствующим образом.

В конструктор и деструктор класса добавлены строки следующего вида: *cout << "конструктор отработал" << endl*. Строки не нужны, они служат только для того, чтобы проиллюстрировать тот момент выполнения программы, в который будут работать конструктор и деструктор для динамической и статической переменных.

Листинг 97 (часть 2)

```
void VectorType::Show(const char *annot)
{     cout << annot << " " << name << " = { ";
      for(int i=0; i<count; i++)
            cout << vect[i] << " ";
      cout << "}" << endl;
}
// ------------------------------------------------------------------
void _tmain(int argc, _TCHAR* argv[])
{     setlocale (LC_ALL, "Russian");
      srand(time(NULL));

      VectorType D; // здесь вызывается конструктор для объекта D
      D.Show("");

      VectorType *E;
      E= new VectorType; // здесь вызывается конструктор для объекта E
      E->Show("");
      delete E; // здесь вызывается деструктор для объекта E

      cin.get();
      // здесь вызывается деструктор для объекта D
}
```

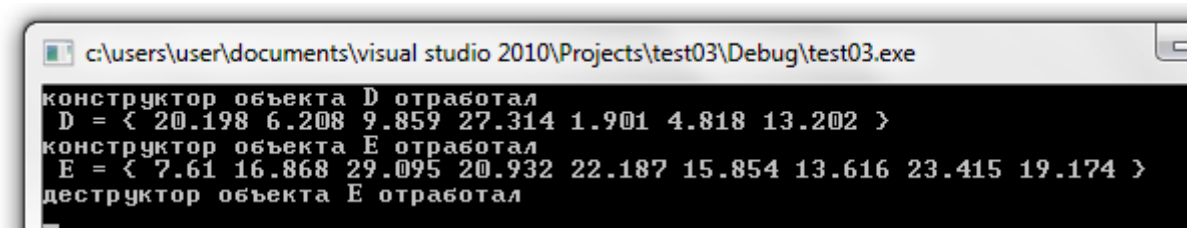

Рис. 58. Результаты работы программы

Из рисунка (Рис. 58) видно, что конструкторы и деструкторы объектов класса *VectorType* корректно запускаются, выделяют и освобождают память при работе с динамическими полями класса.

В описании класса *VectorType* присутствуют еще два прототипа конструкторов: *VectorType::VectorType(const char *n, int c);* – это *конструктор с параметрами*, позволяющий передать конструктору класса параметры, которые можно присвоить закрытым полям класса. А также *VectorType::VectorType(const char *NameFile);* – это *конструктор чтения*, в котором объект конструируется и заполняется читаемыми из файла данными. Рассмотрим их работу (*Листинг 97*).

Листинг 97 (часть 3)

```
VectorType::VectorType(const char *n, int c) // конструктор с параметрами
{     count= c; // копирование параметра c
      name= new char[strlen(n)+1];
```

```
      strcpy(name, n); // копирование параметра n
      vect= (long double*)calloc(count,sizeof(long double));
      for(int i=0; i<count; i++)
            vect[i]= rand()/1000.0;
}

// ---------------------------------------------------------------------
VectorType::VectorType(const char *NameFile) // конструктор чтения
{     char *s= new char[255]; // вспомогательная строка
      ifstream fin(NameFile, ios_base::in); // открыли файл для чтения
      if (fin.is_open()) // если файл успешно открылся
      {     fin >> s; // считать первое слово из файла - это имя массива
            name= new char[strlen(s)+1];
            strcpy(name, s);
            fin >> count; // считать второе слово из файла - это размер
            vect= (long double*)calloc(count,sizeof(long double));
            for(int i=0; i<count; i++)
            {     fin >> s; // чтение и преобразование типа элементов
                  vect[i]= (long double)atof(s); }
            fin.close(); } // закрыть файл
      else // если файл открыть не удалось
      {     count= 0; // создать пустые массивы
            name= new char[1]; name[0]='\0';
            vect= (long double*)calloc(1,sizeof(long double));
            vect[0]= 0.0;
            cout << "файл " << NameFile << " открыть не удалось\n"; }
      delete[]s; // вспомогательная строка
}

// ---------------------------------------------------------------------
void _tmain(int argc, _TCHAR* argv[])
{     setlocale (LC_ALL, "Russian");
      srand(time(NULL));

      VectorType X;
      X.Show("создан конструктором по умолчанию");

      VectorType *Y;
      Y= new VectorType;
      Y->Show("создан конструктором по умолчанию");
      delete Y;

      VectorType *Z= new VectorType("массив Z",22);
      Z->Show("создан конструктором с параметрами");
      delete Z;

      VectorType A("vector.txt");
      A.Show("считан из файла");
      cin.get();
}
```

```
создан конструктором по умолчанию L = { 6.904 28.469 9.54 17.456 22.482 }
создан конструктором по умолчанию A = { 9.406 9.492 25.696 5.57 25.403 }
создан конструктором с параметрами массив Z = { 1.951 7.928 2.7 13.671 9.41 7.38
7 24.643 22.626 6.317 20.165 13.507 25.199 4.393 6.575 20.012 25.079 0.134 3.575
 1.404 28.113 31.956 6.587 }
считан из файла массив_A = { 1.11 2.22 3.33 4.44 5.55 6.66 7.77 8.88 9.99 }
```

```
файл vector2.txt открыть не удалось
считан из файла  = { }
```

Рис. 59. Результаты работы программы

Результаты работы программы, приведенные на Рис. 59,*а* иллюстрируют работу конструктора с параметрами и конструктора чтения для статических и динамических объектов класса. Кроме того, можно видеть на Рис. 59,*б* как отработает конструктор чтения в случае, если файл открыть не удастся.

10.3. Перегрузка операторов

В языке C++ значительное число операторов (полный список содержит *Таблица 2*), но что это такое – операторы? Пришло время разобраться, как они устроены.

Оператор – это специального рода функция, которая описывает *унарную или бинарную* операцию над переменными определенных типов. Оператор отличается от любой другой функции только принципом вызова и служебным словом *operator*. *Унарный* оператор имеет один аргумент (например *i++*, **ptr*, *!flag* и др.), а *бинарный* оператор имеет два аргумента (*a+b*, *x==y*, *c[i]* и др.).

Мы научились проектировать новые типы данных – классы, логично иметь возможность создавать для своих классов необходимые операции. В примере ниже (*Листинг 98, часть 1*) к нашему классу *VectorType* добавляются три функции – методы класса, выполненные в виде операторов. Это оператор индексации, называемый также «квадратные скобки» – *operator [ ]*, оператор присваивания *operator =* и оператор сравнения *operator ==*.

Листинг 98 (часть 1)

```
class VectorType
{     private:
            int count;
            char *name;
            long double *vect;
      public:
            VectorType();
            VectorType(const char *n, int c);
            VectorType(const char *NameFile);
            ~VectorType();
            void Show(const char *annot);
            long double& operator [] (int indx);
            VectorType& operator = (const VectorType&);
      friend bool operator == (const VectorType&, const VectorType&);
}; // конец описания класса

// ------------------------------------------------------------------
long double LD_MIN= -1.7976931348623158e+308; // самое маленькое число
// ------------------------------------------------------------------
long double& VectorType::operator [] (int indx)
{     if ((indx < 0) || (indx > count-1)) // если ошибка диапазона
            return LD_MIN;
      return vect[indx];
}

// ------------------------------------------------------------------
VectorType& VectorType::operator = (const VectorType& V2)
{     if (this==&V2) // если присваивание самому себе
            return *this;
      free(vect); // очистить память старого массива
      count= V2.count; // новый размер
      vect= (long double*)calloc(count,sizeof(long double));
      for(int i=0; i<count; i++)
            vect[i]= V2.vect[i]; // поэлементное заполнение массива
      return *this;
}
```

Оператор индексации *long double& operator [] (int indx)* в качестве аргумента получает *int indx* – целое число, определяющее *индекс* (номер) элемента в массиве, а возвращает ссылку на искомый элемент массива (*long double&*). Если индекс лежит в диапазоне [*0, count-1*], то возвращается элемент массива с номером *indx*, а вот если индекс выходит за границы массива, что делать? Есть несколько решений – можно

прервать работу программы, выдать сообщение об ошибке и пр. – решение за программистом. В нашем случае принято решение возвращать всегда самое маленькое из возможных чисел типа *long double*.

Рассмотрим, как вызываются созданные операторы в основной программе:

Листинг 98 (часть 2)

```
void _tmain(int argc, _TCHAR* argv[])
{     setlocale (LC_ALL, "Russian");
      srand(time(NULL));

      VectorType X("X",8);
      X.Show("");

      cout << "X[0]= " << X[0] << endl; // оператор индексирования
      cout << "X[3]= " << X[3] << endl;
      cout << "X[7]= " << X[7] << endl;
      cout << "X[8]= " << X[8] << endl << endl;

      VectorType Y("Y",4);
      Y.Show("");
      Y=X; // оператор присваивания объекта класса
      Y.Show("Y=X\n");

      cin.get();
}
```

```
 X = { 23.81 16.443 19.938 21.482 30.216 28.544 11.854 20.528 }
X[0]= 23.81
X[3]= 21.482
X[7]= 20.528
X[8]= -1.79769e-308

 Y = { 28.636 24.659 16.936 24.044 }
Y=X
 Y = { 23.81 16.443 19.938 21.482 30.216 28.544 11.854 20.528 }
```

Рис. 60. Результаты работы программы

Результаты работы программы (*Рис. 60*) иллюстрируют использование операторов индексации и присваивания. Можно видеть, как отрабатывает оператор индексации выход индекса за границы диапазона.

Оператор присваивания *VectorType& operator = (const VectorType&)*, получает в качестве аргументов ссылку на объект *VectorType&*, производит поэлементное присваивание содержимого одного вектора другому, возвращает ссылку на исправленный объект *VectorType&*. При реализации оператора (*Листинг 98, часть 1*), вначале производится проверка на самокопирование, кроме того, копирование реализовано так, что вектор данных *vect* и размер вектора *count* заменяются новыми, а название массива *name* остается прежним.

Если рассмотреть внимательно реализацию оператора присваивания, то может вызвать удивление, что функция-оператор *имеет только один параметр*, несмотря на то, что перегружается операция *бинарная*: одному объекту присваивается значение другого. Это связано с тем, что при реализации бинарного оператора с использованием функции-метода ей передается явным образом только второй (правый) аргумент. Первым аргументом служит указатель *this* на текущий (левый) объект, от имени которого и вызывается оператор =. Так сказать, *this* – это указатель на самого себя.

```
Y=X; // оператор присваивания одного объекта класса другому
Y.operator=(X); // другой способ вызова того же бинарного оператора =
```

В примере выше приведены два альтернативных способа вызова одного и того же бинарного оператора присваивания. Сразу становится ясно, что в данном случае

объект *X* – это правый аргумент оператора, а *Y* – это его левый аргумент. Указатель *this* содержит адрес именно его – вызывающего объекта *Y* класса *VectorType*.

Оператор сравнения рассмотрим в следующем параграфе.

10.4. Дружественные функции (friend)

Некоторые функции могут быть объявлены как дружественные (*friend*) к создаваемым классам, они получают доступ к полям и методам, объявленным не только как *public* и *protected*, но и как *private*. Для объявления дружественной функции используется ключевое слово *friend*. В нашем примере (*Листинг 98, часть 1*) механизм создания дружественной функции использован при написании *оператора сравнения*.

Обратите внимание: *bool operator == (const VectorType&, const VectorType&)* не является методом класса *VectorType*, не является его членом, а является лишь внешней функцией, получившей доступ к полям и методам класса посредством спецификатора *friend*. Поэтому *operator ==* получает в качестве аргументов две ссылки на переменные типа *VectorType*: не являясь членом класса, оператор сравнения не имеет указателя *this*.

Листинг 98 (часть 3)

```
// оператор сравнения ------------------------------------------------
bool operator == (const VectorType& V1, const VectorType& V2)
{     if (V1.count != V2.count) //сравниваем размеры массивов
      {     return false; }
      else //проверяем равны ли данные в ячейках массивов
      {     for (int i=0; i<V1.count; i++)
            {     if (V1.vect[i] != V2.vect[i])
                  {     return false; }  }  }
      return true;
}

// ------------------------------------------------------------------
void _tmain(int argc, _TCHAR* argv[])
{     VectorType *X= new VectorType("X",7);
      VectorType *Y= new VectorType("Y",7);
      X->Show("");
      Y->Show("");
      if (*X==*Y) cout << " векторы X и Y равны" << endl;
      else cout << " векторы X и Y неравны" << endl;

      *Y=*X; // оператор присваивания объектов класса
      X->Show("");
      Y->Show("");
      if (*X==*Y) cout << " векторы X и Y равны" << endl;
      else cout << " векторы X и Y не равны" << endl;
}
```

```
X = { 30.423 24.014 19.014 23.676 11.245 17.548 23.111 }
Y = { 23.804 8.478 26.662 12.338 14.935 2.047 25.07 }
векторы X и Y не равны
X = { 30.423 24.014 19.014 23.676 11.245 17.548 23.111 }
Y = { 30.423 24.014 19.014 23.676 11.245 17.548 23.111 }
векторы X и Y равны
```

Рис. 61. Результаты работы программы

Проиллюстрировать работу операторов сравнения и присваивания можно на примере динамических объектов *Y* и *X* класса *VectorType* (*Листинг 98, часть 3*). Обратите внимание на то, что данные в операторах сравнения и присваивания передаются «*по ссылке*». А в нашем случае *Y* и *X* – это указатели, хранящие адреса динамических ячеек типа *VectorType*, поэтому используется запись (**X==*Y*) и **Y=*X*, что значит «*содержимому указателя Y*» присвоить «*содержимое указателя X*», также и сравнение.

Заметьте, если бы мы написали (*X==Y*) и *Y=X*, то для сравнения и присваивания использовались бы не сконструированные нами операторы *operator == и operator =* класса *VectorType* (*Листинг 98, части 1, 3*), а типовые операторы *сравнения указателей* и *присваивания указателей*, реализованные в стандарте языка *C++*.

10.5. Отделение интерфейса от реализации

Для написания различных программ можно пользоваться разработанными ранее классами, для этого необходимо подключить (*#include*) заголовочный файл (**.h*) библиотеки, в котором они объявлены. Так мы пользуемся целым набором классов из библиотек <*iostream.h*>, <*fstream.h*>, <*time.h*> и др.

Разберемся, как реализуется подобная архитектура на примере класса *date*, рассмотренном нами ранее (*Листинг 96*). Эта технология называется «*отделение интерфейса класса от его реализации*» – т.е. описание класса – его интерфейс размещаются в заголовочном файле *date.h*, реализация методов этого класса – в файле реализации *date.cpp*, а создание объектов этого класса и вызов его методов – в программе *_tmain( )* в файле, например, *test04.cpp* (*см.* Рис. 62).

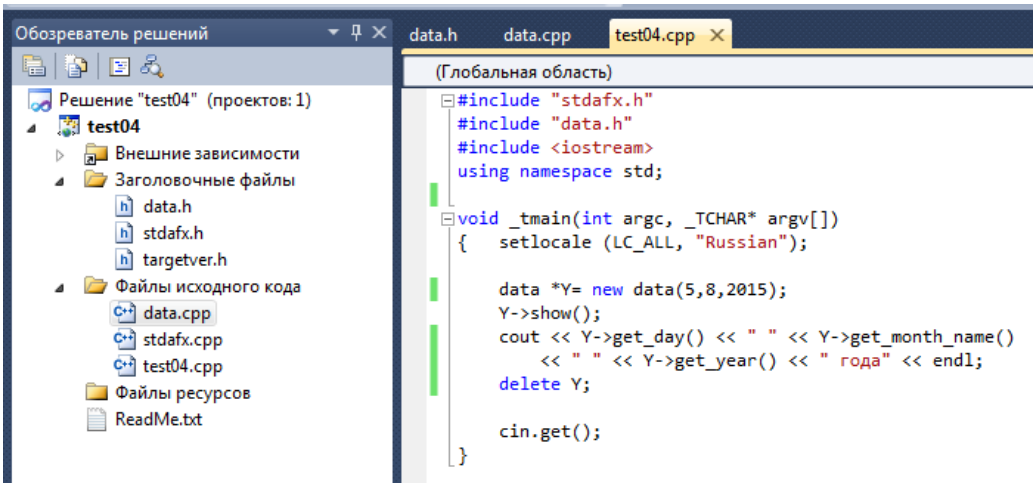

Рис. 62. Архитектура Visual C++ проекта test04

Для чего это делается? В коммерческих целях. Организованная таким образом архитектура доступа к классу позволяет предоставить пользователю только описание – интерфейс создаваемого класса *date* в файле *date.h*. После компиляции программного кода файл *date.cpp* может быть скрыт (вместо него останется на диске откомпилированный файл *date.obj* или специальным образом скомпанованные файлы библиотек *date.lib* или *date.dll* – *см. Рис. 63*).

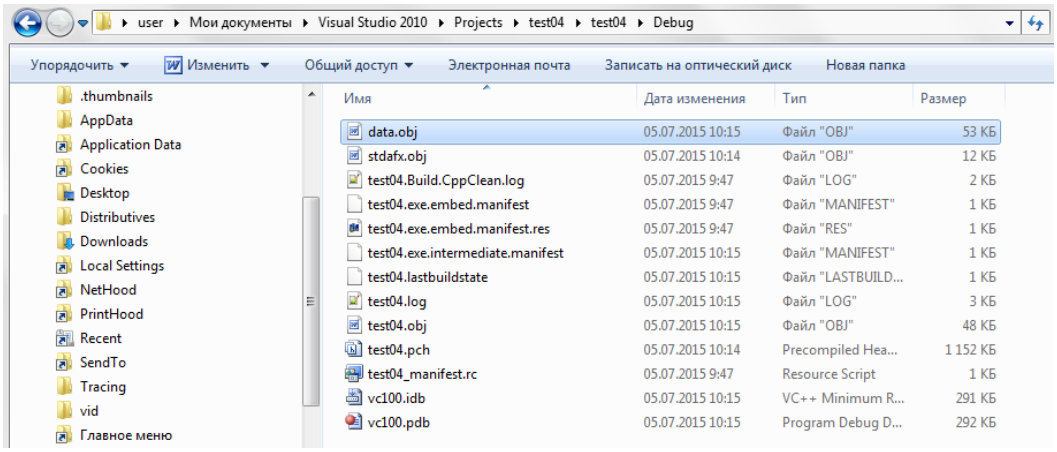

Рис. 63. Откомпилированный файл date.obj в папке проекта test04

Пользователю теперь передается заголовочный файл *date.h* и откомпилированный файл *date.obj* (*date.lib*, *date.dll* и т.п.), а исходные программные коды реализации методов класса *date* не показываются. Можно пользоваться нашим классом, но нельзя увидеть, как он работает и как сделан.
Рассмотрим отделение интерфейса от реализации класса *date* на примере (*Листинг 99*).

Листинг 99 (часть 1)

```
// date.h заголовочный файл
#ifndef DATE_H // если имя DATE_H ещё не определено
#define DATE_H // определить имя DATE_H
#pragma once

class date
{     private:
            int day, month, year;
      public:
            date(int, int, int);
            date();
            ~date();
            void set(int, int, int); // установка даты
            int get_year();
            int get_month();
            int get_day();
            void show(); // отобразить текущую дату
            char* get_month_name(); // строка - название месяца
};

#endif DATE_H // если имя DATE_H уже определено, повторно не определять
```

Заголовочный файл *date.h* предваряется и завершается директивами компилятору. Эти директивы называются «*препроцессорная обертка*», поговорим о ней позже.

Интерфейс класса

Интерфейс класса – это описание класса, определяющее его методы и свойства, (задание типов данных и прототипов методов), но не раскрывающее алгоритмов.

Реализация класса – это способ осуществления алгоритмов работы класса – определение/задание тел методов, ответ на вопрос «как работают те или иные методы?».

До этого мы не отделяли интерфейс класса от его реализации, то есть реализация методов осуществлялась внутри класса. Отделение интерфейса от реализации класса выполняется для того, чтобы скрыть способ осуществления работоспособности класса. Отделение интерфейса от реализации выполняется за шесть шагов:

1. добавить в проект заголовочный файл *.h (*Рис. 62*);
2. определить интерфейс класса в заголовочном файле (*Листинг 99, часть 1*);
3. добавить в проект исполняемый файл *.cpp (*Рис. 62*);
4. в исполняемом файле выполнить реализацию класса (*Листинг 99, часть 2*);
5. подключить заголовочный файл к программе (*Листинг 99, часть 1*);
6. убрать файл реализации и заменить его откомпилированным файлом (*Рис. 63*).

Имена заголовочному и исполняемому файлам даются, как правило, одинаковые.

Теперь рассмотрим содержимое файла реализации методов класса *date* (*Листинг 99, часть 2*). Нужно помнить – так как методы класса объявляются вне тела класса, то необходимо связать реализацию метода с классом. Необходимо явно указать, к какому классу относятся реализуемые методы, для этого перед именем метода необходимо написать имя класса и поставить *бинарный оператор принадлежности* "**::**", эта операция привязывает метод, объявленный извне, к классу.

Листинг 99 (часть 2)

```
// date.cpp файл реализации
#include "stdafx.h"
#include <iostream>
#include "date.h" // подключить интерфейс класса

date::date(void) { set(1, 1, 1900); }

date::~date(void) { }

date::date(int n_day, int n_month, int n_year )
{ set(n_day, n_month, n_year); }

void date::set(int n_day, int n_month, int n_year)
{ day = n_day; month = n_month; year = n_year; }

void date::show() // отобразить текущую дату
{ cout << "date: " << day << "." << month << "." << year << endl; }

int date::get_year() { return year; }

int date::get_month() { return month; }

int date::get_day() { return day; }

char* date::get_month_name() // строка - название месяца
{     if (month<1 || month>12) return ".0."; // неправильный месяц
      char* month_name[12] = {"января", "февраля", "марта", "апреля",
      "мая", "июня", "июля", "августа", "сентября", "октября", "ноября",
      "декабря" };
      return month_name[--month];
}
```

Итак, интерфейс класса определён, методы класса объявлены, осталось подключить заголовочный файл в исполняемом файле в программе *_tmain( )*. Чтобы главная функция увидела созданный нами класс и смогла его использовать, необходимо включить определение класса *#include "date.h"* в исполняемом файле. Кроме того, необходимо, чтобы файл с откомпилированными методами *date.obj* находился в папке проекта (*Рис. 63*).

Листинг 99 (часть 3)

```
// test04.cpp главный файл проекта
#include "stdafx.h"
#include "date.h"
#include <iostream>
using namespace std;

void _tmain(int argc, _TCHAR* argv[])
{     setlocale (LC_ALL, "Russian");

      date *Y= new date(5,8,2015);
      Y->show();
      cout << Y->get_day()  << " " << Y->get_month_name()
                            << " " << Y->get_year() << " года" << endl;
      delete Y;
      cin.get();
}
```

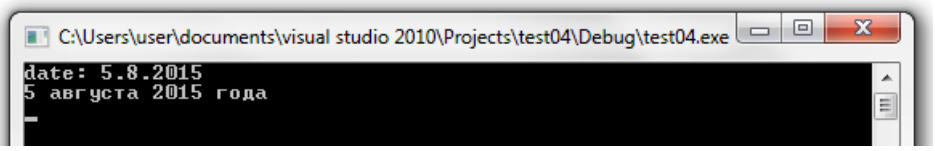

Рис. 64. Результаты работы программы

Препроцессорная обертка

Поскольку код расположен не в одном файле, а подключение заголовочных файлов необходимо не только в главном файле проекта, но и в других, то существует вероятность многократного включения в программу одного и того же заголовочного файла, что в свою очередь приводит к ошибке компиляции:

```
Ошибка 1 error C2011: CppStudio: переопределение типа "class date"
```

Чтобы не возникало такого рода ошибок, в *C++* существует специальная структура кода (директива компилятора), которую ещё называют *препроцессорная обёртка:*

```
// структура препроцессорной обёртки
#ifndef ИМЯ_ЗАГОЛОВОЧНОГО_ФАЙЛА_H  // если этот файл ещё не определен
#define ИМЯ_ЗАГОЛОВОЧНОГО_ФАЙЛА_H  // то определить этот файл

// здесь дается определение класса

#endif ИМЯ_ЗАГОЛОВОЧНОГО_ФАЙЛА_H  // иначе не включать этот файл
```

Данная директива предотвращает попытку многократного включения заголовочных файлов. Препроцессорные директивы обрабатываются до этапа компиляции, программой-препроцессором, который не допускает многократного определения одного и того же класса. Директива *#ifndef* проверяет, определено ли имя *DATE_H*, если нет, то управление передаётся директиве *#define* и определяется интерфейс класса. Если же имя *DATE_H* уже определено, управление передаётся директиве *#endif,* пропуская определение класса.

Обратите внимание на то, как написано имя класса, используемое в сочетании с директивами препроцессорной обёртки: *DATE_H*. В имени заголовочного файла, в котором объявлен класс, символы переведены в верхний регистр, а вместо точки - символ нижнего подчёркивания.

С использованием препроцессорной обёртки, попытки подключения одного и того же файла, ошибки переопределения не вызовут. Этот же приём применяется и для предотвращения многократного определения функций, если они вынесены в отдельный файл.

10.6. Лабораторная работа № 11. Классы

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Научиться создавать класс и описывать его свойства и методы. Изучить спецификаторы доступа *private* и *public*. Освоить применение конструктора, деструктора, *set-* и *get-* методов. Освоить работу с объектами класса: корректно создавать, инициализировать, получать значения свойств, визуализировать и удалять. Научиться выделять память под динамические объекты класса и освобождать ее. Освоить обращение к свойствам и методам динамических объектов. Научиться программировать *интерфейс класса* и *реализацию класса* в соответствующих **.h* и **.cpp* файлах, корректно подключать их к проекту, используя *препроцессорную обертку*.

Задание на лабораторную работу

1. Задать класс, реализующий операции с данными, в соответствии с индивидуальным заданием (*Таблица 20*).
2. Написать конструктор и деструктор класса.

3. Свойства объектов класса хранить со спецификатором *private*. Написать методы *set* и *get* обращения ко всем свойствами класса.
4. Написать метод визуализации (вывода на экран) свойств класса.
5. Интерфейс, реализацию и вызов объектов класса запрограммировать в трех различных файлах: *класс.h, класс.cpp* и *main.cpp* – соответственно.
6. В главной программе продемонстрировать работу со статическими и динамическими переменными – объектами класса: корректно создать, заполнить данными, визуализировать и удалить.
7. Продемонстрировать *перегрузку* хотя бы одного *оператора*.
8. Продемонстрировать использование дружественной (*friend*) функции.
9. В отчете привести листинг программы, скриншоты тестирования программы, рисунок структуры класса. Листинг программы без комментариев не принимается.

Таблица 20 Варианты индивидуальных заданий

№	Задание
1.	Класс, реализующий операции с комплексными числами (z=x+iy). Вычислять модуль комплексного числа и корректное сложение комплексных чисел.
2.	Класс, хранящий вершины треугольника (x1, y1; x2, y2 и x3, y3) и производящий вычисление длин его сторон, а также определяющий прямоугольный он или нет.
3.	Класс для рисования кругов (хранить радиус R и координаты x,y центра). Вычислять площадь круга и длину окружности.
4.	Класс для работы с секундомером. Хранить время и отсчитанное число секунд. Считать секунды до нажатия клавиши, результат переводить в минуты и часы.
5.	Класс, хранящий две строки символов s1 и s2. Создать метод, производящий посимвольное сравнение строк и возвращающий строку несовпадений символов.
6.	Класс, хранящий 8 бит информации (флагов) в виде битовых полей. Выполнять поразрядное отрицание "~" хранимого байта, а также операции поразрядное И "&", ИЛИ "|" и исключающее ИЛИ "^" с вводимым байтом.
7.	Класс, хранящий два целых аргумента (месяц m и день d) и вычисляющий количество дней, прошедших с начала года.
8.	Класс, хранящий три целых аргумента (часы h, минуты m и секунды s) и вычисляющий количество секунд, прошедших с начала дня.
9.	Класс, хранящий вектор в трехмерном пространстве (x, y, z) и вычисляющий модуль этого вектора.
10.	Класс, хранящий дату дня рождения (месяц m и день d) и вычисляющий количество дней, оставшихся до дня рождения от текущей даты (m1, d1)
11.	Класс, хранящий координаты точки (x,y) на плоскости. Написать метод, определяющий, в каком квадранте находится точка.
12.	Класс, хранящий строку s1 символов. Создать метод, производящий посимвольное сравнение s1 с вводимой строкой s2 и возвращающий строку совпадающих символов, если они есть.
13.	Класс, хранящий дату рождения человека. Написать метод, вычисляющий число прожитых дней.
14.	Класс, хранящий 8 бит информации – байт флагов b1. Создать методы, выполняющие поразрядный сдвиг влево "<<" и поразрядный сдвиг вправо ">>" на заданное число позиций N. Производить проверку условия N<8.
15.	Класс, хранящий (часы h, минуты m и секунды s) и вычисляющий количество секунд, оставшихся до конца суток.

№	Задание
16.	Класс, хранящий вектор - пару точек пространства (x1, y1, z1 и x2, y2, z2) и вычисляющий площадь треугольника с вершинами в этих точках и начале координат.
17.	Класс, для операций с точками в пространстве (хранить x, y, z - координаты). Вычислять расстояние до начала координат.
18.	Класс для хранения координат вершин вектора на плоскости (x1, y1 и x2, y2). Вычислить длину этого вектора и его координаты.
19.	Класс, хранящий фамилию, имя и отчество человека. Написать метод вывода на экран полной информации и сокращенной – в формате «Фамилия И. О.».
20.	Класс, хранящий две строки символов s1 и s2 разной длины. Создать метод, выводящий на экран наибольшую из них и вычисляющий количество символов в каждой.
21.	Класс, хранящий параметры сфер в пространстве (радиус R и координаты центра – x, y, z). Написать метод, определяющий пересекается ли заданная сфера с другой – координаты которой (R2, x2, y2, z2).
22.	Класс для хранения координат прямоугольных «окон» на экране (хранить координаты левого верхнего и правого нижней вершин x1, y1 и x2, y2). Вычислять площадь окна.

Контрольные вопросы:

1. Что такое класс? Для чего он используется, в чем его особенности, преимущества?
2. Спецификатор доступа *private* – для чего он применяется?
3. Что называется полями и методами класса? Что такое объект класса?
4. В чем смысл технологии отделения интерфейса от реализации методов класса?
5. В чем различие понятий класс и объект класса?
6. Может ли быть в классе несколько конструкторов, деструкторов, *set*- и *get*-методов?
7. Когда вызывается деструктор класса для статических и динамических объектов?
8. Как организовано в классах ограничение и разрешение доступа к полям и методам?
9. Конструктор и деструктор – для чего они нужны? Когда используются?
10. Спецификатор *public* – как он используется в конструировании класса?
11. Оператор принадлежности к классу («*двойное двоеточие*», «::») – для чего и как он применяется? Можно ли обойтись без него?
12. Что такое *set*- и *get*- методы? В каком случае они необходимы?
13. Как реализовано обращение к полям и методам статических и динамических объектов класса?
14. Что такое унарные и бинарные операторы? Что такое приоритет операций?
15. Какие спецификаторы доступа вы знаете? В чем их особенность?
16. Что такое заголовочный файл и чем он отличается от файла реализации методов? Как их использовать в проекте?
17. Можно прописать реализацию методов класса в самом теле класса, а можно – в другом месте и даже в другом файле. В чем различие?
18. Что такое спецификатор доступа *friend*? Как он используется?
19. Что представляют собой операторы? Можно ли создавать операторы для собственных классов?
20. Что такое «препроцессорная обертка»? Как это применяется?

11. Наследование

Наследование – это механизм создания нового класса на основе ранее созданного. Наследование имеет смысл, если множество разнородных объектов имеют общие характеристики или функции.

Рассмотрим пример классов для работы со студентами (*Листинг 100, часть 1*). Целесообразно выделить в отдельный (базовый) класс *THuman* информацию о фамилии, имени, дате рождения и пр., а на основе этого класса создавать новые (дочерние) классы для более детального описания студента.

Листинг 100 (часть 1)

```
#ifndef THUMAN_H
#define THUMAN_H

class THuman // базовый класс
{    private:
          int ID;
          char *first_name;
          char *second_name;
          int year_birth;
          bool gender;
     protected:
          int get_age();
     public:
          THuman();
          ~THuman();
          void show();
};
//---------------------------------------------------------------------------

class TStudent: public THuman // наследуемый класс от класса THuman
{    protected:
     private:
          char *university;
          char *faculty;
          char *speciality;
     public:
          TStudent();
          ~TStudent();
          void show();
};

#endif THUMAN_H
```

Используя механизм наследования на основе базового *THuman*, создадим дочерний класс с именем *TStudent* для хранения дополнительной информации – названия университета, факультета и специальности. Для этого после имени дочернего класса *TStudent* ставится символ «**:**», а затем пишется имя базового класса *public THuman* (с указанием уровня доступа).

Уровень наследования *public* определяет доступ класса *TStudent* ко всем свойствам (полям) и методам (функциям) класса *THuman*, которые не являются частными (private). Т.е. в нашем примере (*Листинг 100, часть 1*) доступ к полям *ID*, *first_name*, *second_name*, *year_birth* и *gender* для методов класса *TStudent* закрыт.

Доступ к методам класса *THuman*, у программ класса *TStudent* имеется, в виду того, что они имеют спецификатор доступа *public*, это иллюстрирует пример (*Листинг 100, часть 2*). Здесь представлены реализации некоторых методов рассматриваемых классов и показаны возможности применения наследуемых методов, в том числе, конструктора и деструктора.

Листинг 100 (часть 2)

```
void THuman::show()
{    cout << ID << ". " << first_name << " " << second_name << " ";
cout << year_birth << " года рождения\n"; }

int THuman::get_age()
{    return (2015-year_birth); }

//-------------------------------------------------------------------------
TStudent::TStudent(): THuman() // наследование конструктора
{ }

TStudent::~TStudent()
{    THuman::~THuman(); } // наследование деструктора

void TStudent::show()
{    THuman::show(); // наследование метода от базового класса
     cout << university << " " << faculty << " " << speciality << "\n";
}
```

Обратите внимание на уровень доступа *protected*, предоставленный классом *THuman* своему методу *get_age( )* (*Листинг 100, часть 1*).

Уровень доступа protected

Уровень доступа ***protected*** обеспечивает доступ к методам и свойствам класса всем классам-потомкам, т.е. наследующим от данного базового класса, но доступа к этим методам и свойствам для внешних подпрограмм не предоставляется.

Обобщим еще раз:

• *public* – доступ предоставляется для любых подпрограмм: для методов данного класса, для методов классов-наследников и для любых внешних функций;

• *private* – доступ предоставляется только методам данного класса;

• *protected* – доступ предоставляется для методов данного класса и для методов классов-наследников. Для внешних подпрограмм доступ закрыт.

Ниже приводится пример (*Листинг 100, часть 3*) вызова метода *get_age( )* базового класса *THuman* из метода *TStudent::show( )* наследуемого класса *TStudent.* Доступ разрешен, поскольку *protected.* Там же приводится пример неправильного обращения к закрытым (*private*) свойствам *ID* и *first_name* класса *THuman.*

Листинг 100 (часть 3)

```
void TStudent::show()
{    cout << ID << ". " << first_name; // доступ запрещен (private)
     THuman::show(); // доступ разрешен (public)
     cout << "прожил " <<get_age()<< " лет"; // доступ разрешен (protected)
     cout << university << " " << faculty << " " << speciality << "\n";
}
```

Рассмотрим пример (*Листинг 100, часть 4*) вызова метода *get_age( )* класса *THuman* из основной программы *main( ).* Видно, что этот вызов здесь уже является неправильным, поскольку главная программа закрыта для доступа спецификатором *protected.* В этом месте не имеет значения, элемент какого класса (*THuman* или *TStudent*) вызвал метод *get_age( )* – для всей программы *main( )* он недоступен.

Листинг 100 (часть 4)

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

int _tmain(int argc, _TCHAR* argv[])
{    setlocale (LC_ALL, "Russian");

     THuman X;
```

```
    X.show(); // вызывается метод THuman::show()
    cout << X.get_age(); // доступ запрещен (protected)

    TStudent S;
    S.show(); // вызывается метод TStudent::show()
    cout << S.get_age(); // доступ запрещен (protected)

    cin.get();
    return 0;
}
```

Стоит обратить внимание на вызов одноименных методов *THuman::show( )* и *TStudent::show( )* – в зависимости от того, объектом какого класса вызван этот метод, такой и используется. Точнее сказать так: метод *THuman::show( )* замаскирован новым одноименным методом *TStudent::show( )* в классе *TStudent.*

Абстрактные классы

Использовать только конструктор по умолчанию не всегда удобно. Если у класса имеется несколько конструкторов, то можно указать явно, какой конструктор базового класса вызывать из конструктора класса-потомка:

Листинг 101

```
class THuman // базовый класс – абстрактный класс
{   protected:
        THuman(); // конструктор по умолчанию
        THuman(int, const char*); // конструктор чтения из файла
        THuman(THuman&); // конструктор копирования
        ~THuman();
};
//---------------------------------------------------------------------------
class TStudent: public THuman // наследуемый класс от класса THuman
{   public:
        TStudent(): THuman() { }; // конструктор по умолчанию
        TStudent(int LN, const char* File): THuman(LN, File);
        TStudent(TStudent&); // конструктор копирования
        ~TStudent();
};
```

Теперь конструктор «по умолчанию» *TStudent()* будет вызывать конструктор *THuman()* базового класса, а конструктор с параметрами *TStudent(int, const char*)* будет вызывать конструктор *THuman(int, const char*).* При этом конструктор копирования *TStudent(TStudent&)* явно не вызывает соответствующий конструктор *THuman(THuman&)* – только, может быть, в теле подпрограммы.

В рассматриваемой задаче (*Листинг 101*), может оказаться так, что класс *THuman* является вспомогательным, и никогда не появится необходимость создавать его объекты (экземпляры) для непосредственной работы с ними. Поэтому целесообразно защитить его от возможности создания, поместив конструкторы данного класса в раздел *protected*.

Теперь мы не сможем в главной программе (и в любых других подпрограммах) создать объект данного класса, ведь доступ к его конструкторам закрыт. Такие классы называются *абстрактными классами*, т.е. они не могут существовать как самостоятельные объекты, а служат только для создания новых, дочерних классов.

При этом другие методы абстрактных классов, описанные как *public* можно вызывать через объекты классов-потомков.

Рассматриваемый нами пример – класс *TStudent* позволяет с помощью конструкторов класса *THuman* заполнять свойства, связанные с личными данными человека, но свойства *university, faculty* и *speciality* по-прежнему придется заполнять

вручную. Имеется ли возможность наследования одним потомком свойств и методов от нескольких базовых классов? Да, имеется.

11.1. Множественное наследование

Язык *C++* предоставляет возможность создавать дочерние классы на основе *нескольких* базовых, что приводит к концепции *множественного наследования.*

Реализуем концепцию множественного наследования, создав иерархию классов в соответствии с *Рис. 65*, добавив еще один базовый класс с именем *TJob*, который будет отвечать за свойства, соответствующие тем профессиональным функциям, которые выполняют студенты (*TStudent*), преподаватели (*TTeacher*) и декан (*TDean*).

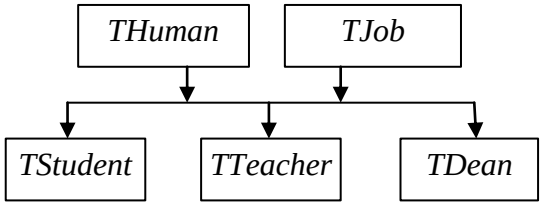

Рис. 65. Множественное наследование

Листинг 102 (часть 1)

```
class THuman // базовый класс
{    private:
          int ID;
          char *first_name;
          char *second_name;
          int year_birth;
          bool gender;
     protected:
          int get_age();
          THuman();
          ~THuman();
     public:
          void show();
};
//---------------------------------------------------------------------------
class TJob // базовый класс
{    private:
          char *university;
          char *faculty;
          int year_work;
          float salary;
     protected:
          TJob();
          ~TJob();
     public:
          void show();
};
```

В примере (*Листинг 102, часть 1*) свойства *university, faculty* выделены в отдельный базовый класс *TJob*, кроме того, в него добавлены поля *year_work* (стаж) и *salary* (зарплата). Проверьте себя, ответив на вопрос: являются ли эти базовые классы абстрактными? Почему?

Теперь обратим внимание на описание классов-наследников (*Листинг 102, часть 2*) *TStudent* и *TTeacher*, здесь демонстрируется наследование от двух базовых классов *THuman* и *TJob*. Естественно, все свойства обоих базовых классов передаются в классы-наследники со спецификатором *private* и поэтому закрыты для доступа, а методы базовых классов – доступны, поскольку они либо *protected*, либо *public*.

Классы *TStudent* и *TTeacher* различаются собственными, а не наследуемыми свойствами. Класс *TStudent* содержит поле *speciality* (специальность), реализованное в виде динамической строки, а класс *TTeacher* включает в себя поле *count* (количество предметов) и поле *subjects* (предметы), организованное как динамический массив строк.

Листинг 102 (часть 2)

```
class TStudent: public THuman, public Tjob
{    protected:
     private:
          char *speciality; // символьная строка
     public:
          TStudent();
          TStudent(int id, const char* fn, const char* sn, int yb, bool g,
                   const char* u, const char* f, int yw, float s,
                   const char * sp);
          ~TStudent();
          void show();
};
//---------------------------------------------------------------------------
class TTeacher: public THuman, public Tjob
{    protected:
     private:
          int count; // количество строк
          char **subjects; // динамический массив строк
     public:
          TTeacher();
          TTeacher(int id, const char* fn, const char* sn, int yb, bool g,
                   const char* u, const char* f, int yw, float s,
                   const char * sub);
          ~TTeacher();
          void show();
};
```

Можно видеть, что классы-наследники *TStudent* и *TTeacher*, содержат по два конструктора – «*по умолчанию*» и «*с параметрами*» (*Листинг 102, часть 2*), деструктор и метод *show( )*. Ясно, что в конструктор с параметрами требуется передавать начальные значения для всех полей – как собственных, так и унаследованных от классов *THuman* и *TJob*. Рассмотрим реализацию методов классов-наследников:

Листинг 102 (часть 3)

```
TStudent::TStudent(): THuman(), TJob()
{    speciality= new char[1]; speciality[0]='\0'; }

TStudent::TStudent(int id, const char* fn, const char* sn, int yb, bool g,
            const char* u, const char* f, int yw, float s,
            const char * sp): THuman(id, fn, sn, yb, g), TJob(u, f, yw, s)
{    speciality= new char[strlen(sp)+1]; strcpy(speciality, sp); }

TStudent::~TStudent()
{    THuman::~THuman();
     TJob::~TJob();
     delete[]speciality; speciality= NULL; }

void TStudent::show()
{    THuman::show();
     TJob::show();
     cout << "студент специальности: " << speciality << endl; }
```

Из примера (*Листинг 102, часть 3*) видно, как последовательно вызываются конструкторы, деструкторы и методы базовых классов из соответствующих методов класса *TStudent*, кроме того, можно видеть и работу с собственными свойствами *speciality* класса-наследника. Обращение к собственным свойствам базовых классов производится только через их методы, так как они *private*. Методы базовых классов доступны в классе наследников, так как они либо *protected*, либо *public*. Полезно вспомнить, как создаются, удаляются и копируются динамические строки (*см. гл. 6*).

Наследующий класс *TTeacher* реализован чуть-чуть иначе (*Листинг 102, часть 4*): в нем содержится динамический массив строк *subjects* и целочисленное поле *count*. Поскольку работа с динамическими массивами является пройденным материалом, читателю предлагается самостоятельно вспомнить, как организуется создание динамических массивов, их удаление и копирование (*п.п. 5.6, 6.2, 8.5*).

В данном примере конструктор с параметрами последним параметром получает строку *const char* sub*, содержащую несколько слов, разделяемых пробелами, запятыми или точками – это интерпретируется как список предметов. В конструкторе эта строка разбивается на слова, каждое из которых помещается в динамический массив *subjects*, а количество этих строк – в переменную *count*. Вспомнить, как разбить строку на подстроки вам поможет пример (*Листинг 66*).

Листинг 102 (часть 4)

```
TTeacher::TTeacher(): THuman(), TJob()
{    subjects= new char*[1]; // массив указателей на строки
     subjects[0]= new char[1]; // динамическая строка
     subjects[0][0]='\0';
     count= 0; }

TTeacher::TTeacher(int id, const char* fn, const char* sn, int yb, bool g,
            const char* u, const char* f, int yw, float s,
            const char * sub): THuman(id, fn, sn, yb, g), TJob(u, f, yw, s)
{    char *token, *separators = "\t,. ";
     char *buf= new char[strlen(sub)+1]; strcpy(buf, sub);
     int i = 0;
     token = strtok(buf, separators);
     while(token != NULL)
     {    token = strtok(NULL, separators);
          i++; }

     count= i; // количество подстрок в строке buf
     subjects= new char*[count]; // массив указателей на строки

     strcpy(buf, sub);
     i = 0;
     token = strtok(buf, separators); // выделение подстроки
     while(token != NULL)
     {    subjects[i]= new char[strlen(token)+1]; // копирование в массив
          strcpy(subjects[i], token);
          token = strtok(NULL, separators);
          i++;
     } // while
}

TTeacher::~TTeacher()
{    THuman::~THuman();
     TJob::~TJob();
     for (int i=0; i<count; i++) // удаление динамического массива строк
          delete[](subjects[i]);
     delete[]subjects; subjects= NULL; }

void TTeacher::show()
{    THuman::show();
     TJob::show();
     for (int i=0; i<count; i++)
          cout << "\n" << subjects[i]; cout << endl;}
```

На листинге ниже (*Листинг 102, часть 5*) приведена реализация основной программы *main( )*, демонстрирующей использование описанных классов для создания и использования статических и динамических экземпляров этих типов.

Заметьте, что экземпляры *X* и *J* базовых классов *THuman* и *TJob* создать не удается, так как эти классы абстрактные.

Листинг 102 (часть 5)

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

void _tmain(int argc, _TCHAR* argv[])
{    setlocale (LC_ALL, "Russian");

     // THuman X(1,"Иван", "Иванов", 1997, true); // protected - закрыто
     // X.show();

     // TJob J("ТУСУР", "ФЭТ", 2015, 1000.0); // protected - закрыто
     //J.show();

     TStudent S(1,"Петр", "Петров", 1997, true, "ТУСУР", "ФЭТ", 2015,
          0.0, "электроника и наноэлектроника");
     S.show();

     TTeacher *T= new TTeacher(2,"Анна", "Попова", 1991, false, "ТУСУР",
          "ФЭТ", 2014, 35000.0, "математика, физика, электроника");
     T->show();

     cin.get();
}
```

```
1. Петр Петров 1997 года рождения, мужчина. 18 лет.
ТУСУР // ФЭТ с 2015 года // 0
студент специальности: электроника и наноэлектроника
2. Анна Попова 1990 года рождения, женщина. 25 лет.
ТУСУР // ФЭТ с 2014 года // 35000
преподает предметы:
математика
физика
электроника
```

Рис. 66. Результаты работы программы

11.2. Дружественные классы

Классы, так же как и функции (*п. 10.4*), могут быть объявлены как *дружественные к определенным классам*. При этом они получают доступ к свойствам и методам, объявленным не только как *public* и *protected*, но и как *private*.

Для объявления дружественного класса используется ключевое слово *friend*, за которым следует имя класса. Следующий пример (*Листинг 103, часть 1*) демонстрирует, как объявление класса *TTeacher* дружественным (*friend*) базовому классу *TJob*, делает доступными свойства *university*, *faculty*, *year_work* и *salary* класса *TJob* методам класса-наследника *TTeacher*, не зависимо от спецификатора *private*.

Листинг 103 (часть 1)

```
class TJob
{       public:
                friend class TTeacher; // объявление дружественного класса
        private:
                char *university;
                char *faculty;
                int year_work;
                float salary;
        protected:
                TJob();
                ~TJob();
        public:
                void show();
};
```

В результате использования спецификатора *friend* частные свойства *university*, *faculty*, *year_work* и *salary* класса *TJob* оказываются доступными только одному

производному классу *TTeacher* и никакому другому, что обеспечивает их лучшую защиту по сравнению с уровнем доступа *protected*. Класс *TStudent* по-прежнему не имеет доступа к перечисленным частным свойствам класса *TJob*.

Изменим метод *void TTeacher::show()* как показано в примере (*Листинг 103, часть 2*), обратившись из этого метода к частным данным класса *TJob*. *Рис. 67* иллюстрирует тот факт, что доступ получен.

Листинг 103 (часть 2)

```
void TTeacher::show()
{  cout << endl; THuman::show();
   // TJob::show(); // реализуем непосредственный доступ к частным данным
   cout << "С "<< year_work << "года работает в" << university << endl;
   cout << "на факультете"<< faculty << ",получает" << salary  << "руб.\n";
   cout << "преподает следующие предметы:";
   for (int i=0; i<count; i++)
        cout << endl << subjects[i]; cout << endl;}
```

```
1. Петр Петров 1997 года рождения, мужчина. 18 лет.
ТУСУР // ФЭТ с 2015 года // 0
студент специальности: электроника и наноэлектроника

2. Анна Попова 1990 года рождения, женщина. 25 лет.
С 2014 года работает в ТУСУР
на факультете ФЭТ, получает 35000 руб.
преподает следующие предметы:
математика
физика
электроника
```

Рис. 67. Результаты работы программы main( )

Дружественными можно объявлять не только классы, но и отдельные функции классов (*см. п. 10.4*).

11.3. Виртуальные методы

Рассмотрим теперь целиком построенную нами иерархию классов (*Рис. 65*), реализованную в примерах (*Листинг 102, Листинг 103*). Логично иметь возможность работать с массивами или иными хранилищами объектов типа *THuman, TStudent* и *TTeacher*. Такая возможность есть. Никто не мешает создать массив, состоящий из указателей на объекты класса *THuman,* и присваивать этим указателям адреса динамически создаваемых экземпляров типа *THuman, TStudent* или *TTeacher*. Т.е. определяться с требуемым типом на этапе создания каждого конкретного динамического элемента массива:

Листинг 104

```
int cnt= 3;
THuman **people;
people = new THuman*[cnt]; // массив указателей для динамических объектов
people[0]= new THuman();       // создание динамических элементов массива
people[1]= new TStudent();     // различных классов
people[2]= new TTeacher();

for (int i=0; i<cnt; i++)      // удаление динамических элементов массива
       delete(people[i]);
delete[]people; // удаление массива указателей

for (int i=0; i<cnt; i++)
       people[i]->show();// вызов метода THuman::show() для эл-тов массива
```

С таким массивом удобно работать. Например, очистку такого массива просто производить в цикле (*Листинг 104*). Однако есть и недостатки такого подхода – все элементы массива описаны как тип *THuman**, поэтому при вызове методов для объектов этого типа *people[i]->show()*, вызывается один и тот же метод *show()* класса

*THuman**. А хотелось бы, чтобы вызывались методы нужного класса – *THuman, TStudent* или *Tteacher* соответственно, ведь в каждом из них заложен свой одноименный метод *show()* со специфической реализацией. Для решения таких задач используется идеология *виртуальных* методов.

Функция-член класса может содержать спецификатор *virtual*. Такой метод называется *виртуальным*.

Если некоторый класс содержит виртуальную функцию, а производный от него класс содержит функцию *с тем же именем и типами формальных параметров*, то обращение к этой функции для объекта производного класса вызывает функцию, определённую именно в производном классе.

Функция, определённая в производном классе, вызывается даже при доступе через указатель или ссылку на базовый класс. В таком случае говорят, что *функция производного класса подменяет функцию базового класса*. Если типы функций различны, то функции считаются разными, и механизм виртуальности не включается. Ошибкой является различие между функциями только в типе возвращаемого значения.

Листинг 105 (часть 1)

```
class THuman
{    private:
          int ID;
          char *first_name;
          char *second_name;
          int year_birth;
          bool gender;
     protected:
          int get_age();
     public:
          THuman();
          THuman(int id, const char* fn, const char* sn, int yb, bool g);
          ~THuman();
          virtual void show(); // виртуальный метод
};
//---------------------------------------------------------------------
class TGroup
{    private:
          int count; // размерность массива
          THuman **people; // указатель на динамический массив
     public:
          TGroup();
          TGroup(const char* file); // конструктор чтения из файла
          ~TGroup();
          void show_people();
};
```

Последняя строка описания класса *THuman* (*Листинг 105, часть 1*) выглядит следующим образом: *virtual void show();* эта запись обозначает, что метод *show()* является виртуальным (и все методы *show()* наследуемых классов, если такие там есть тоже будут виртуальными.

Создадим класс *TGroup* для хранения массива объектов нужного класса – *THuman, TStudent* или *Tteacher* соответственно (аналогично примеру *Листинг 104*). В этом классе закрытые свойства *count* и *people* для динамического массива указателей типа *THuman*, имеется два конструктора, деструктор и метод *show_people()*. Класс *TGroup* не наследуется от *THuman*, а только содержит в себе динамический массив объектов этого типа. Как видно из реализации конструктора по умолчанию (*Листинг 105, часть 2*), в нем выделяется память под массив указателей, а динамически создаваемые элементы этого массива могут иметь тип любой из классов-наследников *THuman, TStudent* или *Tteacher*.

Листинг 105 (часть 2)

```
TGroup::TGroup()
{    count= 6;
     people = new THuman*[count]; // выделение памяти

     people[0]= new TStudent(1,"Иван", "Иванов", 1996, true, "ТУСУР",
"ФЭТ", 2015, 0.0, "нанотехнологии и микросистемная техника");

     people[1]= new THuman(2,"Янис", "Петерс", 1997, true);

     people[2]= new TStudent(3,"Аврам", "Шмулинсон", 1996, true, "ТУСУР",
"ФЭТ", 2015, 0.0, "электроника и наноэлектроника");

     people[3]= new TTeacher(4,"Анна", "Попова", 1990, false, "ТУСУР",
"ФЭТ", 2014, 35000.0, "математика, физика, электроника, информатика");

     people[4]= new TStudent(5,"Айгюль", "Кабирова", 1997, false, "ТУСУР",
"ФЭТ", 2015, 0.0, "электроника и наноэлектроника");

     people[5]= new TStudent(6,"Марта", "Штейнер", 1997, false, "ТУСУР",
"ФЭТ", 2015, 0.0, "нанотехнологии и микросистемная техника"); }

TGroup::~TGroup()
{    for (int i=0; i<count; i++) // удаление динамических эл-тов массива
          delete(people[i]);
     delete[]people; // удаление массива указателей
}

void TGroup::show_people()
{ for (int i=0; i<count; i++)
     people[i]->show(); // вызов виртуального метода show() для элементов
}                       // массива в соответствии с классом элемента
```

Метод *show_people()* класса *TGroup* вызывает последовательно для каждого элемента метод *show()*, но, поскольку эта цепочка методов виртуальная, вызывается в каждом случае свой метод – *Thuman:show()*, *TStudent::show()*, или *Tteacher::show()*, в зависимости от типа элемента массива *people*.

Листинг 105 (часть 3)

```
TGroup G; // создание объекта типа TGroup
G.show_people(); // вызов методов show() для элементов массива people
```

```
1. Иван Иванов 1996 года рождения, мужчина. 19 лет.
ТУСУР // ФЭТ с 2015 года // 0
студент специальности: нанотехнологии и микросистемная техника

2. Янис Петерс 1997 года рождения, мужчина. 18 лет.

3. Аврам Шмулинсон 1996 года рождения, мужчина. 19 лет.
ТУСУР // ФЭТ с 2015 года // 0
студент специальности: электроника и наноэлектроника

4. Анна Попова 1990 года рождения, женщина. 25 лет.
С 2014 года работает в ТУСУР
на факультете ФЭТ, получает 35000 руб.
преподает следующие предметы:
математика
физика
электроника
информатика

5. Айгюль Кабирова 1997 года рождения, женщина. 18 лет.
ТУСУР // ФЭТ с 2015 года // 0
студент специальности: электроника и наноэлектроника

6. Марта Штейнер 1997 года рождения, женщина. 18 лет.
ТУСУР // ФЭТ с 2015 года // 0
студент специальности: нанотехнологии и микросистемная техника
```

Рис. 68. Результаты работы программы (Листинг 105)

11.4. Практическая работа № 12. Наследование

Продолжительность – 4 часа.

Максимальный рейтинг – 8 баллов.

Цель работы

Научиться создавать классы-потомки на основе базовых классов и описывать их свойства и методы. Научиться применять конструкторы базового класса в соответствующих конструкторах наследуемых классов. Освоить умение работать с абстрактными классами, дружественными и виртуальными функциями и научиться перегружать операторы.

Закрепить навыки конструирования классов, выделения памяти под динамические объекты класса, освобождения памяти, обращения к их свойствам и методам.

Задание на практическую работу

1. Задать базовый класс и классы-потомки на его основе, реализующие операции с данными, в соответствии с индивидуальным заданием (*Таблица 21*). Если возможно, реализовать *множественное наследование*.
2. В базовом классе обязательно создать *конструктор_по_умолчанию* и *конструктор_с_параметрами*, а также деструктор класса. Продемонстрировать применение в конструкторах классов-потомков *соответствующих* конструкторов базового класса.
3. Сделать базовый класс *абстрактным*. Продемонстрировать вызов методов абстрактного класса через объекты класса-потомка.
4. Продемонстрировать использование *дружественной* функции или класса.
5. Продемонстрировать реализацию *виртуальной* функции в базовом классе и в классах-потомках.
6. В отчете привести листинг программы, скриншоты тестирования программы, рисунок структуры класса. Листинг программы без комментариев не принимается.

Варианты индивидуальных заданий

Внимание! Варианты индивидуальных заданий составлены таким образом, чтобы продолжать выполнение индивидуального задания из предыдущей лабораторной работы (*Лабораторная работа № 11. Классы*). Рекомендуется выполнять тот же номер варианта индивидуального задания, что и в предыдущей работе (*Таблица 20*).

Таблица 21 Варианты индивидуальных заданий

№	Задание
1.	Классы, реализующие операции с матрицами (3x3), составленными из комплексных чисел[3]. Написать метод, вычисления определителя.
2.	Классы, производящие полный расчет треугольника[1]: вычисление длин его сторон, и углов между ними, а также определяющий, является ли треугольник равнобедренным, равносторонним и/или прямоугольным.
3.	Классы для хранения массива из 10 окружностей[1]. Написать метод, выбирающий пары пересекающихся окружностей и вычисляющий площадь их пересечения.
4.	Классы для работы с судейским секундомером[1]. Хранить время и отсчитанное число секунд для каждого круга (до 10) и для каждого спортсмена (до 10).
5.	Классы для построчного сравнения[1] двух файлов. Создать метод, определяющий, равны ли файлы и возвращающий файл несовпадений в противном случае.
6.	Классы для хранения байта (в виде битовых полей)[1]. Создать такой метод, чтобы ноль перемещался от разряда к разряду слева направо.
7.	Классы для планирования затрат на подарки к дням рождений[1] ваших друзей. Задать Имена друзей, планируемые подарки, их стоимость и выводить оставшиеся в этом году дни рождения и объем необходимых средств.

[3] *На базе класса, разработанного в индивидуальном задании по предыдущей лабораторной работе.*

8.	Классы для работы с расписанием событий на день[1] – ежедневник. Создать метод, выводящий на экран время, оставшееся до всех еще не прошедших событий.
9.	Классы для задания прямой траектории в трехмерном пространстве[1] по известному *направляющему вектору* ***x*** и *стартовой точке* ***A***. Создать метод, рассчитывающий, проходит ли траектория через введенную пользователем точку ***B***.
10.	Классы для хранения дней рождений[1] всех студентов группы, и вычисляющий количество дней до ближайшего. Вычислите суммарный день рождения группы.
11.	Классы для определения координат точки[1], движущейся по *Архимедовой спирали* с постоянной скоростью. Написать метод, определяющий, в каком квадранте находится точка в заданный момент времени.
12.	Классы для кодирования символьных строк[1], последовательно заменяя каждый символ кодовой комбинацией. Создать метод, производящие закодировать и раскодировать строку по заданному шифру.
13.	Классы для хранения дней рождений[1] всех студентов группы, и вычисляющий количество дней прожитых каждым. Вычислите сколько дней прожила вся группа.
14.	Классы для управления 4 однобайтными выходными портами микроконтроллера (МК) состояние порта хранить в виде битовых полей[1]. Создать метод, реализующий эффект «бегущего огонька» по всем портам микроконтроллера.
15.	Классы для работы с расписанием занятий на неделю[1]. Создать метод, выводящий на экран время, оставшееся до всех еще не прошедших сегодня занятий.
16.	Классы для описания движения метеорита в пространстве[1], определить, на каком расстоянии он пройдет от Земли (начало координат).
17.	Классы для описания движения планет в пространстве[1] по эллиптической орбите вокруг Солнца (начало координат). Вычислять расстояния от Земли (ищи на третьей планете) до других планет Солнечной системы.
18.	Классы, позволяющие определить, встретятся ли летящие из заданных точек по заданным векторами[1] траекториям космические корабли и на каком минимальном расстоянии друг от друга они пролетят.
19.	Классы для базы данных *отдела кадров* предприятия[1]. Обрабатывать информацию о должности, отделе, разряде и окладе сотрудника.
20.	Классы для кодирования символьных строк[1] последовательностью символов азбуки Морзе. Создать метод, позволяющий закодировать и раскодировать строку. Учитывать пробелы, концы слов.
21.	Классы управления взрывом ракеты, летящей по баллистической траектории и имеющей определенный радиус поражения[1]. Задав угол наклона траектории, направление и стартовую скорость снаряда, рассчитать момент взрыва для поражения цели с заданными координатами.
22.	Классы для хранения массива прямоугольных окон на экране[1]. Написать метод, выбирающий пары пересекающихся окон и вычисляющий площадь их пересечения.

Контрольные вопросы:

1. В чем смысл наследования? Для чего применяется спецификатор *protected*?
2. Как наследуются конструкторы и деструкторы? Что такое *абстрактный* класс?
3. Для чего применяются *виртуальные* методы?
4. Что такое *дружественные* классы и функции?

[1] *На базе класса, разработанного в индивидуальном задании по предыдущей лабораторной работе.*

12. Шаблоны

```
- Вот представь: у тебя есть 1000 рублей... Или нет, для круглого счета, пусть у тебя 1024 рубля...
```

12.1. Шаблоны функций

Шаблоны – это средство языка C++, предназначенное для создания обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию). В C++ возможно создание *шаблонов функций и классов*.

Шаблоны начинаются со слова *template*, после которого идут угловые скобки, в которых перечисляется список параметров. Каждому параметру должно предшествовать зарезервированное слово *class* или *typename*.

```
template <class T>
template <typename T>
template <typename T1, typename T2>
```

Ключевое слово *typename* говорит о том, что в шаблоне будет использоваться *встроенный тип данных*, такой как: *int, double, float, char* и т. д. А ключевое слово *class* сообщает компилятору, что в шаблоне функции в качестве параметра будут использоваться *пользовательские типы данных*.

Например, нам нужно запрограммировать функцию, которая выводила бы на экран элементы массива. Задача не сложная, но, чтобы написать такую функцию, мы должны задать тип данных массива, который будем выводить на экран. А если требуется, чтобы функция выводила массивы различных типов – *int, double, float* или *char*? Можно воспользоваться перегрузкой функций (*см. п. 4.2*), но придется написать целых 4 функции, которые отличаются только заголовком функции, тело у них *абсолютно одинаковое*!

Для этого в *C++* и придуманы шаблоны функций. Мы создаем один шаблон, в котором как бы закладываем все возможные типы данных. А какой конкретно тип данных использовать в данной реализации, будет определено позже при компиляции функции.

Листинг 106 (часть 1)

```
template <typename T>
void print_array(const T *array, int count)
{    for (int ix = 0; ix < count; ix++)
         cout << array[ix] << " ";
     cout << endl;
}
```

В листинге (*Листинг 106, часть 1*) перед объявлением функции стоит запись *template <typename T>*. Как раз эта запись и говорит о том, что функция *print_array()* на самом деле является шаблоном функции, так как в первом параметре *print_array()* стоит тип данных *const T**. Это определение шаблона с одним параметром – *T*, причем этот параметр будет иметь один из встроенных типов данных, так как указано ключевое слово *typename*.

По сути *T* – это даже не тип данных, это зарезервированное место под любой встроенный тип данных. То есть когда выполняется вызов этой функции, компилятор анализирует параметр шаблонированной функции и *создает экземпляр* этой функции для соответственного типа данных: *int, char* и так далее. Следующий пример

(*Листинг 106, часть 2*) демонстрирует универсальную подпрограмму поиска максимального элемента в массиве чисел с использованием шаблона *typename*.

Листинг 106 (часть 2)

```
// шаблон функции для поиска максимального значения в массиве
template <typename myType>
myType searchMax(const myType *array, int size)
{    myType max = array[0]; // максимальное значение в массиве
     for (int ix = 1; ix < size; ix++)
          if (max < array[ix]) max = array[ix];
     return max;
}
```

Применение созданных подпрограмм и удобство использования шаблонов иллюстрирует пример (*Листинг 106, часть 3*) при помощи четырех статических массивов различного типа:

Листинг 106 (часть 3)

```
void _tmain(int argc, _TCHAR* argv[])
{       int iArray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        double dArray[6] = {1.2345, 2.234, 6.1545, 3.57, 4.67876, 5.346};
        float fArray[5] = {1.34, 2.37, 3.23, 5.879, 4.8};
        char cArray[8] = {"MARSIAN"};

        cout << "\nМассив типа int: "; print_array(iArray, 10);
        cout << searchMax(iArray, 10);

        cout << "\nМассив типа double: "; print_array(dArray, 6);
        cout << searchMax(dArray, 6);

        cout << "\nМассив типа float: "; print_array(fArray, 5);
        cout << searchMax(fArray, 5);

        cout << "\nМассив типа char: ";print_array(cArray, 8);
        cout << searchMax(cArray, 8);
        cin.get();
}
```

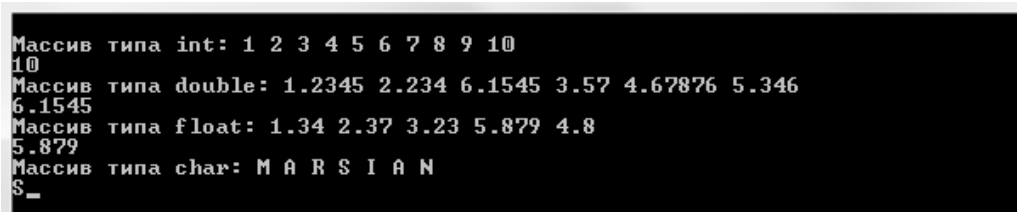

Рис. 69. Результаты работы программы (Листинг 106)

Шаблоны классов будут рассмотрены ниже в применении к динамическим структурам.

12.2. Динамические структуры. Список, очередь, стек

Созданию *динамических структур данных* посвящен значительный объем исследований [*1 - 4, 9*]. Цель этих исследований состоит в конструировании таких способов хранения данных, чтобы они позволяли оптимизировать место на диске и в памяти компьютера, а также обладали свойствами быстрого поиска и/или сортировки элементов.

Нам уже встречались динамические структуры данных при рассмотрении материала о построении *двумерных динамических массивов* (*см. п. 5.6.2*). Массивы удобно использовать в том случае, когда наперед известно точное количество элементов для хранения, если же число элементов заранее не известно, то приходится

или создавать заведомо больший массив, или увеличивать массив в процессе работы, постоянно переприсваивая элементы массивов из меньшего в больший. В первом случае мы нерационально обращаемся с памятью, во втором – со временем. Избежать этого позволяет самая простая динамическая структура – *список*. Рассмотрим следующие классы:

Листинг 107 (часть 1)

```
class node // звено списка
{    private:
          int data; // хранимые данные
          node* next; // адрес следующего звена списка
          node() { data= 0; next= NULL; } // абстрактный класс
          node(int d) { data= d; next= NULL; }
          ~node() { next= NULL; }
     public:
          friend class list; // дружественный класс
};
// ---------------------------------------------------------------------------
class list // список
{    private:
          node* start; // указатель на начало списка
     public:
          list() { start= NULL; }
          ~list();
          void add_to_start(int d); // добавить элемент в начало
          void show();
};
```

Принцип хранения очень прост. Элемент списка *node* (*Листинг 107, часть 1*) состоит из двух структурных ячеек: первая – для хранения данных (*data*), а вторая – для хранения указателя на следующий элемент списка (*next*). При создании объекта класса *node* указатель *next* указывает на *NULL*.

Список *list* представляет собой цепочку, состоящую из динамически создаваемых звеньев *node*, каждое из которых хранит в поле *data* требуемые данные, а в поле *next* – адрес следующего звена списка. Последнее звено указывает на *NULL*, это и обозначает конец списка. Список *list* фактически хранит только указатель *start* на первое звено (*Рис. 70*), а все звенья создаются динамически и включаются в список.

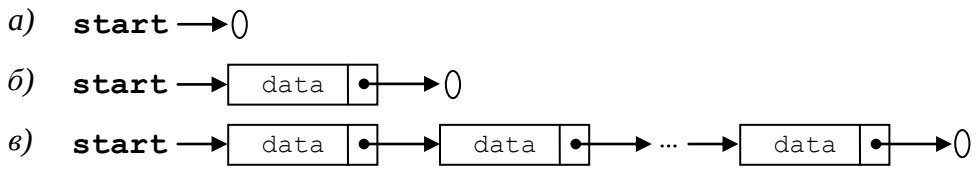

Рис. 70. Список. Принцип хранения данных

Как обращаться к элементам такой конструкции? – Последовательно переходя от элемента к элементу. Ведь список пока не имеет оператора *operator [ ]* – его можно будет написать позже. А пока рассмотрим на примере ниже (*Листинг 107, часть 2*) реализацию метода *show()*, последовательно выводящего на экран элементы списка.

Листинг 107 (часть 2)

```
void list::show()
{       node *p = start;
        while (p) // пока указатель p не равен NULL
        {       cout << p->data << " ";
                p = p->next; } // перейти к следующему элементу списка
}
```

В первой строке подпрограммы указателю *p* присваивается адрес начала списка – указатель *start*. Затем в цикле (пока указатель *p* не станет равным *NULL*) производится вывод данных *p->data* на экран и переход к следующему элементу списка при помощи действия *p = p->next*. Эти действия повторяются, пока не дойдем до конца цепочки. Все очень просто.

Рассмотрим два тонких момента. Во-первых, почему методы класса *list* имеют доступ к закрытым свойствам класса *node*? – Потому что класс *list* – дружественный. Во-вторых, как осуществляется переход к следующему звену списка? Допустим, указатель *p* хранит в себе адрес какого-то звена типа *node* списка *list* (указывает на него). Тогда что такое *p->next*? – Это адрес следующего элемента в цепочке, а присваивание *p = p->next* просто изменяет адрес в ячейке *p* с текущего на следующий.

Не сложнее выглядит процедура добавления элемента *add_to_start(int d)* в начало списка *list*, суть этой подпрограммы демонстрирует *Рис. 71*. Сначала при помощи оператора *new*, создается очередное звено списка – динамический объект *p* типа *node*. Затем перенаправляются два указателя: указателю *p->next* присваивается адрес *start*, а указателю *start* присваивается значение *p* (он теперь первый). Готово.

Листинг 107 (часть 3)

```
void list::add_to_start(int d)
{       node* p = new node(d); // создать динамический объект node
        p->next= start;
        start= p;
}
```

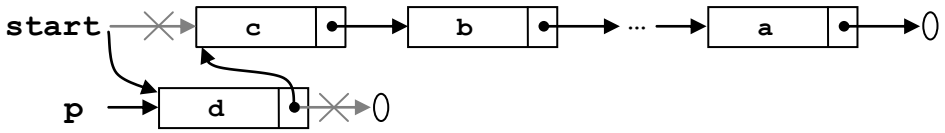

Рис. 71. Список. Добавление узла

Обратите внимание на использование списка в программе на примере (*Листинг 107, часть 4*). Сначала создается динамический объект *L* типа *list*. Это пустой список – в нем нет ни одного звена и указатель *start==NULL*. Затем в цикле при помощи метода *add_to_start( )* в начало списка добавляется по одному случайному значению. После чего список выводится на экран методом *show( )*.

Листинг 107 (часть 4)

```
void _tmain(int argc, _TCHAR* argv[])
{       setlocale (LC_ALL, "Russian");
        srand(time(NULL));

        list *L= new list();
        for (int i=0; i<10; i++)
                L->add_to_start(rand()%1000);
        L->show();

        cin.get();
}
```

```
899 341 619 772 142 543 765 490 443 118
```

Интересно рассмотреть, как размещаются элементы списка в памяти. Для этого создадим подпрограмму *list::show_address( )*, приведенную ниже (*Листинг 107, часть 5*), которая полностью аналогична методу *list::show( )*, за одним исключением – этот метод выводит на экран не только значения *p->data*, хранимые в звеньях списка, но и адреса *p* всех ячеек типа *node* списка, а также адреса следующих ячеек, хранящиеся в указателях *p->next*.

Результаты работы этой подпрограммы приведены на рисунке (*Рис. 72*), обратите внимание на адрес start и на адрес *p->next* последней ячейки списка.

Листинг 107 (часть 5)

```
void list::show_address()
{    printf("\nуказатель start= %p\n", start);
     node *p = start;
     while (p) // пока указатель p не равен NULL
     {    printf("ячейка с адресом %p содержит данные [data=%3d next=%p]\n",
          p, p->data, p->next);
          p = p->next; } // перейти к следующему элементу списка
}
```

```
830 894 115 704 708 537 427 947 2 217
указатель start= 00589000
ячейка с адресом 00589000 содержит данные [data=830 next=00588FB8]
ячейка с адресом 00588FB8 содержит данные [data=894 next=00588F70]
ячейка с адресом 00588F70 содержит данные [data=115 next=00588F28]
ячейка с адресом 00588F28 содержит данные [data=704 next=00588EE0]
ячейка с адресом 00588EE0 содержит данные [data=708 next=00588E98]
ячейка с адресом 00588E98 содержит данные [data=537 next=00588E50]
ячейка с адресом 00588E50 содержит данные [data=427 next=00584398]
ячейка с адресом 00584398 содержит данные [data=947 next=00584350]
ячейка с адресом 00584350 содержит данные [data=  2 next=00584308]
ячейка с адресом 00584308 содержит данные [data=217 next=00000000]
```

Рис. 72. Список. Размещение в памяти

Важным является осознание того, что каждое звено списка создается отдельно оператором *new* – это динамический объект, а следовательно, он хранится в области динамической памяти – в *куче*. В нашем примере адреса ячеек списка располагаются последовательно один за другим (*Рис. 72*), но это не обязательно – в другом случае эти адреса могут быть разбросаны по разным местам динамической памяти. В связи с этим адрес *p+1* (сдвинутый на единицу относительно *p*) это вовсе не обязательно адрес следующей ячейки *p->next*.

Осталось рассмотреть реализацию деструктора класса. Разумеется, нельзя просто уничтожить указатель *start*, ведь в этом случае все остальные элементы списка останутся в памяти компьютера, и обнаружить их адреса станет невозможно, т.к. *start* удален, а адрес каждого звена списка хранится в такой же предыдущей ячейке.

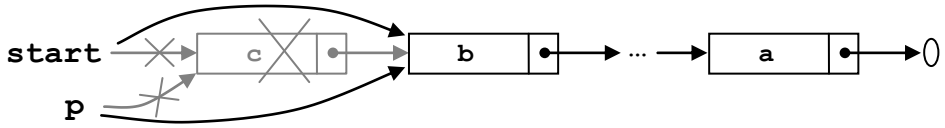

Рис. 73. Список. Удаление узла

Для удаления списка (*Листинг 107, часть 6*) последовательно в цикле (до тех пор, пока список не кончится) повторяются 3 операции: сохранить в указателе *start* адрес *p->next*, удалить *p*, перейти по адресу *p->next*, сохраненному в *start*.

Листинг 107 (часть 6)

```
list::~list()
{       node* p= start;
        while (p) // пока указатель p не равен NULL
        {       start= p->next;
                delete p; // удалить звено
                p=start; }
}
```

В рассмотренном нами примере (*линейного, однонаправленного*) списка добавление и удаление элемента осуществлялось с начала (*start*) цепочки звеньев (*Рис. 74, а*). Это не единственный способ доступа к элементам динамической структуры, рассмотрим другие.

Способы доступа к данным: стек и очередь

По способу добавления элементов в динамическую структуру и удаления из нее списки подразделяются на *структуры с независимым доступом* (такие как массив, где можно обратиться к любому элементу списка по номеру) и *структуры с последовательным доступом – стек и очередь*. [9]

Стек – это динамическая структура данных, составленная по принципу *«последним вошёл – первым вышел»* (*LIFO, Last In First Out*). Для стеков обязательными операциями являются: операция постановки в стек *push( )*, где *x*-элемент, который засылается *в вершину* стека и операция извлечения *pop( )*, возвращающая последний поставленный элемент из вершины.

Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

Очередь – структура данных с дисциплиной доступа к элементам «*первый пришёл – первый вышел*» (*FIFO, First In First Out*). Добавление элемента (принято обозначать *enqueue( )* – поставить в очередь) возможно лишь в конец очереди, выборка – только из начала очереди (что принято называть *dequeue( )* – убрать из очереди), при этом выбранный элемент из очереди удаляется.

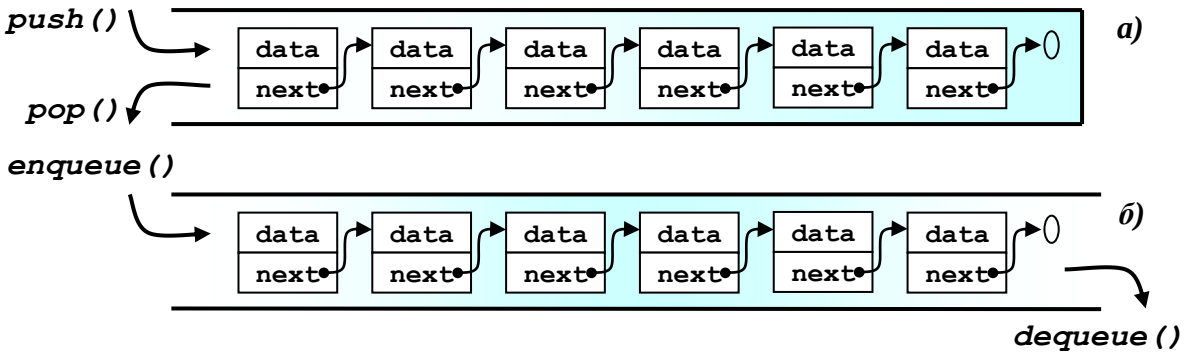

Рис. 74. Способы добавления и удаления элементов: а) стек; б) очередь

Проанализировав предыдущий пример (*Листинг 107*), можно убедиться, что доступ к элементам списка *list* организован по технологии *LIFO* – стек.

12.3. Сложные динамические структуры данных

Мы рассмотрели в *п. 12.2* (*Листинг 107*) самый простой из вариантов списка – *простой список*, называемый в литературе так же *однонаправленным списком*. Но динамических структур много больше и их исследование – отдельная наука. Перечислим кратко основные виды структур.

Двунаправленный список

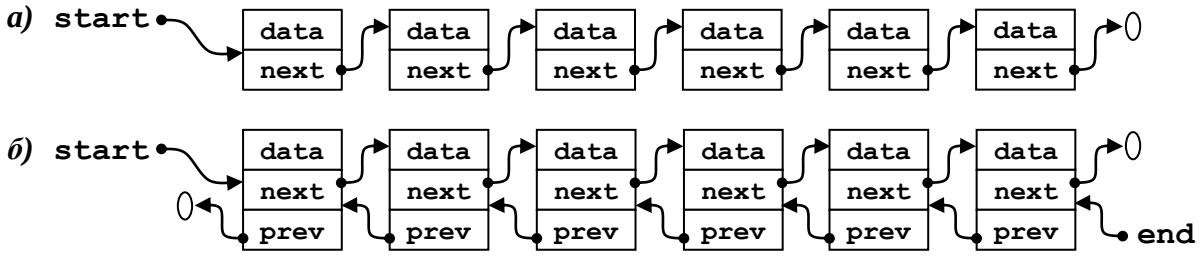

Рис. 75. Список: а) однонаправленный; б) двунаправленный

Двунаправленный список имеет в каждом элементе (узле) два указателя: один – *next* – указатель на последующий элемент списка, а второй – *prev* – указатель на

предыдущий. Удобство такого списка состоит в том, что по нему можно двигаться в обоих направлениях – как от указателя *start* до *end* (совершая в цикле переход *node=node->next*), так и от конца к началу – от указателя *end* до *start* (передвигаясь в цикле *node=node->prev*).

Листинг 108

```
class node // звено двунаправленного списка
{    private:
          char data; // хранимые данные
          node* next; // адрес следующего звена списка
          node* prev; // адрес предыдущего звена списка
          // ...
};
// ---------------------------------------------------------------------------
class list // список
{    private:
          node* start; // указатель на начало списка
          node* end; // указатель на конец списка
          // ...
};
```

Проектирование и реализацию методов этих классов предлагается проделать самостоятельно в качестве тренировки.

Кольцевые списки

Кольцевые списки бывают как однонаправленные, так и двунаправленные, суть *кольцевых списков* (*ring*) в том, что последний элемент указывает не на *NULL*, а на первый элемент списка.

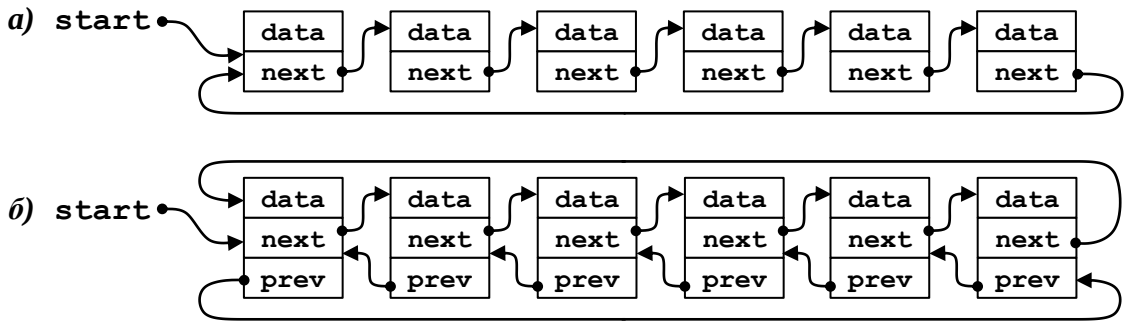

Рис. 76. Кольцевые списки: а) однонаправленный; б) двунаправленный

На рисунке (*Рис. 76*) приведена структура однонаправленного (*Рис. 76, а*) и двунаправленного (*Рис. 76, б*) кольцевых списков. Имея такую структуру можно двигаться в одном (или в двух) направлениях бесконечно – по кольцу. Условие полного обхода и обработки элементов всего списка состоит не в том, чтобы встретить *NULL*, а в том, чтобы встретить начало – указатель *start*.

Если двунаправленный кольцевой список (*Рис. 76, б*) реализован, как показано в примере (*Листинг 108*), то методы добавления элемента *list::add_to_end( )* и прохода по списку могут быть построены следующим образом:

Листинг 109

```
void list::add_to_end(char d) // добавление в конец
{    node* p = new node(d); // создать динамический объект node
     end->next= p;
     p->next= start;
     p->prev= end;
     start->prev=p;
     end= p; } // теперь это - конец списка
```

```
void list::show() // вывод на экран от конца к началу
{    node* p= end;
     do // повторять пока указатель p не дошел до начала (start)
     {    cout << p->data << " ";
          p = p->prev; // перейти к предыдущему элементу списка
     } while (p!=start)
}
```

Упорядоченное хранение данных

Рассмотренные выше структуры данных имели линейную конструкцию: после одного элемента списка располагался один следующий элемент и т.д. Кроме того, списочные структуры предназначены для хранения неупорядоченных данных – элементы располагаются в такой последовательности, в какой помещены в список (по технологии очередь или стек), что, разумеется, существенно усложняет подпрограммы поиска элементов и заставляет расходовать большое количество вычислительных ресурсов – памяти и времени.

Проблема поиска породила целое семейство алгоритмов [9] и технологий поиска и индексирования. В содержание данного курса не входит материал по алгоритмам сортировки и поиска, поэтому лишь кратко перечислим основные принципы ускоренного доступа к данным:

- алгоритмы быстрого поиска;
- предварительная сортировка и упорядоченное хранение данных в структуре;
- индексация данных.

Смысл индексации состоит в том, чтобы вместе с данными, хранимыми в структуре размещать упорядоченные индексы, позволяющие ускорить сортировку и поиск.

Бинарное дерево

Из вышесказанного можно сделать вывод о том, что для более быстрого доступа к данным можно хранить их в списке уже в упорядоченном виде. То есть помещать элемент в список не в конец (или начало), как в стеке и очереди, а сразу за тем элементом, который «меньше» и/или перед тем, который «больше» данного.

Кроме того, существуют структуры данных, специально созданные для упорядоченного хранения. Элементы таких структур, как *бинарное дерево* (*Рис. 77*) принципиально предназначены для размещения данных в систематизированном виде. Бинарное дерево является *рекурсивной структурой*, поскольку каждое его поддерево само является бинарным деревом и, следовательно, каждый его узел в свою очередь является корнем дерева. Узел дерева, не имеющий потомков, называется листом [9].

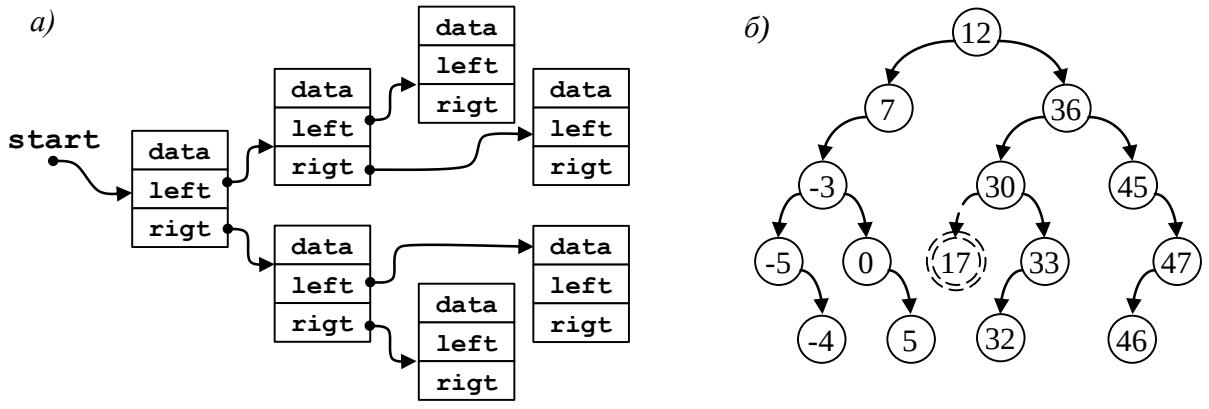

Рис. 77. Бинарное древо

Как видно, каждый элемент бинарного (двоичного) дерева имеет две ветви наследников – «*left*» и «*right*» (или иначе «*больше*» и «*меньше*»). Любой элемент, при добавлении в дерево, размещается не где попало, а по следующему алгоритму –

начиная с *корня* дерева новый элемент, сравнивается с полем *data* очередного узла. Если этот элемент больше, то осуществляется переход к правому узлу, если меньше – к левому, если узла справа (или слева) нет, то на это место и размещается новый элемент.

Аналогично производится поиск элементов в упорядоченном бинарном дереве – существенно быстрее, чем в списке [9].

Листинг 110 (часть 1)

```
class TNode // звено (узел) бинарного дерева
{    public:
          int data; // хранилище данных
          TNode *left, *right; // адреса левой и правой ветвей
          TNode () { left= right= NULL; data= 0; }
          TNode (int dat) { left= right= NULL; data= dat; }
          ~TNode() { left= right= NULL; }
};
// ---------------------------------------------------------------------
class TTree // бинарное дерево
{    public:
          TNode* start; // корень дерева
          TTree () { start= NULL; }
          ~TTree ();
          void add(TNode **ptr, int dat); // добавление элемента
          void show(TNode *ptr); // вывод на экран
};
```

Поскольку бинарное дерево является рекурсивной структурой (каждое его поддерево само является бинарным деревом), то и операции по работе с элементами дерева логично реализовать рекурсивным способом (*Листинг 110, часть 2*). Например, метод *TTree::add(TNode **ptr, int dat)* добавляет элемент на место пустого узла. Если же узел не пустой, то в зависимости от величины элемента, вызывается тот же самый метод *add* для правого *add(&((*ptr)->right), dat)* или левого *add(&((*ptr)->left), dat)* поддерева.

Аналогично и метод *void TTree::show(TNode *ptr)* сначала рекурсивно вызывает сам себя для вывода на экран левого поддерева *show(ptr->left)*, затем выводит текущий узел, а затем рекурсивно вызывает сам себя для вывода правой ветви *show(ptr->right)*.

Листинг 110 (часть 2)

```
void TTree::add( TNode **ptr, int dat)
{    if(!(*ptr)) // ptr==NULL
     {    *ptr= new TNode(dat); }
     else // ptr!=NULL
     {    if((*ptr)->data > dat)
               add(&((*ptr)->right), dat); // рекурсивный вызов левого дерева
          else // рекурсивный вызов правого дерева
               add(&((*ptr)->left), dat); }
}
// ---------------------------------------------------------------------
void TTree::show(TNode *ptr)
{    if(ptr)
     {    show(ptr->left); // рекурсивный вызов левого поддерева
          printf("ячейка с адресом %p [*data=%3d left=%p right=%p]\n",
               ptr, ptr->data, ptr->left, ptr->right);
          show(ptr->right); // рекурсивный вызов правого поддерева
     }
}
```

Пример ниже (*Листинг 110, часть 3*) иллюстрирует размещение элементов бинарного дерева в памяти в соответствии со схемой, приведенной на *Рис. 77, б*. На этом же примере (*Рис. 78*) можно видеть очередность вывода узлов дерева – элементов

TNode в соответствии с рекурсивной последовательностью вызовов подпрограммы *show( )*.

Листинг 110 (часть 3)

```
TTree *s= new TTree(); // создание дерева

s->add(&s->start,12);      s->add(&s->start,7); // добавление элементов
s->add(&s->start,36);      s->add(&s->start,-3);
s->add(&s->start,30);      s->add(&s->start,-5);
s->add(&s->start,45);      s->add(&s->start,0);
s->add(&s->start,33);      s->add(&s->start,32);
s->add(&s->start,5);       s->add(&s->start,-4);
s->add(&s->start,47);      s->add(&s->start,46);
s->add(&s->start,17);

s->show(s->start); // вывод на экран
```

```
ячейка с адресом 006A90D8 [*data= 47 left=00000000 right=006A9120]
ячейка с адресом 006A9120 [*data= 46 left=00000000 right=00000000]
ячейка с адресом 006A8F28 [*data= 45 left=006A90D8 right=00000000]
ячейка с адресом 006A4398 [*data= 36 left=006A8F28 right=006A8E98]
ячейка с адресом 006A8FB8 [*data= 33 left=00000000 right=006A9000]
ячейка с адресом 006A9000 [*data= 32 left=00000000 right=00000000]
ячейка с адресом 006A8E98 [*data= 30 left=006A8FB8 right=006A9168]
ячейка с адресом 006A9168 [*data= 17 left=00000000 right=00000000]
ячейка с адресом 006A4308 [*data= 12 left=006A4398 right=006A4350]
ячейка с адресом 006A4350 [*data=  7 left=00000000 right=006A8E50]
ячейка с адресом 006A9048 [*data=  5 left=00000000 right=00000000]
ячейка с адресом 006A8F70 [*data=  0 left=006A9048 right=00000000]
ячейка с адресом 006A8E50 [*data= -3 left=006A8F70 right=006A8EE0]
ячейка с адресом 006A9090 [*data= -4 left=00000000 right=00000000]
ячейка с адресом 006A8EE0 [*data= -5 left=006A9090 right=00000000]
```

Рис. 78. Бинарное дерево. Размещение элементов в памяти

12.4. Шаблоны классов

В рассмотренном выше примере (*см. п. 12.2*) элементами хранения данных в поле *data* списка является символьный тип *int*. Теоретически, класс для хранения *любых элементов* в виде простого списка ничем не будет отличаться от класса описанного выше кроме типа поля *data*. Разумно было бы применить шаблон (*template*) для создания универсального класса, позволяющего организовать хранение объектов произвольного типа [*1*].

```
template <аргумент1, аргумент2, ...>
class имя_класса
{
  ... внутренности ...
};
```

Для примера напишем шаблон класса для хранения элементов в массиве. Массив может оперировать различными типами данных, такими как *int, long, double* и т.д.

Листинг 111 (часть 1)

```
#include <stdlib.h>
#include <iostream.h>
int pos=0; // для хранения позиции внутри массива

template <class T> // шаблон с класса с параметром T
class Massiv
{       T Array[10]; // пусть тип объявленного массива - из шаблона
        int count; // счетчик элементов
  public:
        void Add(T &); // параметр - ссылка на переменную типа шаблон
        void Show();
};
```

Реализация методов класса приведена ниже.

Листинг 111 (часть 2)

```
template<class T> void Massiv<T>::Add(T &x)
{       Array[pos]=x; //присваиваем текущий элемент
        pos++; // переходим к позиции следующего элемента
        count=pos; // счетчик равен текущей позиции элемента
}
template <class T> void Massiv<T>::Show()
{    for (int i=0; i<count; i++)
         cout << Array[i] << "t"; // вывод всех данных массива на экран
     cout << endl;
     pos=0; // после вывода данных обозначаем, что текущий элемент нулевой
}
```

Класс написан так, как будто он оперирует элементами типа *T*. Хотя на самом деле это не так. Если объявить экземпляр класса для типа *int*, будет сгенерирован исходный код класса, в котором тип *T* будет заменен на тип *int*. Следует также помнить, что шаблон нельзя откомпилировать заранее, т.е. он должен всегда находиться во включаемом *h-файле*, а не исходном *cpp-файле.*

Листинг 111 (часть 3)

```
void _tmain(int argc, _TCHAR* argv[])
{       Massiv <int> Arr; // переменная нашего класса, тип шаблона – int
        Arr.Add(100); // добавляем элементы
        Arr.Add(200);
        Arr.Add(300);
        Arr.Show(); // отображаем массив на экране

        Massiv <char *> A; // переменная нашего класса, тип шаблона – char*
        Arr2.Add("Строка"); // добавляем данные
        Arr2.Add("Начинаю понимать");
        Arr2.Add("УРА");
        Arr2.Show(); // отображаем массив на экране
}
```

При объявлении каждого экземпляра класса *Massiv* компилятором генерируется отдельная копия класса, в котором вместо типа *T* подставляется реально требуемый тип. Фактически можно считать, что у нас на самом деле имеется два разных класса - для типа *int* и *char** (и два разных исходных кода для каждого класса).

Для того чтобы еще яснее показать, что мы используем два разных класса (и заодно упростить синтаксис), можно использовать оператор *typedef:*

```
typedef Massiv <int> MassivInt;
typedef Massiv <char *> MassivCharPtr;
```

Теперь мы можем объявлять наши массивы так:

```
MassivInt M1;
MassivCharPtr M2;
```

Внешне все выглядит так, как будто на самом деле существует два отдельных класса – *MassivInt* и *MassivCharPtr*.

Рассмотрим создание шаблона такого класса на примере бинарного дерева.

Построение бинарного дерева с использованием шаблона

Сначала мы должны определить структуру для создания корня и узлов дерева (*Листинг 112*). Здесь, как и раньше (*Листинг 110*) поля *left* и *right* - это указатели на потомков данного узла, а поле *data* предназначено для хранения информации. Напишем шаблонную рекурсивную функцию *void makeTree(TNode <T> **pp, T x)*, которая будет вызываться при создании узла дерева, эта функция добавляет элемент *x* к дереву,

учитывая величину *x*. При этом новый узел дерева создается упорядоченно в найденной ветви.

Листинг 112 (часть 1)

```
template <class T>
struct TNode
{       T data;
        TNode *left, *right;
        TNode() { left = right = NULL; }
}; // ---------------------------------------------------------------------
template <class T>
void makeTree(TNode <T> **pp, T x)
{       if(!(*pp)) // это лист - создать поддерево
        {       TNode <T> *p = new TNode<T>();
                p->data = x;
                *pp = p; }
        else // это ветвь - перейти к соответствующему поддереву
        {       if((*pp)->date > x)
                        makeTree(&((*pp)->left), x);
                else
                        makeTree(&((*pp)->right), x);
        }
} // ---------------------------------------------------------------------
template <class T>
void walkTree(TNode<T>* p)
{       if(p)
        {       walkTree(p->left);
                cout << p->data << ' ';
                walkTree(p->right);
        }
}
```

Функция *walkTree(TNode<T>* p)*, выполняющая обход дерева, позволяет перебрать все элементы, содержащиеся в дереве. В приведенной ниже реализации функция обхода дерева будет просто выводить на экран значение поля *data* каждого узла дерева (включая его корень).

При работе с деревьями обычно используются рекурсивные алгоритмы. Использование рекурсивных функций менее эффективно с точки зрения затрат времени и памяти ОЗУ, поскольку многократный вызов функции расходует системные ресурсы. Тем не менее, в данном случае использование рекурсивных функций является оправданным, поскольку нерекурсивные функции для работы с деревьями гораздо сложнее и для написания, и для восприятия кода программы.

12.5. Практическая работа № 13. Шаблоны. Динамические структуры.

Продолжительность – 4 часа.
Максимальный рейтинг – 8 баллов.

Цель работы

Получить компетенции в части создания динамических структур данных (списки, кольца, деревья); получить компетенции по методам доступа к данным (стек, очередь); научиться создавать унифицированный программный код (шаблоны функций, классов); освоить умения и навыки в технологии поиска и сортировки.

Закрепить навыки конструирования классов, выделения памяти под динамические объекты класса, освобождения памяти, обращения к их свойствам и методам.

Задание на практическую работу

1. Создать класс, реализующий динамическую структуру хранения данных в соответствии с индивидуальным заданием (*Таблица 22*).
2. Задать шаблон класса, для того, чтобы созданная динамическая структура могла хранить данные различного типа. Продемонстрировать применение созданной динамической структуры как минимум с двумя различными реализациями типа хранимых данных.
3. В созданном классе реализовать конструктор, деструктор, *set*- и *get*- методы. Доступ к элементам динамической структуры организовать в виде метода в соответствии с индивидуальным заданием (*Таблица 23*).
4. Написать метод, реализующий поиск элемента структуры по содержанию.
5. Написать методы, производящие вывод на экран, вывод в файл, чтение из файла всех элементов динамической структуры.
6. В отчете привести листинг программы, скриншоты тестирования программы, рисунок структуры класса. Листинг программы без комментариев не принимается.

Варианты индивидуальных заданий

Таблица 22 Варианты структуры

№	Структура хранения данных
1.	Двунаправленный линейный список
2.	Одномерный динамический массив переменной длины
3.	Однонаправленный линейный список
4.	Двунаправленный кольцевой список
5.	Упорядоченное бинарное дерево
6.	Однонаправленный кольцевой список

Таблица 23 Варианты способа доступа к элементам

№	Способ доступа к элементам структуры
1.	Очередь, размещение элементов с хвоста
2.	Стек, размещение элементов с головы
3.	Упорядоченное размещение элементов по возрастанию
4.	Очередь, размещение элементов с головы
5.	Стек, размещение элементов с хвоста
6.	Упорядоченное размещение элементов по убыванию
7.	Индексация элементов, доступ по индексу

Контрольные вопросы:

1. В чем состоит особенность динамических структур данных?
2. Опишите принцип добавления (удаления) элемента в динамическую структуру.
3. Каков механизм перебора всех элементов для конечного и кольцевого списков?
4. Как осуществляется переход от одного элемента списка к другому?
5. Как создаются (удаляются) объекты классов – динамических структур?
6. Каков механизм реализации перехода от элемента к элементу двунаправленного списка?
7. Как реализуются упорядоченные динамические структуры – деревья?
8. Можно ли создать дерево с произвольным количеством ветвей?
9. В чем отличие шаблона *template <class>* от шаблона *template <typename>?*

Список рекомендуемой литературы

1. Бьярне Страуструп Программирование: принципы и практика использования C++, исправленное издание. / Programming: Principles and Practice Using C++. — М.: «Вильямс», 2011. — С. 1248. — ISBN 978-5-8459-1705-8
2. Герберт Шилдт C/C++ Справочник программиста. — М.: «Вильямс», 2003. — С. 432. — ISBN 5-8459-0459-5
3. Эндрю Кёниг, Барбара Э. Му Эффективное программирование на C++. Серия книг "C++ In-Depth"++. — М.: «Вильямс», 2002. — С. 384. ил. — ISBN 5-8459-0350-5
4. Стэнли Б. Липпман, Жози Лажойе. Язык программирования C++. Вводный курс. – Спб.: Невский Диалект, ДМК пресс, 2002. – С. 444., ил. - ISBN 5-7940-0070-8, 5-94074-040-5
5. Богуславский А.А., Соколов С.М. Основы программирования на языке Си++: Для студентов физико-математических факультетов педагогических институтов. – Коломна: КГПИ, 2007.
6. Демидович Е.М. Основы алгоритмизации и программирования. Язык СИ : Учебное пособие. – СПб.: БХВ-Петербург, 2006.
7. Дэвис С. C++ для «чайников». – К. : Диалектика, 2005.
8. Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб: Питер, 2007.
9. Дональд Кнут Искусство программирования, том 3. Сортировка и поиск = The Art of Computer Programming, vol.3. Sorting and Searching. — 2-е изд. — М.: «Вильямс», 2007. — С. 824. — ISBN 0-201-89685-0