Министерство науки и высшего образования РФ
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++

Учебное пособие

ТОМСК 2022

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 С++: Учебное пособие / С. Г. Михальченко; Томский государственный университет систем управления и радиоэлектроники, Кафедра промышленной электроники – Томск: ТУСУР, 2022. – 187 с. : ил., табл. – Библиогр.: с. 185.

Учебное пособие предназначено для студентов любых форм обучения. Оно может применяться для всех технических направлений подготовки.

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

Руководство может быть использовано для проведения лекционных занятий и для самоподготовки.

Для освоения дисциплины *Информационные технологии* достаточно знаний, полученных 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++*.

1.1. 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 непросто. 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 (отмеченные символом `'//'` в начале строки).

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 C++ 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.

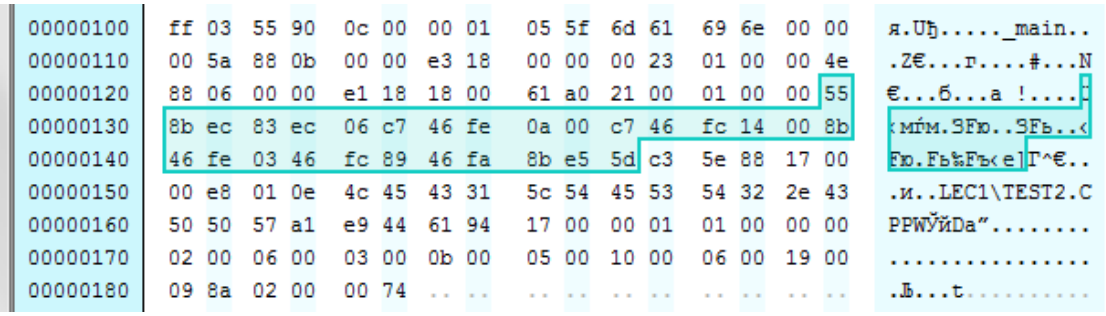

Рис. 2. Фрагмент программы (Листинг 2) в машинных кодах

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

Листинг 3

```
_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
```

4 этап. Линковка (компоновка)

Полученный объектный файл (а их может быть несколько) отдаётся *линковщику (link)*. Линковщик склеивает между собой все поданные ему файлы `*.obj` 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. 3*). 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 main()`.

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

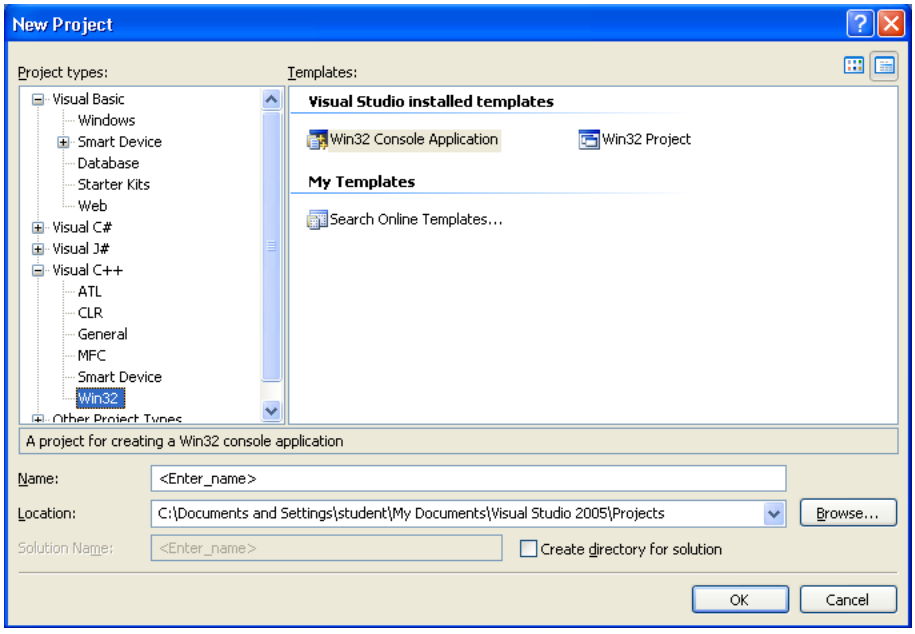

Рис. 3. Создание проекта на Visual C: Выбор шаблона, задание имени проекта

1.3. Структура программы на языке C++

Программа на языке C++ имеет следующую структуру [1, 4]:

Листинг 4

```
#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 5 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 некоторых компиляторах заголовок программы `main()`выглядит как на примере (*Листинг 4*): `int _tmain(int argc, char* argv[])`. Не обращайте пока на это внимание – в нужное время вы все поймете. Пока достаточно понимать, что это входные параметры запуска нашей программы из командной строки консольного окна.

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 5

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

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. 4*) можно запустить ее на выполнение.

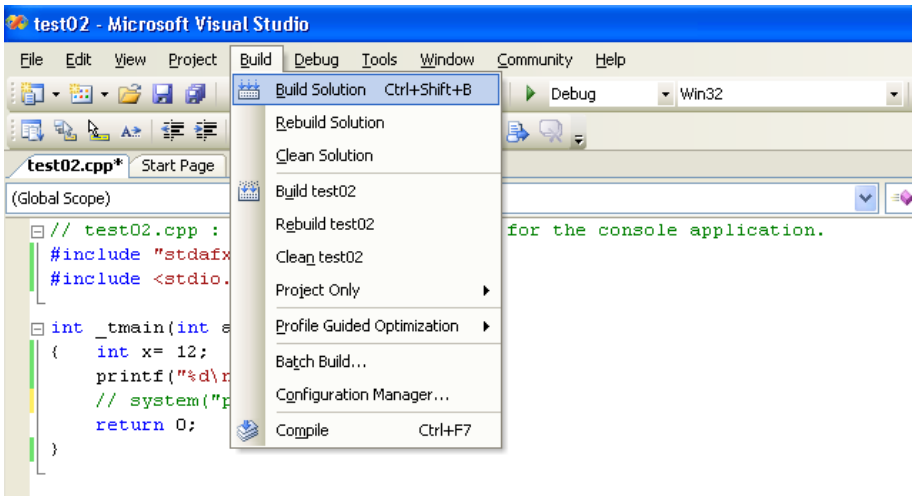

Рис. 4. Запуск созданного программного кода на компиляцию и выполнение

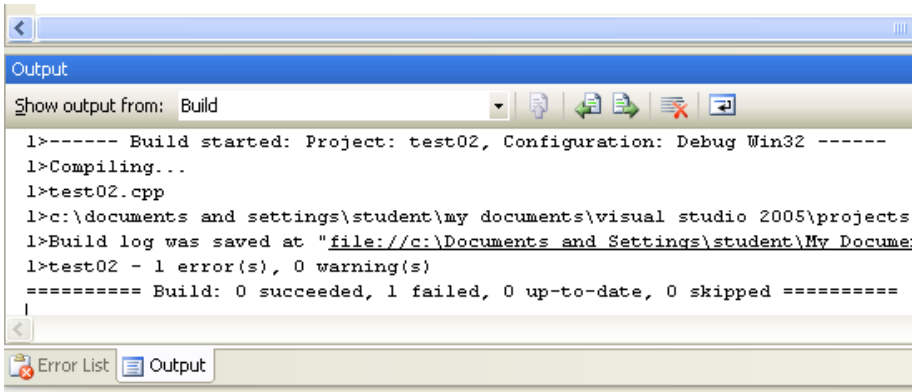

Рис. 5. Окно Output: вывод информации о компиляции проекта и наличии ошибок

В нижней части рабочего окна *Visual C++* имеется несколько вкладок, отражающих процесс выполнения программы. На *Рис. 5* можно видеть окно вывода «*Output*», в котором отражается процесс компиляции и выполнения программы, а также

наличие в ней ошибок. На *Рис. 5*, 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. 6*). 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. 6*) 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*>.

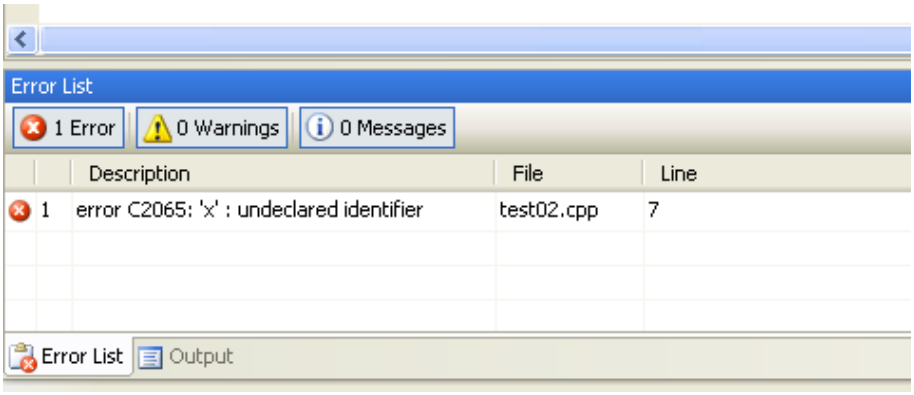

Рис. 6. Окно «Error List»: вывод информации об ошибках

Если ошибок не обнаружено, то проект (полученный исполняемый файл `*.exe`) направляется на выполнение (*Рис. 7*).

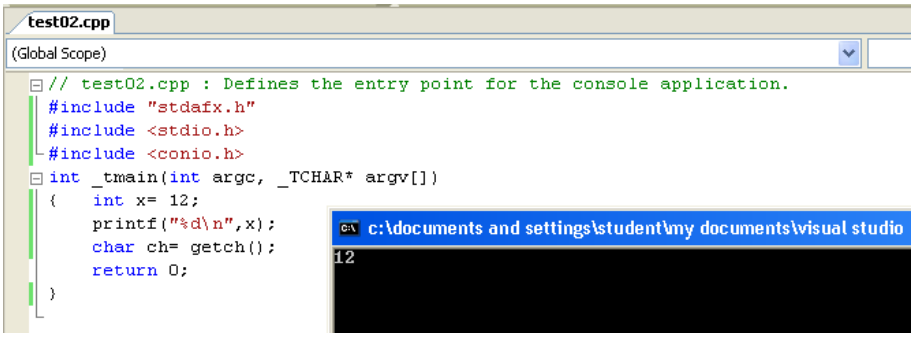

Рис. 7. Выполнение программы в консольном окне

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`);
- логические переменные (`bool`);
- «пустой» тип `void`.

Целочисленные переменные хранятся в памяти в виде двоичного 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 – ячейке памяти) *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 для x86 платформы

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` = `-32768` до `11111111 11111111` = `32767`. 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 6

```
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. 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

Операция	Краткое описание
`/=`	деление с присваиванием
`%=`	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`, приходится использовать в этих целях буквы от `A` до `F`. Например, шестнадцатеричное число `A3F` (записанное как `0xA3F` на языке С) представляется как:

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

Шестнадцатеричная система исчисления традиционно используется в C++ для записи *адресов* переменных – нумерации байт в ОЗУ.

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:

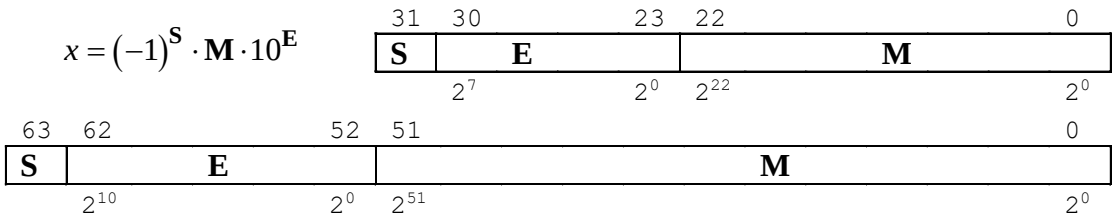

Рис. 8. Форматы числа в формате с плавающей запятой

Для записи двоичных чисел тоже используется такая форма записи. Эта форма записи называется запись числа с плавающей точкой. На рисунке (*Рис. 8*) 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, символу 'B' – код 66, и т.д.

Для примера рассмотрим фрагмент программного кода, приведенный ниже (*Листинг 7*).

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
	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, char* 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 результат – некоторое итоговое 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 целой строки символов – 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 5*). 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 3*) наиболее применимыми из них являются подпрограммы форматного ввода-вывода `printf()` и `scanf()`.

*Таблица 3 Стандартные функции **форматного** ввода-вывода*[1]

Функция	Описание
`printf`	`int printf(const char *format<, argument, ...>);` запись данных в консольное окно в заданном формате. В случае успеха вернет число записанных байт, в случае ошибки – `EOF`
`scanf`	`int scanf(const char *format<, address, ...>);` чтение данных с клавиатуры в заданном формате.
`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.

Функция форматного вывода printf()

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 строка* определяет вид сообщения, которая выводится на экран. Она может содержать специальные управляющие символы для вывода переменных, определяющие ***как*** выводить данные на экран. Далее, идет необязательный *список выводимых переменных* – перечисленные через запятую переменные, значения которых необходимо вывести на экран. Список выводимых переменных указывает ***что*** будет отображаться на экране. Функция возвращает либо число отображенных символов, либо, в случае неправильной работы – отрицательное число.

```
printf("Выводимый текст");
```

Структура форматного модификатора `"формат"` обязательно записывается в двойных кавычках как константная строка и 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 *модификаторами* и которые начинаются со знака «процент» `"%"`.

Нужно понимать, что модификаторы `"%d"`, `"%e"` и др. указывают на то ***как*** выводить переменные, но сами они *не выводятся*. Эти модификаторы *будут заменены* соответствующими по порядку переменными при выводе в окно консольного приложения. Перечень форматных модификаторов содержит *Таблица 4*.

[1] Здесь константа *EOF* – код, возвращаемый как признак конца файла

Таблица 4 Модификаторы форматного ввода\вывода

Модификатор	Описание типа переменной
%c	char – при вводе читается и передается 1 байт; при выводе переменная преобразуется к типу char
	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, использует цифры a, b, c, d, e, f
%X	16-ричное целое без знака, использует цифры A, B, C, D, E, 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
%G	число со знаком в формате E или F
%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 в шестнадцатеричном формате, размер зависит от модели памяти

Пример, приведенный ниже (*Листинг 8*), иллюстрирует задание двух переменных i и x, присвоение им значений и вывод их на экран.

Листинг 8

```
#include "stdafx.h" // подключение библиотек
#include <stdio.h>
#include <conio.h>
//---------------------------------------------------------------------------
int _tmain(int argc, _TCHAR* argv[]) // главная программа
{
	system("chcp 1251"); // подключение русской таблицы ASCII
	//
	int y = 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", y); // вывод на экран переменной y в формате %d
	printf("переменная x = %f\n", x); // вывод на экран переменной x в формате %f
	system("pause"); // ожидание нажатия клавиши
	return 0; // выход из программы
}
```

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

Обратите внимание на то, как выводится на экран переменная y: printf("переменная i = %d\n", y). На экран будет выведено сообщение что переменная i = 5, а не y. В константной строке может быть любой текст, именно он будет выводиться в окно консоли. Но значение вместо модификатора "%d" будет

подставлено из переменной y, так как именно она стоит в списке выводимых переменных. Схема на *Рис. 9* иллюстрирует структуру вызова подпрограммы форматного вывода.

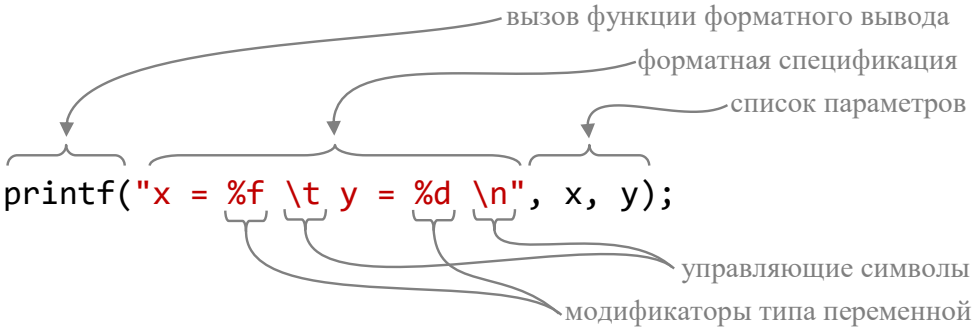

Рис. 9. Структура вызова подпрограммы форматного вывода

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 5*). Строка форматной спецификации содержит также *модификаторы типов переменных* `%f` и `%d` (*Таблица 4*), 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`.

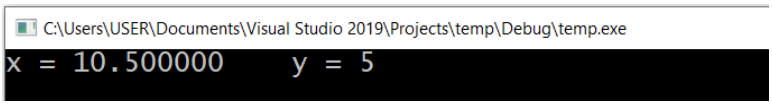

Рис. 10. Результат вывода на экран переменных форматным способом

На *Рис. 9* приводится пример использования нескольких управляющих последовательностей, их еще называют управляющими символами. Полный список управляющих символов содержит *Таблица 5*.

Таблица 5 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 9*).

Листинг 9

```
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"
```

Функция форматного ввода scanf()

Аналогично используется и функция `scanf()` (*Таблица 3*), эта функция предназначена для форматного *ввода* данных с клавиатуры, она также располагается в библиотеки `<stdio.h>` 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 4*). Затем через запятую идет ***список адресов*** вводимых переменных.

Листинг 10

```
int _tmain(int argc, _TCHAR* argv[]) // главная программа
{
	system("chcp 1251"); // подключение русской таблицы ASCII
	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.\tВозраст = %d, вес = %f.\n", name, age, weight);
	system("pause"); // ожидание нажатия клавиши
	return 0; // выход из программы
}
```

```
Введите ваше имя: Terminator
Введите ваш возраст: 18
Введите ваш вес: 95

Здравствуйте, Terminator.	Ваш возраст = 18, ваш вес = 95,000000.
```

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 4* при знакомстве с подпрограммами.

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 11

```
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 11*), 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 11* – `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.

Напомним еще раз, что функции форматного ввода-вывода содержит *Таблица 3*, модификаторы формата ввода\вывода – *Таблица 4*, а управляющие символы - *Таблица 5*.

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

Наравне с форматным способом ввода-вывода данных существует потоковый метод. Он сложнее организован, но пользоваться им значительно проще. Тут не нужно знать формат ввода и вывода – просто указывай что тебе нужно отобразить на экране консоли и все.

Операторы потокового ввода-вывода содержатся в библиотеке `<iostream.h>`. В этом учебном пособии основные функции и объекты этой технологии содержит *Таблица 6*.

Потоковые объекты ввода/вывода `std::cout` и `std::cin` – это стандартные объекты классов `istream` (от *Input Stream* - поток ввода) и `ostream` (от *Output Stream* - поток вывода) соответственно, они используются при помощи операторов сдвига – *извлечения из потока (`>>`)* и *вставки в поток (`<<`)* [8, 14]. Организация системы классов ввода/вывода довольна сложна, но нам не придется её изучать. Будем пользоваться этим инструментом пока без понимания как он устроен. Базовым классом является класс `std::ios` (от *Input/Output Stream* - потоковый ввод/вывод). У класса `std::ios` большое количество производных классов. На данный момент нас интересует лишь некоторые – `std::cin` и `std::cout`.

Префикс `std::` принадлежности к адресному пространству `namespace std` обычно опускается, если перед началом работы программы указано, что в ней будет использованы объекты этого блока `std`. Для этого перед началом программы `main()` необходимо прописать строку `using namespace std`.

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

Таблица 6 Стандартные функции потокового ввода-вывода[2]

Функция	Описание
`cout <<`	оператор сдвига влево `<<` (вставка в поток). Левым операндом (куда вставлять) выступает объект `cout`, а правым – последовательность выводимых данных, в роли которых могут выступать *константы и значения переменных* любых стандартных типов языка C++
`cin >>`	оператор сдвига вправо `>>` (извлечение из потока). Левым операндом (откуда извлекать) выступает объект `cin`, а правым – получаемые данные, в роли которых могут выступать *переменные* любых стандартных типов языка C++
`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. В случае ошибки 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 `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`

Для использования потокового ввода и вывода необходимо подключить библиотеку `#include <iostream>` и задать объект адресного пространства `using namespace std`, в котором создаются и настраиваются потоковые объекты `cout` и `cin`. Объекты `cout` и `cin` содержат ряд подпрограмм (методов) и важнейшими являются операторы извлечения из потока `>>` и вставки в поток `<<`. Пример применения этих операторов показан ниже (*Листинг 12*).

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

Листинг 12

```
#include <iostream.h> // библиотека потокового ввода-вывода
// using namespace std; не подключено адресное пространство std
int _tmain(int argc, char* argv[])
{
	std::cout << "Hello, World"; // оператор вывода в поток cout
	std::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`.

Обратите внимание, что в программе (*Листинг 12*) адресное пространство `namespace std` не подключено, поэтому к операторам `cout <<` и `cin >>` нужно обращаться полностью: `std::cout <<` и `std::cin >>` соответственно. В программе ниже (*Листинг 13*) адресное пространство подключено, и эти операторы можно использовать в краткой форме.

Операторы `cout <<` и `cin >>` позволяют 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 13

```
#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;
	system("pause");
	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 – *Таблица 5*).

В программах довольно часто можно встретить функцию `getch()`. Она ожидает нажатия символа без отражения на экране и возвращает код нажатой клавиши.

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 7*.

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'
	std::cin >> st; // ввод строки с клавиатуры
	cout << st; // вывод строки на экран
	char ch= getch(); // ожидание нажатия клавиши и передача символа в ch
}
```

Альтернативой операторам *извлечения из потока* `>>` и *вставки в поток* `<<` могут выступать функции `gets()` и `puts()` потокового ввода-вывода, см. *Листинг 15*. 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 15

```
#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!");
}
```

Таблица 6 содержит стандартные потоковые функции ввода-вывода.

Нужно добавить, что в библиотеке работы со строками `<string.h>` содержатся свои собственные подпрограммы ввода-вывода – методы классов `string`. Но о них речь пойдет позже.

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, long 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 16*) переменной `y` присваивается константа `4` явно преобразованная `y = (double)4` в вещественный вид `4.0`.

Листинг 16

```
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
```

Неявное преобразование типа введено в язык для удобства программиста. Оно производится неявно, но все равно производится. Компилятор *всегда* преобразует один тип к другому для их сравнения, присваивания или любых других операций. А явно это оформлено в программе или нет – для процессора, реализующего вычисления не важно.

Если вещественной переменной присваивается целое значение, то это целое число предварительно переводится в вещественный формат добавлением нулевой дробной части:

```
double x = 1.99999;
int b = 90;
x = b; // неявное преобразование типа - добавление нулевой дробной части
printf("b = %d x = %f\n", b, x);
```

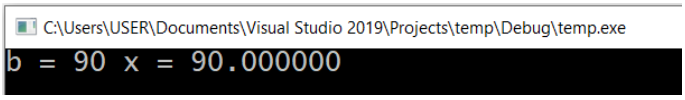

Рис. 12. Пример неявного преобразования типов (целое – в вещественное)

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

```
  double x = 1.99999;
  int a = 67;
  a = x; // неявное преобразование типа - отбрасывание дробной части
  printf("x = %f\t", x);
  printf("a = %d\n", a);
```

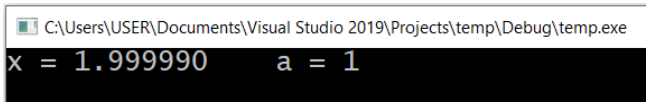

Рис. 13. Пример неявного преобразования типов (вещественное – в целое)

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 17*), а если вещественный – то вещественнозначное.

Листинг 17

```
#include <iostream>
using namespace std;
int main()
{ system("chcp 1251"); // подключение русских символов
  double x; // вещественная переменная x
  x = 13 / 2;
  printf("x = 13/2 = %f - целочисленное деление\n", x);
  x = 13 / 2.0;
  printf("x = 13/2.0 = %f - неявное преобразование типа\n", x);
  x = 13 / (double)2;
  printf("x = 13/(double)2 = %f - явное преобразование типа\n", x);
  system("PAUSE"); // ожидание нажатия клавиши
}
```

В приведенном примере `13/2` деление целочисленное, так как `13` и `2` целые числа. Деление `13/2.0` – вещественнозначное, так как `2.0` – вещественная константа. При делении `13/(double)2` явно указывается тип данных – вещественный. Здесь сперва

число 2 переводится в 2.0, а потом производится деление. Результаты работы программы приведены на *Рис. 14*.

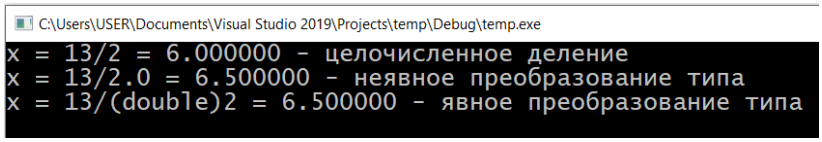

Рис. 14. Пример преобразования типов данных при делении

Полезно знать так же оператор «%», позволяющий вычислить *остаток от деления* нацело (*Листинг 18*). С помощью него удобно определять четность какого-либо числа.

Листинг 18

```
for (int i = 10; i < 16; i++)
  cout << i << "%3= " << i % 3 << "    ";
cout << "\n\n";
```

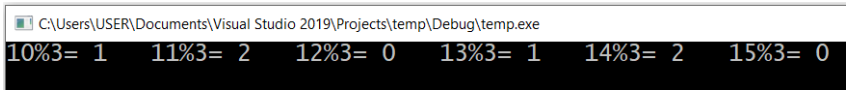

Рис. 15. Пример работы оператора % - остаток от деления нацело

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 - 14*].

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

2.1. Операторы выбора

Оператор if-else

Условный оператор `if-else` работает так (*Листинг 19*): Сперва производится вычисление выражения в круглых скобках – проверка условия. Это выражение должно быть логического (`bool`) типа – оно называется `условие`. Если условие истинно (`true`), то выполняются `операторы` в фигурных скобках, расположенные сразу после условия. А `альтернативные_операторы` (расположенные в фигурных скобках после слова `else`) пропускаются. На этом условный оператор считается выполненным и управление переходит на последующие действия.

Иначе, если условие не выполняется (`false`), то пропускаются основные `операторы`, которые расположены после `if(условие)`, а выполняются `альтернативные_операторы`, расположенные после `else`. После этого оператор считается выполненным.

Важно понимать, что всегда выполняется только один блок операторов – основной или альтернативный и только один раз. После этого управление переходит на последующие действия (*Рис. 16, а*):

Листинг 19

```
if (условие) // проверка условия
{// если условие выполняется (истинно)
  операторы;
}
else
{ // если условие не выполняется (ложно)
  альтернативные_операторы;
}
// последующие действия
```

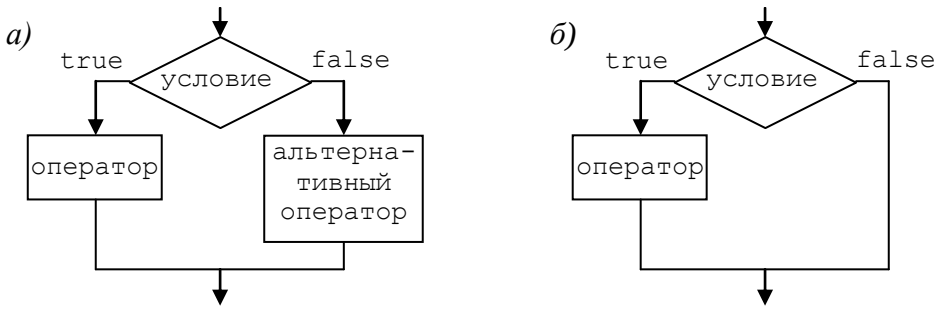

Рис. 16. Блок-схемы условного оператора а) `if-else`; б) `if`

Возможна ситуация, когда альтернативного оператора нет (*Листинг 20*, *Рис. 16, б*). При этом условный оператор `if` вычисляет `условие` и, если условие истинно (не равно `0`) выполняются основные `операторы`. На этом условный оператор считается выполненным и управление переходит на последующие действия.

Листинг 20

```
if (условие) // проверка условия
{// если условие истинно
  операторы;
}
// последующие действия
```

Допускаются вложения условных операторов, при этом `else` относится к ближайшему `if`. Чтобы разобраться в этом рассмотрите пример, в котором выясняется является ли введенная переменная положительной, отрицательной или нулевой:

Листинг 21

```
#include <iostream>
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";
		}
	}
	system("pause");
}
```

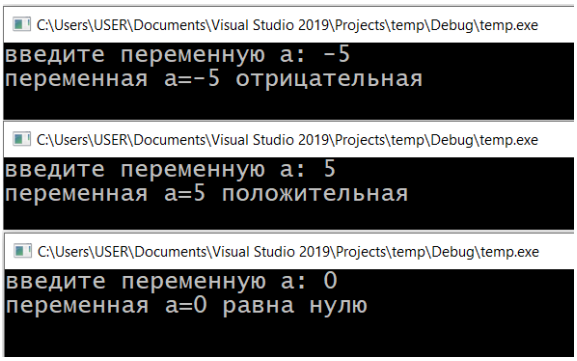

Рис. 17. Результат работы программы

Оператор switch

Оператор выбора `switch` предназначен для того, чтобы делать выбор не из двух альтернатив, как у оператора `if-else`, а из большего (обязательно целого) числа вариантов.

Он содержит в своем описании некоторое наперед заданное число вариантов (`case`) и передает управление на одну из этих меток `case` в зависимости от значения целочисленного `выражения`. Это выражение записывается в круглых скобках

switch(выражение), а после него в фигурных скобках перечисляются варианты case и операторы для каждого из них.

```
switch (выражение) // проверка значения выражения
{
       case N: Оператор для метки N;
       case M: Оператор для метки M;
       case P: Оператор для метки P;
       ...
       case Z: Оператор для метки Z;
       default: Оператор, выполняемый для любого значения выражения;
};
```

Значение выражения всегда должно быть целым числом. Если это число совпадает со значением какой-то из меток case, то выполняются все действия, начинающиеся с данной метки. Важно, что выполняться будут и все последующие метки, если их выполнение не ограничить искусственно.

Существует метка *«по умолчанию»* (default), которая срабатывает в том случае, если ни один из вариантов case не совпадает с величиной вычисленного выражения. Если метка по умолчанию default отсутствует, то в случае несовпадения ничего не происходит.

Рассмотрим пример (*Листинг 22*) и результат его выполнения (*Рис. 18*):

Листинг 22

```
int a = 0;
cout << "введите целую переменную a от 1 до 5: ";
cin >> a; // ввод переменной
switch (a) // проверка значения выражения
{
       case 1:  cout << "переменная равна единице" << "\n";
       case 2:  cout << "переменная равна двойке" << "\n";
       case 3:  cout << "переменная равна тройке" << "\n";
       case 4:  cout << "переменная равна четырем" << "\n";
       case 5:  cout << "переменная равна пяти" << "\n";
       default: cout << "число a не попадает в диапазон [1..5]" << "\n";
};
```

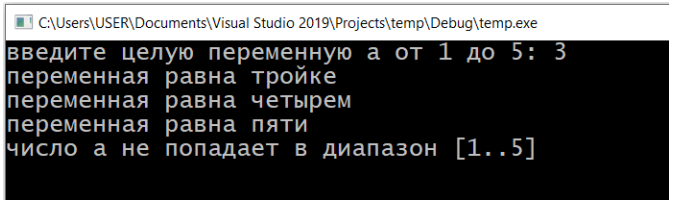

Рис. 18. Результат работы программы

В приведенном выше примере при помощи cin >> было введено значение 3 в переменную a, которая передана в качестве выражения в оператор switch(a). Была выполнена case 3: и все следующие за ней метки case, включая default.

Получилось не совсем то, что хотелось. Как сделать так, чтобы выполнялась только какая-нибудь одна из меток case? Для выхода из блока case служит оператор break. Если после выполнения блока case присутствует оператор break, то следующий блок case выполняться не будет. Рассмотрим альтернативный вариант (*Листинг 23*) и результат его выполнения (*Рис. 19*):

Листинг 23

```
int a = 0;
cout << "введите целую переменную a от 1 до 5: ";
cin >> a; // ввод переменной
switch (a) // проверка значения выражения
```

```
{
       case 1:  cout << "переменная равна единице" << "\n";
       case 2:  cout << "переменная равна двойке" << "\n";
       case 3:  cout << "переменная равна тройке" << "\n";
       case 4:  cout << "переменная равна четырем" << "\n";
       case 5:  cout << "переменная равна пяти" << "\n";
       default: cout << "число a не попадает в диапазон [1..5]" << "\n";
};
```

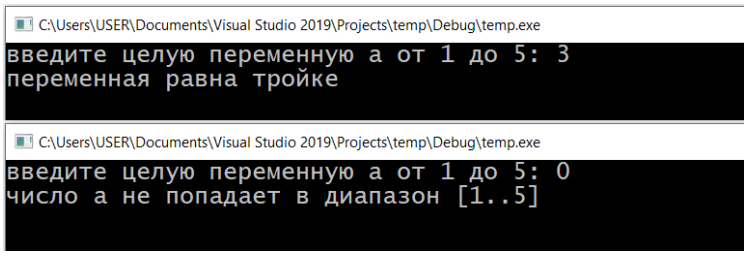

Рис. 19. Результат работы программы для значений `a` *равным* `3` *и* `0`

На листинге ниже (*Листинг 24*) представлен примитивный фрагмент программы, в котором по коду нажатого символа `ch` происходит распознавание нажатой клавиши.

Листинг 24

```
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;
};
```

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 24*).

В языке C++ некоторые управляющие символьные последовательности (*Таблица 5*) заданы специальными константами, например символ `'\n'` – «конец строки» определен в библиотеке `<stdio.h>` константой `endl` (см. *Листинг 24*).

В качестве метки используются только целочисленные (или перечисляемые – *enum*) константы.

2.2. Константы и перечислимый тип данных enum

В языке C++ довольно активно используется модификатор `const`, означающий неизменность объекта в процессе работы программы.

Данный модификатор используется в описании интерфейсов подпрограмм, для фиксации объектов, которые нужно передать в функции и методы так, чтобы не опасаться, что они будут там изменены – за эти проследит компилятор.

Объекты со спецификатором `const` не могут быть изменены, поэтому их значения должны быть заданы при инициализации. Модификатор `constexp` описывает *вычислительные константы*, рассчитываемые во время компиляции. Используется для размещения данных в памяти программ, где они не будут повреждены, а также для увеличения производительности.

Рассмотрим на примере, как эти модификаторы применяются:

Листинг 25

```
const int a = 12; // константа
a = 13; // ошибка: нельзя менять значение константы (слева от присваивания)
```

```
int x = 2 * a * 2; // использовать константы в вычислениях справа можно
const char c; // ошибка: константа должна быть проинициализирована
constexpr double y = 12 * a - a * a + 134; // константное выражение
constexpr double z = 2 * x; // ошибка: x не является константой
const int massiv[] = { 1, 2, 3, 4 }; // все элементы massiv[i] являются константами
massiv[2] = 3; // ошибка: massiv[2] является константой
```

Обратите внимание, что модификатор `const` формально изменяет тип объекта: переменная `a` имеет тип `const int`, а не `int`, для компилятора это разные типы данных. Таким способом модификатор `const` ограничивает варианты использования объекта.

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` *(enumerated)*, 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 – как числа. При отсутствии инициализатора первая константа имеет значение `нуля`, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

Листинг 26

```
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` и `ERR_CONVERT` в примере (*Листинг 26*) присваиваются значения `0`, `1` и `2` соответственно. Однако если программисту удобно нумеровать константы перечисляемого типа иначе, он может это сделать:

```
enum { two = 2, tre, for, ten = 10, elvn, fifty = ten + 40 };
// Константам tre и for присваиваются значения 3 и 4, константе elvn – 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 выполнение (повторение) `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 (аварийного) выхода из цикла служат операторы прерывания – `break`, `return` или `continue`, которые мы обсудим в конце этой главы.

На *Рис. 20* приведены блок-схемы трех альтернативных операторов цикла – *цикл с предусловием* (`while`), *цикл с постусловием* (`do-while`) *и цикл со счетчиком* (`for`). Они полностью взаимозаменяемы. Рассмотрим каждый из них более подробно.

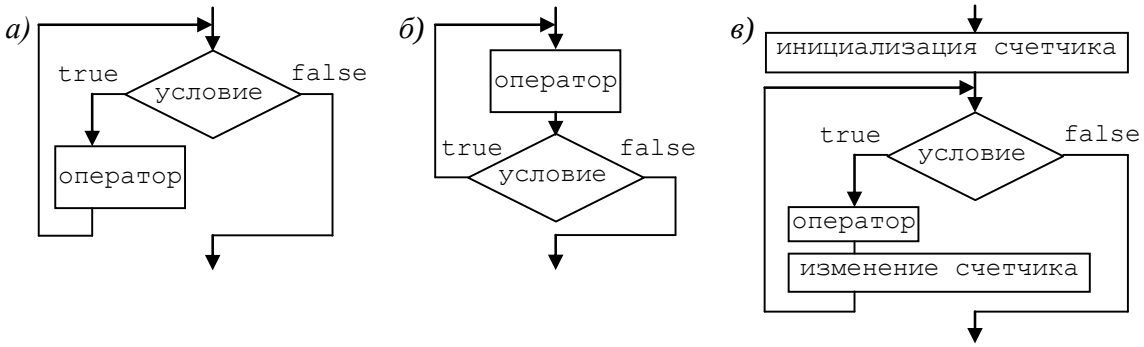

Рис. 20. Операторы цикла а) с предусловием; б) с постусловием; в) со счетчиком

Цикл с предусловием

Данный цикл определяется оператором `while` (*Рис. 20, а*). На каждой итерации сначала проверяется `условие`, а затем выполняется `оператор` или совершается выход из цикла (в зависимости от истинности условия). Если условие истинно (`true`), то совершается итерация, если условие ложно (`false`), то – выход. Если `условие` изначально `ложно`, то оператор не будет выполнен ни разу.

```
while (условие) // пока условие истинно
{ // совершается итерация
   оператор
}
// последующие действия
```

Рассмотрим пример вывода на экран последовательности целых чисел (*Листинг 27*). Прошу обратить внимание на инкремент `i++` (увеличение на единицу) переменной `i` в теле цикла `while`. Очевидно, что именно эта переменная определяет, когда условие цикла истинно, а когда – ложно. Если оператор `i++` убрать из тела цикла, то значение переменной `i` от итерации к итерации меняться не будет и цикл станет бесконечным.

Листинг 27

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

```
C:\Users\USER\Documents\Visual Studio 2019\Projects\temp\Debug\temp.exe
0 1 2 3 4 5 6 7 8 9 10
```

Рис. 21. Результат работы цикла с предусловием

Цикл с постусловием

```
do
{ // совершается итерация
  оператор
}
while (условие); // пока условие истинно
// последующие действия
```

В цикле с постусловием `do-while` (*Рис. 20, б*) условие проверяется после выполнения тела цикла. Таким образом, всегда выполняется хотя бы одна итерация цикла. Условие выхода из цикла то же самое – если условие ложно.

Пример (*Листинг 28*) иллюстрирует работу цикла с постусловием. Результаты приведены на *Рис. 22*. Мо

Листинг 28

```
int i= 0;
do // цикл с постусловием
{
       cout << i << " ";// оператор
       i++;
} // для выхода из цикла
while (i <= 10); // условие
```

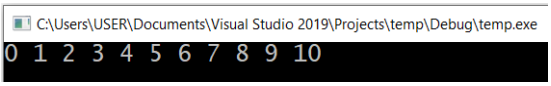

Рис. 22. Результат работы цикла с постусловием

Цикл по счетчику

Цикл по счетчику `for`, как правило, используется в том случае, когда заранее известно число итераций. Для подсчета итераций в нем используется специальная переменная – счетчик (`counter`).

В цикле по счетчику `for` (*Рис. 20, в*) `условие` проверяется до выполнения тела цикла так же как у `while`. Однако здесь дополнительно можно указать `начальные значения` для счетчика и `поститерационные действия`, производимые после выполнения каждой итерации цикла.

Предитерационные действия (инициализация счетчика и других локальных переменных), условие и поститерационные действия записываются в круглых скобках после служебного слова `for`. Эти блоки не являются обязательными – некоторых из них может не быть.

```
for (инициализация_счетчика; условие; поститерационные_действия)
{ // совершается итерация
       оператор;
}
// последующие действия
```

Перед началом выполнения тела цикла, один раз(!), производится `инициализация счетчика` или других локальных переменных цикла. Перед каждой итерацией проверяется `условие`, в зависимости от истинности условия выполняется `оператор` или совершается выход из цикла. После каждой итерации производятся `поститерационные действия`, обычно это инкремент (`++`) или декремент (`--`) счетчика.

Листинг 29

```
for (int i = 0; i <= 10; i++)
{
       cout << i << " ";// оператор

}
```

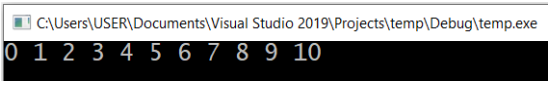

Рис. 23. Результат работы цикла со счетчиком

2.4. Операторы прерывания и безусловного перехода

В языке C++ имеются три оператора немедленного и безусловного прерывания 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.

Оператор прерывания и возврата из подпрограммы `return`

Оператор `return` завершает выполнение текущей подпрограммы или основной программы `main()`.

Рассмотрим приведенный ниже (*Рис. 24*) пример использования оператора `return`. Можно видеть, что цикл со счетчиком `for` должен проделать ровно `10` итераций, в которых вычисляется значение переменной `x`, кратной `15`, и выводится на экран значение переменной `x` и счетчика `count`. Однако, в теле цикла присутствует оператор выбора `if`, сравнивающий величину переменной `x` и число `60` и, в случае равенства, запускающий оператор выхода из подпрограммы `return 13`.

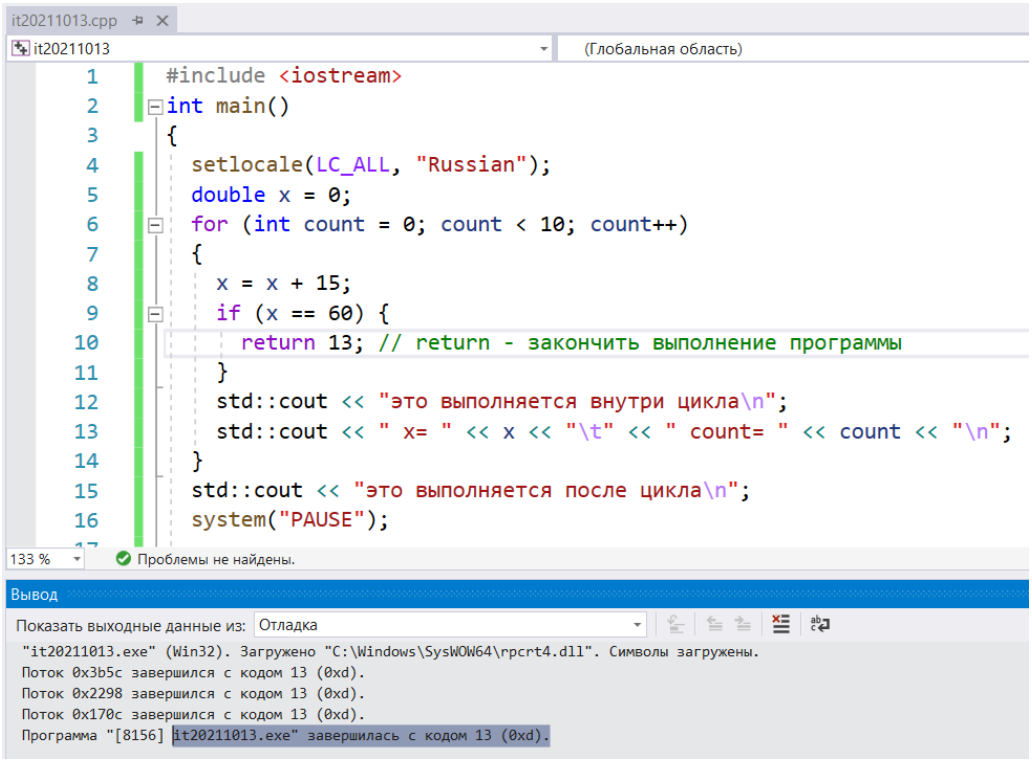

Рис. 24. Пример использования оператора `return`

Переменная `x` кратна `15`, поэтому на четвертой итерации цикла `for` условие `if (x==60)` становится истинным, а значит, вызывается оператор `return`. Оператор `return` тотчас же прерывает выполнение цикла и выходит из подпрограммы. В данном примере – при выходе из программы `main()`, оператор `return` возвращает заданное значение `13`. Поэтому выполнение текущей программы будет прервано на четвертом шаге цикла, консольное окно закроется, и программа возвратит результат своей работы – число `13`.

Если оператор вызывается внутри какой-то подпрограммы, то он немедленно прерывает её работу и переходит в то место головной программы, из которого была вызвана подпрограмма. Причем возвращает в эту точку – точку вызова (или точку возврата из подпрограммы) то значение, которое указано после него в подпрограмме:

```
return возвращаемое_значение;
```

Оператор прерывания цикла `break`

Оператор `break` служит для немедленного и безусловного *выхода из цикла*, но не из подпрограммы. Управление передается следующей команде после оператора цикла. Рассмотрим пример, приведенный ниже (*Листинг 30*).

Листинг 30

```
int main()
{
	double x = 0;
	for (int count = 0; count < 10; count++)
	{
		x = x + 15;
		if (x == 60) {
			break; // прерывает цикл, выходит из цикла
		}
		std::cout << "это выполняется внутри цикла\n";
		std::cout << " x= " << x << "\t" << " count= " << count << "\n";
	}
	std::cout << "это выполняется после цикла\n";
	system("PAUSE");
}
```

Оператор прерывания цикла `break` расположен здесь в той же позиции, которую занимал в предыдущем примере оператор `return`, но результат будет другой. При выполнении оператора `break` выполнение цикла будет немедленно и безусловно остановлено и управление передастся следующему за циклом оператору. Поэтому операторы, выводящие значения переменных `x` и `count` будут выполнены только для трех начальных итераций – потом цикл прерывается (*Рис. 25*). Но программа в целом на этом не обрывается, а выполняет все операторы после цикла, и ждет нажатия клавиши для завершения.

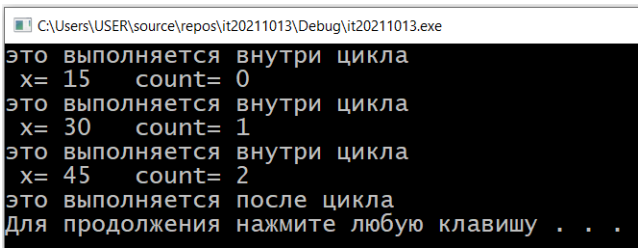

Рис. 25. Пример использования оператора `break`*, результат работы программы*

Оператор прерывания итерации `continue`

Оператор `continue` служит для немедленного и безусловного прерывания ***только одной*** (текущей) итерации цикла, но при этом выполнение цикла не прекращается, а управление переходит к началу следующей итерации цикла. Рассмотрим пример, приведенный ниже (*Листинг 31*).

Листинг 31

```
	double x = 0;
	for (int count = 0; count < 10; count++)
	{	x = x + 15;
		if (x == 60) {
			continue; // прерывает итерацию, переходит к следующей
		}
		std::cout << "это выполняется внутри цикла\n";
		std::cout << " x= " << x << "\t" << " count= " << count << "\n";
	}
	std::cout << "это выполняется после цикла\n";
```

Оператор прерывания итерации `continue` расположен здесь в той же позиции, которую занимал в первом примере оператор `return`. Условие `if (x == 60)` как и в предыдущих случаях сработает на четвертой итерации цикла (`count = 3`). Оператор `continue` прерывает выполнение только этой – четвертой итерации и передает управление следующей – пятой. Поэтому операторы, выводящие значения переменных `x` и `count` будут пропущены один раз (*Рис. 26, метка A*) потом цикл продолжается.

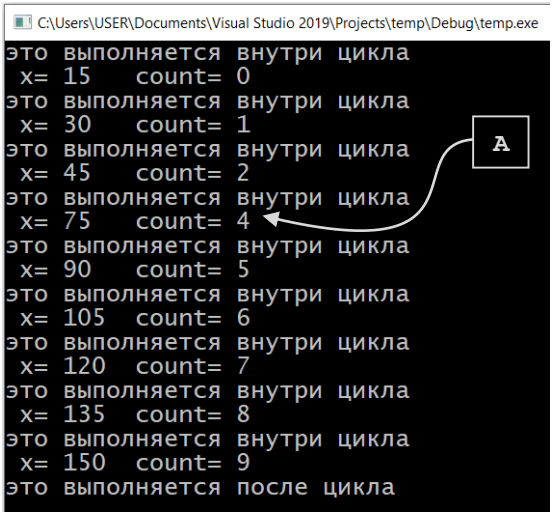

Рис. 26. Пример использования оператора `continue`

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 32

```
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";
```

2.5. Использование переменных логического типа (bool)

При формировании условия выхода из цикла (*Рис. 20*) или условия выбора (*Рис. 16*) довольно часто приходится организовывать весьма сложные логически конструкции на базе логических операторов (*см. Таблица 7*). Результатом действия какой-либо логической операции может являться одно из двух логических значений (`bool`): `false` (ложь) или `true` (истина).

Таблица 7 Логические операции языка C++

Операция	Краткое описание	Пример
Операции сравнения		
`<`	меньше	`(7<8) == true;  (5<5) == false;`
`<=`	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 33

```
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 33*), 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

целое 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 (см. *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.

2.6. 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 34*). 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 34

```
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[])
{      system("chcp 1251");
       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";
}
```

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 не 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 34*) 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`.

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 и ссылки

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

Указатель - это *переменная, которая содержит адрес* некоторого объекта (переменной, ячейки памяти, функции) [*1 - 13*]. Говорят, что указатель *ссылается* на переменную, функцию и т.д. Но как он на нее ссылается? – *По адресу*. Он хранит в себе адрес какой-то переменной и тем самым ссылается на нее.

А что такое адрес в памяти компьютера? *Адрес* – это целое число, являющееся, по сути, номером байта в ОЗУ. Для программиста вся память компьютера представлена в виде последовательности пронумерованных байтов. Номер того байта, с которого начинается любой объект размещенный в памяти, называется адресом этого объекта.

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. В них сосредоточена вся мощь и красота языка C++.

Указатель описывается при помощи символа `"*"` (*звездочка*):

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

Специальных операций для работы с указателями всего две:

- унарная операция `"&" (амперсент)`, означающая "*взять адрес переменной*" или иначе – ***адресация***;
- унарная операция `"*" (звездочка)`, означающая "*взять значение переменной, расположенное по указанному адресу*" – ***разадресация***.

Листинг 35

```
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
```

Рис. 27 иллюстрирует расположение в памяти компьютера (ОЗУ) переменных и указателей из программы (*Листинг 35*). Под каждой ячейкой подписан её адрес (32-битное число в шестнадцатеричном формате), это значит, что все переменные имеют свои адреса. А вот имена имеют не все, но о безымянных переменных – позже.

Переменная `x` содержит в себе число `101` и имеет адрес `002CFBA8`. Переменная-указатель `y` расположена по адресу `002CFBCC`, а внутри себя содержит значение `002CFBA8` – это адрес переменной `x`. Таким образом `y` указывает на переменную `x` (схематически помечено стрелкой).

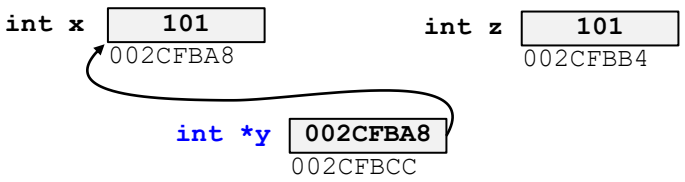

Рис. 27. Расположение в памяти переменных и указателей

Переменная `z`, лежащая по адресу `002CFBB4`, содержит в себе число `101`. Но как это произошло, ведь ей мы напрямую ничего не присваивали (см. *Листинг 35*)?

Присваивание переменной z произошло через указатель. Эта команда: z=*y, понимается так: «*переменной z присвоить содержимое, лежащее по адресу y*». А что лежит по адресу, хранящемуся в указателе y? – Там лежит значение переменной x, т.е. число 101.

3.1. Типизированные и нетипизированные указатели

Указатели бывают *типизированные* и *нетипизированные* (void*). 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.

Рассмотрим пример (*Листинг 36*), поясняющий разницу между указателями, имеющими тип и указателем void*. В листинге задан указатель temp, имеющий тип double*. Затем ему присваивается адрес переменной d: temp = &d. Присваивание происходит корректно, так как типы левого и правого операндов совпадают: &d также имеет тип double*.

Листинг 36

```
double d = 123.456;
double* temp; // указатель на тип double имеет тип double*
temp = &d; // типы совпадают: &d имеет тип double*
int i = 78;
temp = &i; // ошибка: типы не совпадают: &i имеет тип int*
void* vptr; // нетипизированный указатель
vptr = temp; // указателю void* можно присвоить любой адрес
vptr = &i;   // указателю void* можно присвоить любой адрес
vptr = &d;   // указателю void* можно присвоить любой адрес
temp = vptr; // ошибка: типы не совпадают
temp = (double*)vptr; // принудительная типизация адреса
```

В пятой строке листинга делается попытка присвоить указателю temp адрес переменной int i. Очевидно, типы не совпадают, так как &i имеет тип int*. Затем задается нетипизированный указатель void* vptr. Ему можно присваивать любой адрес – проверка на совпадение типов компилятором не производится. Программист отвечает за то, насколько корректно используется void*.

А вот адрес, хранящийся в указателе vptr, присвоить типизированным указателям нельзя. В предпоследней строке листинга опять ошибка: левый операнд присваивания double* не совпадает по типу с правым операндом void*. При присваивании типизированных указателей компилятор проверяет совпадение типов. Как же использовать void*?

Нужно типизировать его вручную, как показано в последней строке программного кода (*Листинг 36*). Сначала адрес, лежащий в указателе vptr из типа void* преобразуется в тип double*, а затем осуществляется присваивание.

Операции с указателями

Указатели можно использовать как операнды в арифметических операциях. Над указателями можно выполнять унарные операции: инкремент (++) и декремент (--). При этом 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 37*).

Листинг 37

```
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);
```

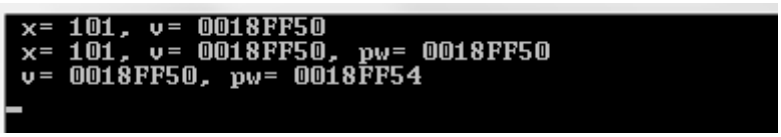

Рис. 28. Результаты выполнения программы

Из рисунка (*Рис. 28*), иллюстрирующего работу программы (*Листинг 37*), можно видеть, что адрес указателя `pw` после операции `pw++` изменился на `4`, с адреса `0018FF50` на адрес `0018FF54`. Значение указателя `pw` увеличилось 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`, т.е. на 12 байт. Это справедливо для любых объектов (`int*`, `char*`, `float*` и др.); транслятор будет масштабировать приращение адреса в соответствии с *типом*, указанным в определении *типизированного указателя*.

Это называется ***косвенный доступ*** к ячейкам памяти «`адрес + смещение`». Здесь `pw` – адрес, а «`+3`» - смещение относительно этого адреса на 3 ячейки заданного типа.

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 38

```
int* ptr1;
int* ptr2 = NULL;
int 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` или `nullptr`. Любой адрес можно проверить на равенство (`==`) или неравенство (`!=`) со специальным значением `NULL`. Это позволяет определить – указывает на какую-то конкретную ячейку данный указатель или нет.

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 в памяти 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` (создание динамической переменной) и `delete` (её удаление).

Листинг 39
```
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 ячеек* заданного размера, состоящий из элементов (ячеек) определенного типа:

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

В этом примере память выделяется под массив из 4 элементов типа `int` и возвращает указателю `pia` адрес первого элемента массива. Эти 4 ячейки памяти типа `int` расположены подряд, без разрыва, одна за другой. Работа с массивами будет рассмотрена детально 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 40*, *Рис. 29*) приводится пример работы с указателями. Указатель `a_ptr` хранит адрес `0018FF50` статической ячейки с именем `a` (типа `double`). А указатель `x_ptr` указывает на *безымянную* динамическую ячейку памяти `0018FF5A` (она выделена цветом на *Рис. 29*).

Листинг 40
```
double a = 10.1;
double* a_ptr = &a;
double* x_ptr = new double(13.5);
delete x_ptr; // освобождение объекта
```

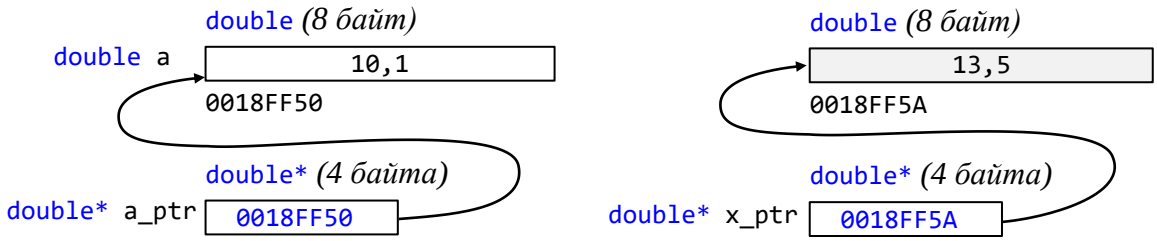

Рис. 29. Динамическое выделение памяти

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 41*, *Рис. 30*):

Листинг 41

```
#include "stdafx.h"
#include <conio.h>
#include <iostream>
//------------------------------------------------------------------------------
int _tmain(int argc, _TCHAR* argv[])
{
	system("chcp 1251"); // 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 %p в ней лежит число %f\n",
			temp, *temp);
		i++;
	}
	delete temp; // освобождение только последнего объекта
	system("pause");
}
```

В приведенном примере заводится указатель `temp`, которому в цикле 101 раз присваиваются адреса новой выделяемой операционной системой динамической ячейки памяти. Из рисунка (*Рис. 30*) видно, что адрес выделяемой ячейки каждый раз новый (отличающийся от предыдущего на 8 байт – размер ячейки типа `double`).

На каждом 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 41*) память, лежащая по указателю `temp`, очищается командой `delete temp`, но это делается один раз и удаляется только одна последняя динамическая ячейка по адресу `013338B0`, а предыдущие 100 ячеек с адресами `0132DD00 – 01333568` утеряны и не очищены.

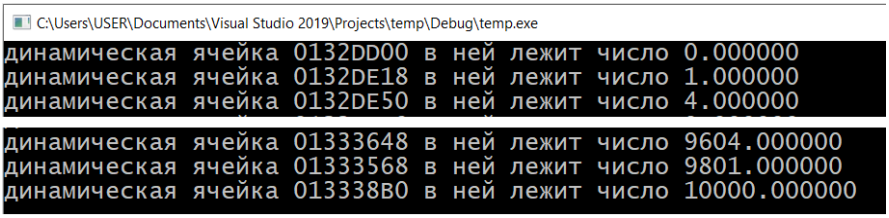

Рис. 30. Результаты выполнения программы

Приведем ниже (*Листинг 42*) 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 42

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

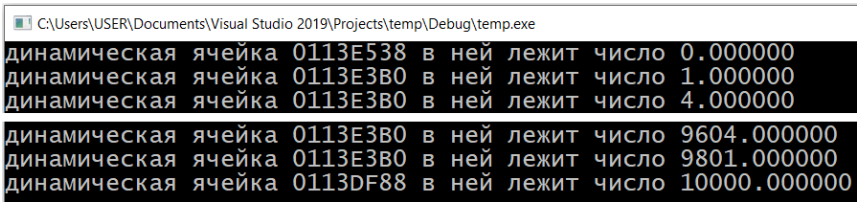

Рис. 31. Результаты выполнения программы

Запустите на своём компьютере программы (*Листинг 41*) и (*Листинг 42*). Проследите адреса выделяемых ячеек. Обратите внимание, что в первом листинге (*Рис. 30*) адреса всех ячеек – уникальные, а во втором (*Рис. 31*) – адреса иногда совпадают. Это объясняется тем, что в первом случае, при выделении очередной ячейки, все выведенные на экран адреса заняты и не освобождены. ОС приходится выбирать новую ячейку из новых свободных. А во втором случае – адреса могут повторяться, так как они были заняты под `temp`, использованы для хранения числа и вывода на экран и затем – освобождены на этой же итерации цикла. На следующем шаге цикла ОС может выделить под указатель `temp` снова ту же ячейку (уже использованную ранее). Так экономится память.

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* (как в программе *Листинг 36*) в указатель объявленного типа:

Листинг 43
```
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 43*), при выделении памяти, происходит последовательно 5 операций:
- функция calloc() запрашивает у ОС 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.

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 44*), он иллюстрирует, как можно проверить, успешно ли выделилась память.

Листинг 44
```
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
printf("%p", str);
free(str); // освобождение памяти
```

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 45
```
char* str;
str = (char*)calloc(10, sizeof(char));
strcpy(str, "1234567890"); // копирование символов "1234567890" в созданный массив
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 45*), 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 с модификатором `"%p"`, будет выведен на экран адрес первого байта выделенного блока. При выводе с модификатором `"%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 разделе *Гл. 7*.

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 непростая задача. Для задач обучающего уровня достаточно использования подпрограммы выдачи *псевдослучайного 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 самому 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)]`. Следовательно, число `X%(b-a)+a` расположено в искомом диапазоне `[a, b]`.

В примере ниже (*Листинг 46*) показывается, как сгенерировать 50 случайных чисел из диапазона от 100 до 200.

Листинг 46

```
#include <iostream>
#include <time.h>
using namespace std;
int _tmain(int argc, char* 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 << " ";
	}
	system("pause");
}
```

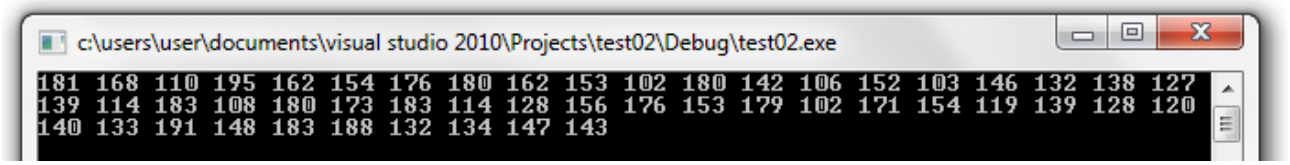

Рис. 32. Результаты выполнения программы

3.5. Ссылки

Понятие ссылки является одним из главных инструментов, благодаря которым язык C++ обладает всей широтой функционала, недоступной другим языкам программирования.

Ссылка в языке C++ это особый вид данных, реализуемый путем хранения адреса объекта, но семантически эквивалентный самому объекту на который ссылается.

Иными словами, ссылка – это *новое имя для уже заданной ранее переменной.* Ссылка – это нечто среднее между именем переменной и её адресом: операции, производимые над ссылкой на самом деле производятся над исходной переменной, на которую установлена данная ссылка. Синтаксически ссылка задается при помощи символа `"&"` (по какой-то причине этот же знак используется для операции взятия адреса, но путать их не нужно – это разные понятия). Рассмотрим пример:

Листинг 47

```
unsigned int x; // переменная

unsigned int* y = &x; // указатель, указывающий на переменную x
unsigned int& z = x; // ссылочная переменная, ссылающаяся на x
                     x++; // увеличить значение x на 1
                     (*y)++; // увеличить x на 1 через указатель
                     z++; // увеличить x на 1 по ссылке
```

В примере (*Листинг 47*) команда `y = &x` описывает взятие адреса переменной `x` и присваивание его в переменную-указатель *y*, метка (2): таким же значком обозначается факт создания ссылочной переменной `z` и инициализации её именем `x`. При использовании ссылки на переменную говорят «`z` *ссылается на переменную* `x`» или «*обращаемся к переменной* `x` *по ссылке* `z`».

Строго говоря, ссылка не является переменной, поскольку изменить значение самой ссылки нельзя – если ссылка стоит в выражении слева от операции, то значит операция будет произведена не над ссылкой, а над переменной на которую она ссылается.

Поэтому переменную ссылочного типа нельзя описать без инициализации, т.е. без задания начального значения.

Этот тонкий момент связан с тем, что в языке C++ *инициализация* и *присваивание* – это разные операции. Если указатель, в отличии от ссылки, это обычная еще одна переменная, просто в ней хранится адрес, то ссылка – это как бы новое имя для переменной, и сама по себе она не существует. Если ссылка с самого начала не связана с конкретной переменной, то потом уже её нельзя будет использовать, ведь все действия, которые мы попытаемся сделать со ссылкой, фактически должны будут производиться над исходной переменной, а она не определена.

Описав в примере (*Листинг 47*) ссылку `z` на переменную `x`, мы фактически завели *синоним* переменной `x`. Такое использование ссылок выглядит бессмысленным и может показаться излишним (оно действительно встречается редко), но при передаче данных в подпрограмму ссылки используются очень активно.

3.6. Константные указатели и ссылки

При инициализации указателя задействуются два объекта: сам указатель и тот объект, на который он указывает. Поэтому модификатор `const` может стоять в двух позициях:

`const тип *имя` – обозначая то, что константой является объект, лежащий по адресу, хранимому в указателе имя (при этом адрес можно изменить);

`тип* const имя` – фиксируя тот факт, что константой является собственно адрес какого-то объекта (объект при этом может быть изменен).

Листинг 48

```
int n, m = 10;
const int* p = &m; // p имеет тип const int* - указатель на константу
p = &n; // допустимо изменение указателя (адреса)
m = 13; // допустимо изменение переменной на которую указывает const int*
*p = 12; // ошибка: попытка изменения содержимого указателя на константу
```

Префиксное объявление модификатора `const` *делает константным объект на который он указывает, а не сам указатель*. На примере выше (*Листинг 48*) можно видеть этот вариант. Не вызывает ошибки у компилятора тот факт, что указатель на константу `p` проинициирован адресом переменной `m` (а не константы).

Описание переменной `const int *p = &m` можно интерпретировать так: «*переменная* `p` *имеет тип* `const int*` *– она является указателем на константу*» или так: «*содержимое* `*p` *указателя* `p` *имеет тип* `const int` *– оно является константой*».

Понимая обозначение таким образом, можно видеть, почему не вызовет ошибки присваивание указателю `p` другого адреса (сам указатель не является константой). Корректным будет также явное изменение переменной `m` через её имя, (ведь никаких ограничений на переменную `m` не наложено). Но попытка косвенного (через указатель) изменения данных, хранимых в ячейке, на которую указывает `p`, вызовет ошибку, так как это – указатель на константу. Константное выражение недопустимо для изменения левосторонним значением.

Следующий пример (*Листинг 49*) иллюстрирует использование константного указателя – константой является сам адрес, хранимый в переменной-указателе, а не содержимое, которое по этому адресу хранится – оно может быть изменено.

Листинг 49

```
int n, m = 10;
int *const p = &m; // константный указатель
p = &n; // ошибка: попытка изменения указателя (константного адреса)
*p = 12; // допустимо изменение значения на которое указывает int*const
```

Приведенное здесь (*Листинг 49*) описание переменной `int* const p = &m` можно интерпретировать так: «*переменная* `p` *имеет тип* `int* const` *– она является константным указателем*» или так: «*константа* `const p` *имеет тип* `int*` *– она является указателем*».

Теперь понятно, что ошибкой будет присваивание указателю *p* другого адреса (поскольку сам указатель является константой). Константное выражение недопустимо для изменения левосторонним значением.

Корректным будет явное (по имени) или косвенное (через указатель) изменение переменной `m` или содержимого `*p`, лежащего по этому *константному адресу* `p`.

Понятно, что возможен вариант *константного указателя на константу*, при котором компилятор следит за неизменностью и адреса, и его содержимого.

```
int m = 10;
const int *const X = &m; // константный указатель на константу
```

Логично, что объект, являющийся константой при доступе через один указатель, может изменяться при доступе иным способом. Это свойство широко применяется при описании аргументов подпрограмм. Объявив параметр функции как *указатель на константу*, можно быть уверенным, что функция не изменят объект, на который указывает параметр. Данная тема будет детально разобрана в следующей главе.

Константные ссылки

Аналогично указателям, ссылочные переменные также могут ссылаться на константы, а могут быть константами сами, а ссылаться на изменяемые переменные:

`const тип& имя` – константой является объект на который делается ссылка;

`тип& const имя` – константой является ссылка, сам объект при этом может изменяться.

Листинг 50

```
double a = 12.12;
const double& d1 = a;
double& const d2 = a;
d1 = 13; // ошибка: значение по ссылке q - константа, оно не меняется
d2 = 13; // ок: константная ссылка на изменяемую переменную
```

Во втором случае не понятно, в чем смысл создания именно константной ссылки, ведь ссылка (в отличие от указателя) и так не имеет смысла сама по себе, её саму (а не тот объект, на который она ссылается) и так не изменить. В случае описания переменной `d2` модификатор `const` не несет никаких дополнительных ограничений и не накладывает никаких дополнительных проверок.

Листинг 51

```
double x;
const double& q = x; // ссылка на константу
double y = q + 5; // использование константы в правой части выражения
x = 56; // double x - не константа
q = 12; // ошибка: значение по ссылке q - константа, оно не меняется
```

Рассмотрим нюансы (*Листинг 51*). Здесь создается ссылка на константу, её можно использовать в правой части выражения, но присвоить ей новое значение нельзя, ведь она ссылается на константу. Для присваивания выражение должно быть допустимым для изменения левосторонним значением, а это не так. При этом, сама переменная `x` может быть изменена.

Листинг 52

```
const double a = 5;
double& r = a; // ошибка: попытка сослаться на константу изменяемой ссылкой
const double& r = a; // константная ссылка инициирована константой
double const& r = a; // константная ссылка инициирована константой
r = 21; // ошибка: попытка изменить константу
```

Пример (*Листинг 52*) иллюстрирует тот факт, что изменить константу при помощи *неконстантной ссылки* на нее не удастся: попытка сослаться на константу изменяемой ссылкой будет остановлена компилятором. Константой можно инициировать только константную ссылку. Ну и, следовательно, изменить константную переменную, на которую ссылается константная ссылка невозможно по двум очевидным причинам. Назовите их.

Обычные ссылки в англоязычной литературе по программированию принято называть *modifiable lvalues*, а константные ссылки *non-modifiable lvalues* (изменяемые и неизменяемые левосторонние выражения).

В тех частях программы, где программист уверен в том, что значение переменной не должно быть изменено, он должен использовать константные модификаторы – это дает команду компилятору контролировать процесс изменения данных, что в свою очередь уменьшает число ошибок в программе.

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* (запускать) подпрограмму в разных частях головной программы ее необходимо *описать* и *задать*. [*1 - 14*]

Описать функцию (описать заголовок, *прототип* функции) – означает определить ее `имя`, `тип_возвращаемого_значения` и `список_формальных_параметров`, записываемых через запятую как на рисунке (*Рис. 33*).

Листинг 53

```
// прототип (описание) функции, её заголовок
тип_возвращаемого_значения имя(список_формальных_параметров);
//---------------------------------------------------------------------------
// головная программа, вызывающая подпрограмму имя()
int _tmain(int argc, char* argv[])
{
	...
	x = имя(список_фактических_параметров); // вызов функции
	...
}
```

Задать подпрограмму – значит, помимо описания заголовка, задать (определить, прописать) и все её команды, составляющие `тело подпрограммы` (*Рис. 33*).

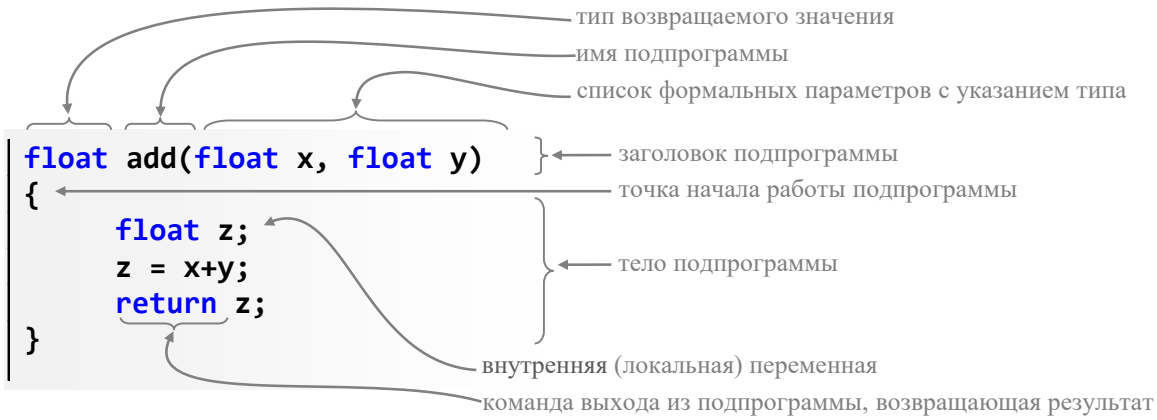

Рис. 33. Структура задания функции (подпрограммы) в языке C++

Если подпрограмма не должна возвращать никакого значения, то `тип_возвращаемого_значения` нужно задать как `void`, такая подпрограмма называется *процедурой*. Если подпрограмма возвращает какое-то значение, то в теле функции должен присутствовать оператор `return`, возвращающий `значение` соответствующего типа. Подпрограммы, возвращающие значение, называются *функциями*.

Листинг 54

```
//---------------------------------------------------------------------------
// задание (реализация) функции
тип_возвращаемого_значения имя(список_формальных_параметров)
{
	// тело функции
	...
	return возвращаемое_значение;
}
```

`Тип_возвращаемого_значения` определяет, какого типа должен быть *результат* вычислений, возвращаемый в программу, вызвавшую нашу функцию. Для возврата из функции используется оператор `return` и стоящее после него выражение (оно должно быть требуемого типа). Первый встретившийся в любом месте функции оператор `return` немедленно прекращает выполнение подпрограммы, возвращая в вышестоящую подпрограмму `значение` расположенного после него выражения (*Рис. 34*).

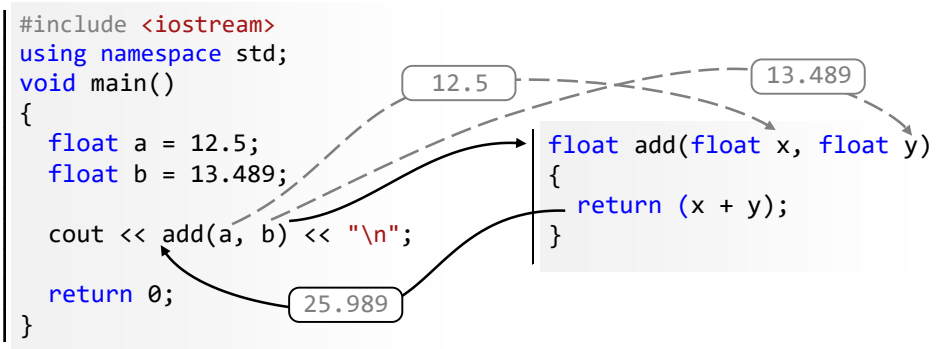

Рис. 34. Логика передачи управления в подпрограмму и возврата из неё.

После того, как функция описана и задана, она может быть вызвана (запущена) из тела главной функции `main()` или других функций программы. Вызывающая программа по отношению к вызываемой подпрограмме, называется *головной*. Та точка головной 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 (*Рис. 34*) точка возврата находится в строке после строки символов `cout <<`.

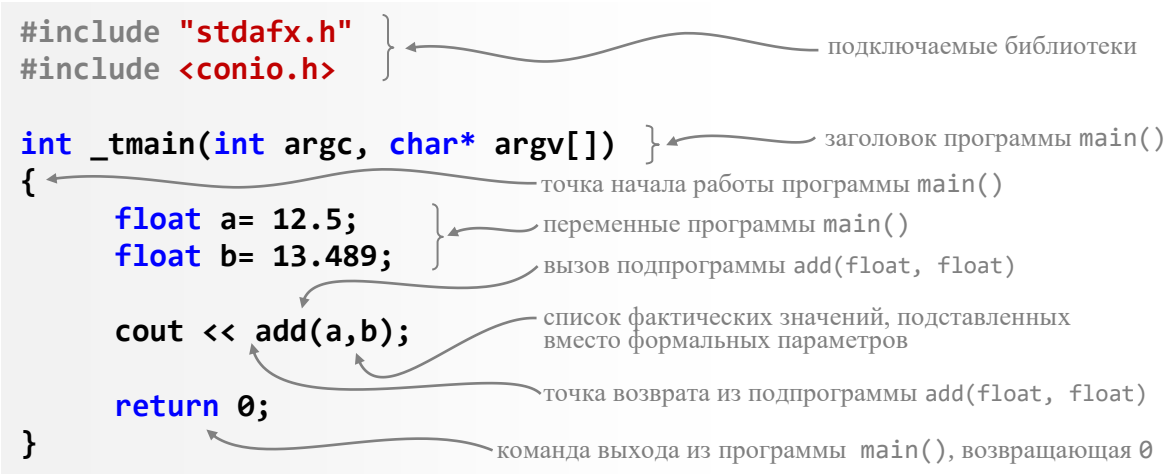

Рис. 35. Структура вызова (использования) функции в главной программе

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 даже можно не указывать при описании подпрограммы, как в примере ниже.

Рассмотрим пример (*Листинг 55*) в котором описывается, задается и вызывается подпрограмма `int min(int, int)` для вычисления минимума из двух чисел. Во второй

строке листинга описывается прототип (заголовок) функции. Реализация подпрограммы размещена в 18-23 строках, здесь задается тело функции. В теле программы `main()` эта функция трижды вызывается для различных фактических параметров: в строке 11 вызывается подпрограмма `min()` для сравнения чисел `12` и `54`, в строке 12 она вызывается с параметрами `12` и `5`, и в строке 13 функция `min()` в качестве фактических значений принимает значения `54` и `-7`.

Листинг 55

```
#include <iostream>
  //-------------------------------------------------------------------------
  int min(int, int); // прототип подпрограммы
  //-------------------------------------------------------------------------
  void main()
  {
    system("chcp 1251");
    system("cls");
    int X, Y;
    X = 12;
    Y = 54;
    std::cout << "X= " << X << " Y= " << Y << "\n";
    std::cout << "min(X,Y)= " << min(X, Y) << "\n";
    std::cout << "min(X,5)= " << min(X, 5) << "\n";
    int Z = min(Y, -7);
    std::cout << "min(Y,-7)= " << Z << "\n";
    std::cout << "\n\n";
    system("PAUSE");
  }
  //-------------------------------------------------------------------------
  int min(int A, int B) // реализация подпрограммы
  {
    if (A <= B)
      return A;
    return B;
  }
  //-------------------------------------------------------------------------
```

Обратите внимание на то, в каком месте программы `main()` вызывается функция `min()`. В первых двух случаях возвращаемое подпрограммой значение передается оператору `std::cout <<` в качестве аргумента, а в третьем варианте результат вычислений функции `min()` присваивается переменной `Z`. Разумеется, в этом случае тип переменной `Z` и тип возвращаемого функцией `int min(int, int)` значения должны совпадать.

Результат выполнения программы (*Листинг 55*) приведен ниже (*Рис. 36*).

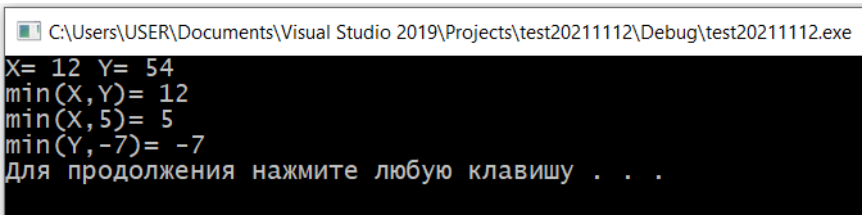

Рис. 36. Результат вызова одной функции с различными параметрами

Заголовочные файлы подключаемых библиотек – файлы с расширением `*.h`: `"stdafx.h"`, `<iostream.h>`, `<conio.h>` и др. содержат прототипы используемых нами подпрограмм `system()`, `std::cout <<`, `std::cin >>`, `srand()`, `time()`, `rand()`, `strcpy()`, `printf()`, `scanf()`, `calloc()`, `malloc()` и др. Вызвать описание всех доступных подпрограмм можно по комбинации клавиш *F12*, *<ALT>-<F12>* и *<CTRL>-<F12>* или вызвав контекстную помощь. Реализация этих подпрограмм размещена в файлах `*.lib` или `*.dll` (*библиотеках*) в откомпилированном виде.

Рассмотрим программу, печатающую степени числа 2:

Листинг 56

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

Первая строка листинга, прототип функции указывает на то, что `float pow(float, int)` – это 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`, и запустит на выполнение функцию `pow(2.0, i)`.

Функция `pow()` может быть определена рекурсивно:

Листинг 57

```
float pow(float x, int n) // задание функции pow()
{
	if (n < 0) // если отрицательный показатель для pow()
		return 1/pow(x, -n);
	switch (n)
	{
	case 0: return 1; // выход из рекурсии
	case 1: return x;
	default: return x * pow(x, n - 1); // рекурсия
	}
}
```

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. 37*). 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 *локальные* (видимые и используемые только внутри подпрограммы) переменные `p`, `q` и `r`.

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. После этого возобновляется выполнение функции `main()` – переменной `z` присваивается возвращаемое значение.

```
// вызываемая функция
int F(int p, int q)
{
  int r = p + q;
  return r;
}
// вызывающая функция
int _tmain(int argc, char* argv[])
{
  ...
    int x = 15, y = 4;
    int z = F(x, y);
  ...
}
```

область видимости переменных функции `F()`

точка возврата из функции `F()`

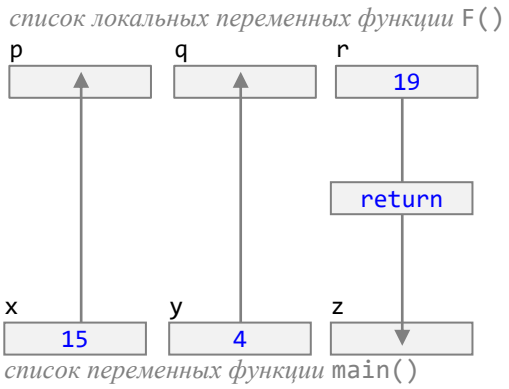

Рис. 37. Передача параметров по значению

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 58*). 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 оператором `return` 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(a, b)` и вновь значения `a` и `b` выводятся на экран. Результаты работы программы (*Листинг 58*) приведены ниже на рисунке (*Рис. 38*).

Листинг 58

```
#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, char* argv[])
{
	system("chcp 1251"); // 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);
        system("pause"); // 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. 38. 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 `0032FB80` и `0032FB74` глобальных переменных `a` и `b`.

Затем производится вывод *локальных* переменных `a` и `b` из подпрограммы `sum2ab()`. Можно видеть, что их адреса `0032FA9C` и `0032FAA0` 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*!

Можно сделать вывод о том, что изменение локальных переменных внутри подпрограммы не влияет на глобальные переменные в головной программе. Потому что передаваемые «по значению» переменные в подпрограмме – это копии глобальных переменных. Изменение «копий» не изменяет «оригиналы».

Передача параметров по ссылке

Передачу параметров «*по ссылке*» удобно рассмотреть на примере функции `S()` меняющей местами свои аргументы `p` и `q` (*Рис. 39*). 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`.

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

В примере (*Рис. 39*) переменные `p` и `q` в теле функции `S()` не существуют как самостоятельные отдельные ячейки памяти, а служат лишь для обозначения ячеек `x` и `y`. В момент вызова подпрограммы, заданные в списке фактических параметров ссылки `&p` и `&q` инициируются какими-то глобальными переменными (в нашем примере – переменными `x` и `y`). Дальше в теле подпрограммы со ссылочными переменными `p` и `q` производятся операции так же, как и над обычными ссылками (см. *п.3.5*).

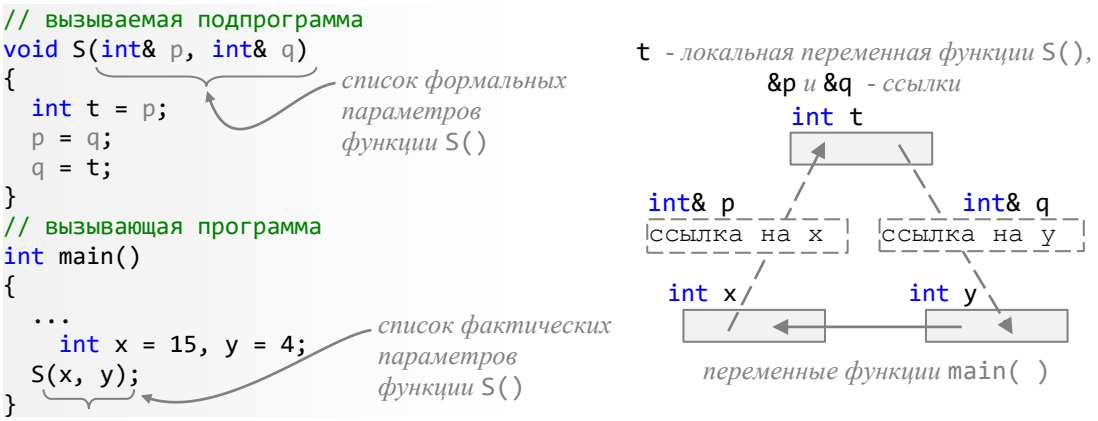

Рис. 39. Передача параметров по ссылке

Изменим пример, приведенный выше (*Листинг 58*), так, чтобы параметры передавались в функцию *по ссылке*. (*Листинг 59*). 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 59

```
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. 40*). Во-первых, можно видеть, что адреса переменных **a** и **b**, выводимые из основной программы и из подпрограммы `sum2ab()` одни и те же. Это можно понять по адресам этих переменных – `001EFA8C` и `001EFA80`. В головной программе и в вызываемой подпрограммы мы работаем с одними и теми же ячейками памяти. Естественно, что значения переменных как удвоились в подпрограмме, так и остались удвоенными при повторном выводе из программы `main()`.

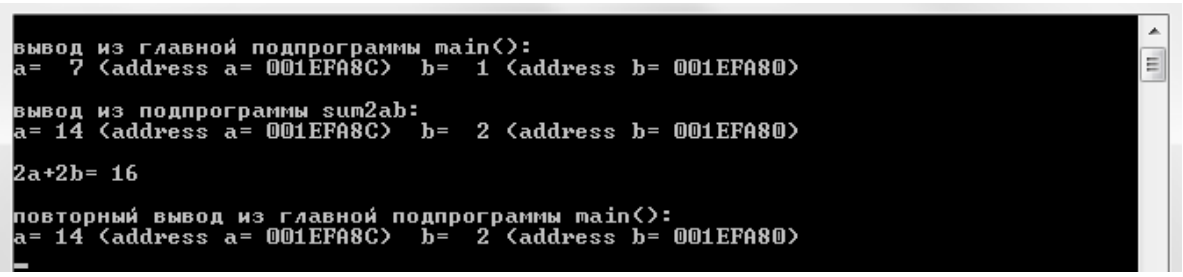

Рис. 40. Результаты выполнения программы

Передача параметров по указателю

Передача параметров по указателю (*Рис. 41*) 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)`. Напомню, что указатели хранят в себе *адреса*.

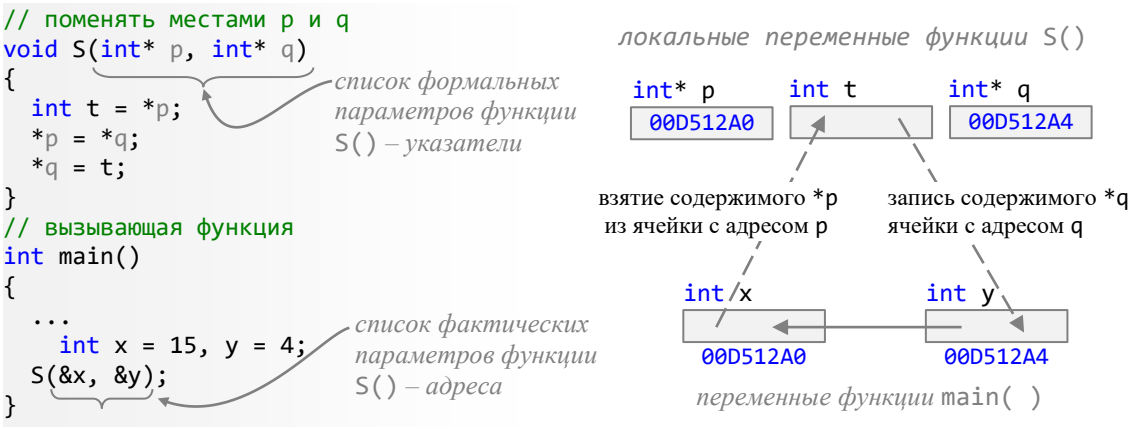

Рис. 41. Передача параметров по указателю

В точке вызова функции `S()` приостанавливается выполнение функции `main()`, в памяти компьютера выделяется еще 3 ячейки под локальные переменные `p`, `q` каждая размером `sizeof(int*)` и ячейку `t` размером `sizeof(int)`. Поскольку `p` и `q` – указатели, то в них копируются не значения глобальных переменных `x` и `y`, а их адреса `00D512A0` и `00D512A4`. Поэтому при вызове функции необходимо это указать `S(&x, &y)`.

В подпрограмме `S()` производится обмен значениями между переменной `t` и содержимым `*p` и `*q` ячеек с адресами, хранящимися в переменных `p` и `q`. А в них хранятся адреса `00D512A0` и `00D512A4` – адреса глобальных переменных `x` и `y`. Поэтому фактически производится обмен данными между `x` и `y`.

Пример, аналогичный (*Листинг 58*), приведен ниже (*Листинг 60*). Рассмотрим его более подробно. В этом примере реализуется передача параметров *«по указателю»*. 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 60

```
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, char* argv[])
{
	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);
}
```

Нужно понимать, что сами по себе переменные-указатели `a` и `b` в подпрограмме `set_ab()` являются локальными переменными и имеют свои собственные адреса (они нас не интересуют). На рисунке ниже (*Рис. 42*) 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`.

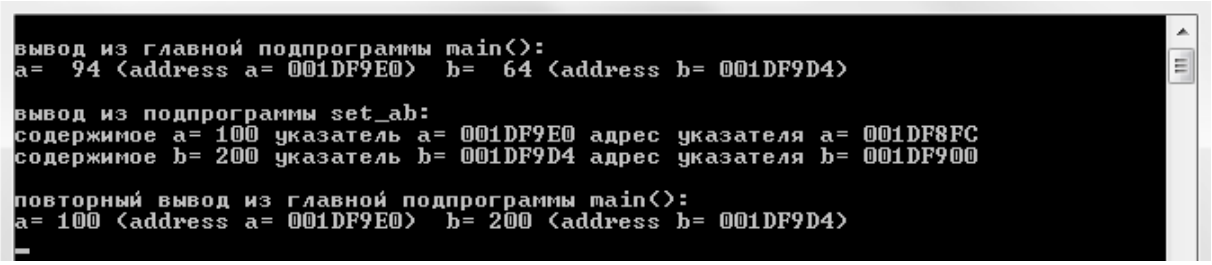

Рис. 42. Результаты выполнения программы

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

Таблица 9. 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`
	`f(&b); // передача адреса b`
	`// b== 123;`
	`}`

Стрелками в таблице (*Таблица 9*) обозначено то, что передаваемые «*по значению*» параметры могут выступать только для ввода данных в подпрограмму, возвращаемые оператором `return` – только для вывода из подпрограммы. А формальные параметры, прописанные «*по указателю*» и «*по ссылке*» могут служить и для ввода, и для вывода данных.

Параметрами функции могут являться и *константные величины*. В этом случае объявленная константой величина (или указатель, или ссылка) не будут изменяемы внутри подпрограммы.

Объект, который является константой при доступе через указатель, может быть изменяемым при доступе другим способом. Это применяется при передаче данных в функцию по указателю, если нежелательно, чтобы изменился указатель или его значение. При объявлении аргумента функции – указателя константой, адрес в нем не будет изменяться в теле функции, если же константой объявлено содержимое указателя, то функции запрещено изменять объект, на который указывает указатель. Вот, например, два варианта программы из библиотеки `<string.h>`, предназначенные для поиска символа в строке (массиве) символов.

```
const char* strchr(const char* p, char c);
char* strchr(char* p, char c);
```

Приведенный в примере первый вариант программы используется для строк, элементы которых не должны быть изменены функцией и возвращает указатель на `const`, который не позволяет изменять результат. Вторая версия используется для изменяемых строк.

Константные ссылки в списке параметров позволяют передавать в функцию в качестве аргумента не само значение переменной (а оно может быть очень большим), а ссылку на неё, но при этом не разрешая её изменять в теле функции. В подпрограмме при этом производится обращение к ссылочной переменной как к обычной локальной переменной подпрограммы. И если ссылка применяется в выражении, где требуется значение исходной переменной, то ссылка автоматически преобразуется к своему значению. Более точно это процесс называется *извлечением значения* из памяти.

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 61 иллюстрирует перезагрузку различных функций, отличающихся списком параметров. По результатам работы программы (*Рис. 43*) можно судить о том, какие из функций вызывались и почему:

```
                                                                Листинг 61
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. 43. Результаты выполнения программы

При прегрузке функций важно понимать, что функции с одинаковыми именами различаются по списку формальных параметров (но не по возвращаемому значению!) и здесь может возникнуть неопределенность в случае преобразования типов. Рассмотрим, пример, приведенный ниже (*Листинг 63*) – в нем описаны две функции с одинаковым именем и разными списками параметров – константная символьная строка и вещественное значение. Первые две строки программы `main()` будут поняты компилятором однозначно: для выполнения команды `print("string")` будет вызвана первая функция, а при выполнении `print(1.234567)` – вторая.

Листинг 62

```
void print(const char* str)
{      printf("вывод строки = %s\n", str);
}
void print(double X)
{      printf("вывод double X= %e\n", X);
}
void _tmain()
{      print("string"); // однозначная интерпретация
       print(1.234567);
       print(1); // неявное преобразование типа данных
       print(0); // ошибочный вариант – неоднозначная интерпретация

}
```

С третьей строкой нужно разбираться: при вызове функции `print(1)`, её аргумент `1` (целое число), может быть неявно преобразован к типу `double`, а к типу `const char*` преобразован быть не может. Поэтому вызвана будет вторая функция `print(double)` с типом аргумента `double`.

А вот в четвертой строке программы `main()`, при вызове `print(0)`, аргумент функции может быть преобразован как в вещественное число `0.0`, так и в константную

строку "\0". Поэтому такой вызов будет помечен компилятором как ошибка. Неоднозначная интерпретация аргумента не дает компилятору сделать выбор. Забавно, что если бы в программе присутствовала только какая-то одна из этих функций, причем любая, ошибки бы не было.

К понятию перегрузки функций языка C++ относится и метод *маскировки* (или маскирования) – это способ перегрузки функции (или метода класса) при котором полностью всё 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 63*) замаскировать стандартную функцию библиотеки <math.h> возведения в степень double pow(double, double):

Листинг 63

```
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. Но вот, зачем нам такое делать?

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 10)*, 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 10. 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)

Функция	Описание
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
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 и параметров работы с математическими расширениями , настроенными в операционной системе.

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. 5* можно видеть окно вывода

«*Output*», в котором отражается процесс компиляции и выполнения программы, а также наличие в ней ошибок.

Для просмотра информации об обнаруженных компилятором ошибках необходимо перейти во вкладку «*Error List*» (*Рис. 6*). 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.

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. 44*).

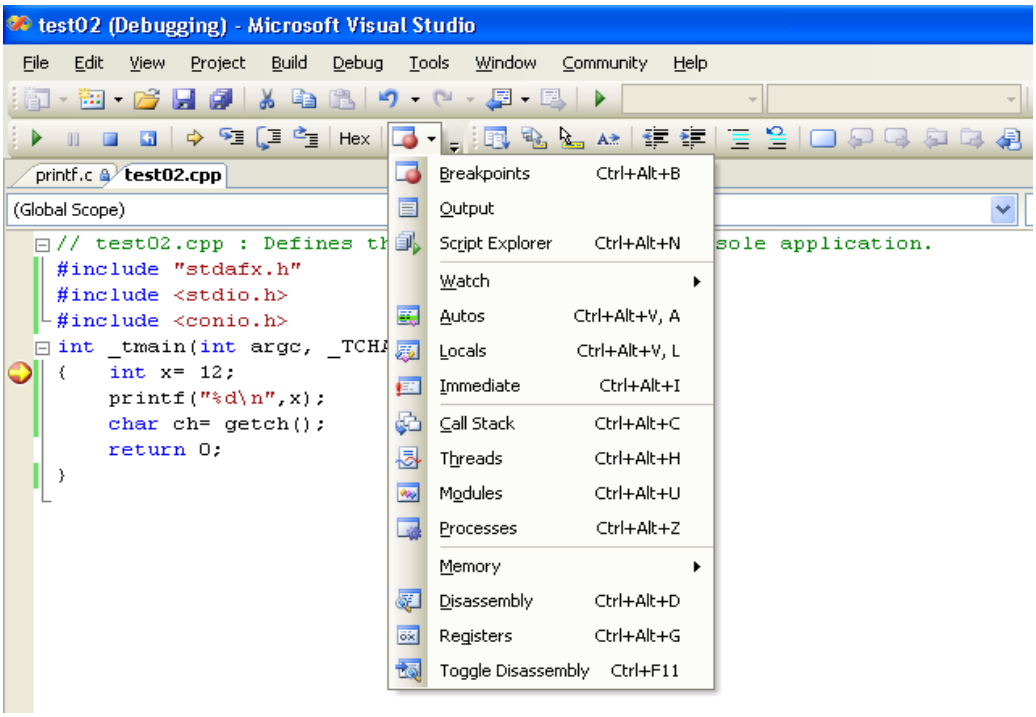

Рис. 44. Пошаговая трассировка программного кода. Точка останова

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. И дальше выполнение программы пойдет под «ручным» управлением программиста пошагово.

Расставив в нужных местах программы точки останова (на *Рис. 44* точка останова отображена красным кружком) можно пошагово проследить выполнение операторов программы при помощи команд *StepInto* <*F11*>, *StepOver* <*F10*> и *StepOut* <*Shift+F11*>: на *Рис. 44* 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. 45*). В этих вкладках указываются имена, типы и текущие значения переменных.

На рисунке (*Рис. 45*) 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.

В процессе отладки в текущей строке кода, отмеченной желтым указателем выполнения, при наведении указателя мыши на любой объект программного кода, можно получить контекстную подсказку по нему.

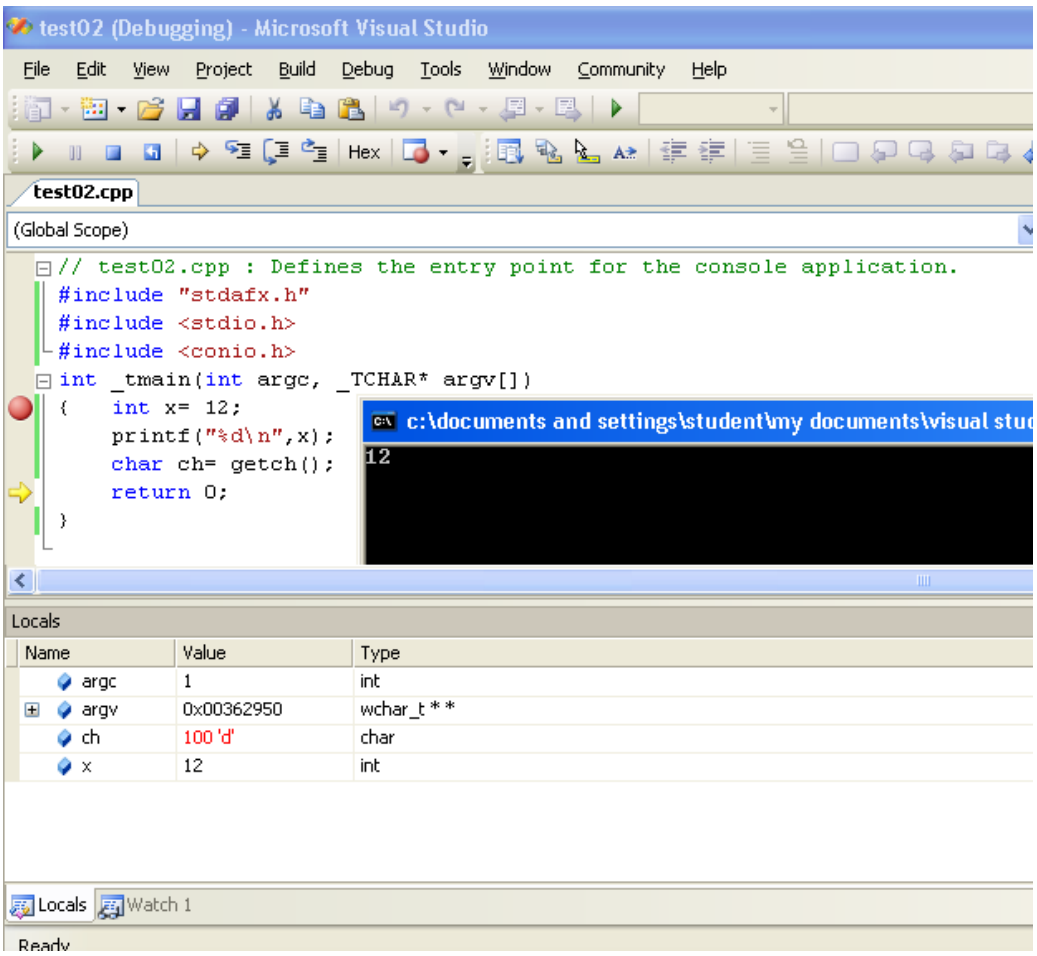

Рис. 45. Пошаговая трассировка. Просмотр значений переменных

Если в отлаживаемой строке программного кода (помеченной желтой стрелкой) находится подпрограмма, то можно выполнить её за один шаг («перешагнуть» её – *StepOver*), нажав <*F10*>. А можно войти внутрь подпрограммы (*StepInto*) <*F11*>, продолжив пошаговую трассировку команд, содержащихся внутри подпрограммы. При этом передача и возврат данных в/из подпрограммы будет осуществляться аналогично процессу реального выполнения приложения.

Контрольные вопросы:

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

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

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

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

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

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

7. 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 «*массив*» – это набор данных *одного и того же типа*, собранных под одним именем. *Элемент массива* определяется именем массива и порядковым номером (*индексом*). [*3, 8*] 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]...[размерN]; // 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 64

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

void _tmain(int argc, char* 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]));
	system("pause");
}
```

Рис. 46. Вывод на экран массива чисел с адресами ячеек

Элементы массива хранятся в одном блоке памяти последовательно и без разрывов (*Рис. 46*). Например, если 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, занимающее 4 байта (*Рис. 47*). Поэтому расстояния между адресами идущих подряд ячеек в данном случае будет 4 байта.

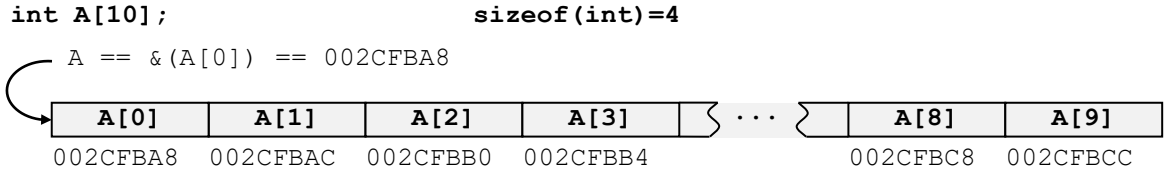

Рис. 47. Размещение в памяти элементов массива

`Имя_массива` – это адрес, начиная с которого последовательно хранятся все ячейки массива (см. *Листинг 64, Рис. 46*).

5.1. Указатели и массивы в C++

В языке C++ существует два способа обращаться к ячейкам памяти (переменным) – *по имени* и *по адресу*. Ячейки массива расположены в памяти одним монолитным блоком (*Рис. 47*) и имеют одно общее имя. Но они имеют и адреса, отстоящие друг от друга на размер ячейки массива (*Рис. 46*). Поэтому доступ к членам массива можно получать как через *имя массива и индекс*, так и по методу *адрес+смещение*. Имя массива в этом случае понимается как *адрес* начального (нулевого) элемента массива, а индекс ячейки – *смещение* этой ячейки относительно начального адреса.

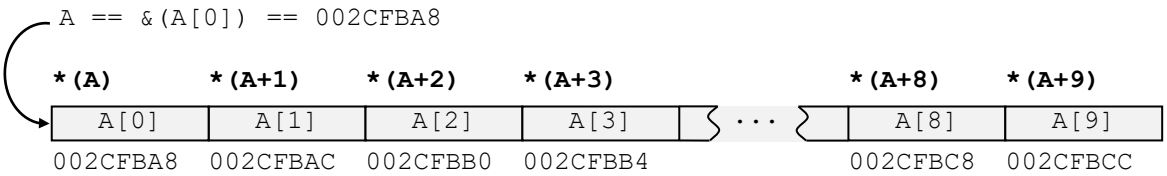

Рис. 48. Доступ к элементам массива по адресу и смещениию

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 65*) 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 означает увеличение адреса `A` не на `i` байт, а на `i` ячеек соответствующего размера, в нашем случае – на `i*sizeof(int)` байт.

Листинг 65

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

void _tmain(int argc, char* 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);
	}
	system("pause");
}
```

Результаты работы программы (*Листинг 65*) приведены на рисунке ниже (*Рис. 49*), 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.

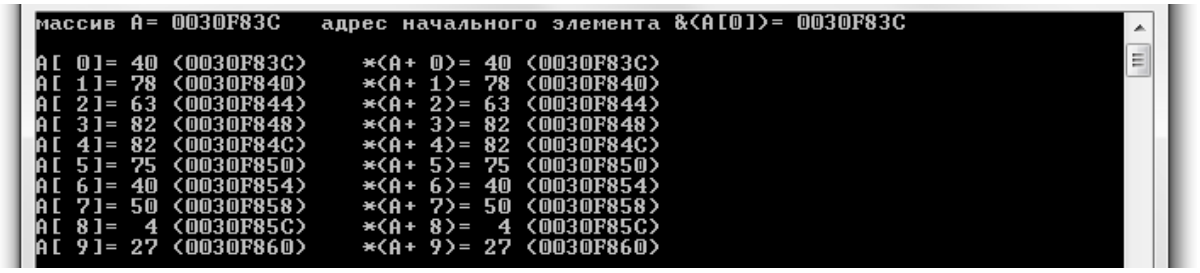

Рис. 49. Доступ к элементам массива по адресу и смещению

Нужно понимать, что приоритет операции `"*"` выше, чем у операции `"+"`. Поэтому во всех выражениях типа `*(A+i)` обязательно ставятся скобки. Они обеспечивают первоочередность операции вычисления смещения. Иными словами, запись `*(A+i)` в первую очередь производит вычисление смещенного адреса ячейки массива `A+i`, и только во вторую очередь – взятие содержимого по этому адресу. Если же скобки не ставить в выражении `*A+i`, то сперва будет вычислено значение `*A`, лежащее по адресу `A`, а затем это значение будет увеличено на `i` как число. Ясно, что эта ошибка не будет обнаружена компилятором, так как формально – ошибки нет.

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()`.

Функция `malloc( )` из примера (*Листинг 66*) возвращает нетипизированный адрес выделенной под массив блок памяти в виде `void*`, так что его перед сохранением в указателе `B` необходимо принудительно типизировать `B= (double*)…`

Листинг 66

```
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
```

Возвращаемый функцией выделения памяти указатель можно использовать для контроля корректного выделения памяти, как показано в примере ниже (*Листинг 67*). 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 67

```
#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. 50. 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 67, Рис. 50*) 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. 50*) 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 и сравнение массивов

Как уже многократно говорилось: *имя массива – это адрес,* начиная с которого хранится массив. Поэтому невозможно присвоить один массив другому одним оператором присваивания `current=prev` (*Листинг 68*). При этом будет скопирован только адрес статического массива `prev` в указатель `current` (*Рис. 51*). Память под новый динамический массив в ОЗУ компьютера при этом выделена не будет. Массивы должны копироваться *поэлементно*, например, при помощи цикла:

Листинг 68

```
float prev[20000], * current;
for (int i = 0; i < 20000; i++)
       prev[i] = i;
// current = prev; // не правильно – копируется только адрес, а 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++;

}
```

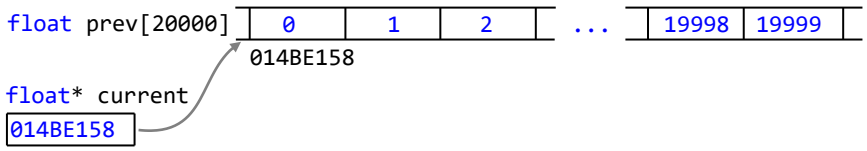

Рис. 51. Копирование адреса динамического массива вместо его содержимого

Корректное выделение памяти под динамический массив `current` и копирование в него элементов из массива `prev` (*Листинг 68*) показано на схеме (*Рис. 52*).

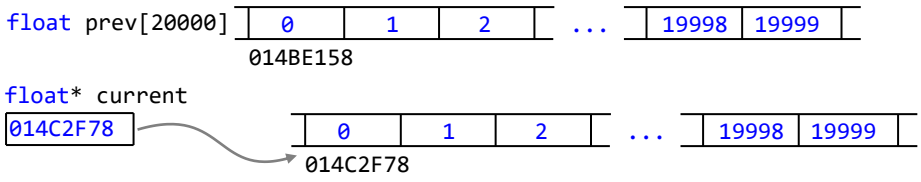

Рис. 52. Корректное копирование динамического массива

Очевидно, что при сравнении массивов также возможна путаница между их содержимым и их адресами. Сравнение `prev == current`, естественно, будет сравнивать адреса массивов. В первом случае (*Рис. 51*) это сравнение (логическое выражение) даст результат `true`, а во втором (*Рис. 52*) – `false`.

Ни тот, ни другой случай не являются формальной ошибкой, компилятор такие вещи не анализирует. Поэтому программист вправе производить и поэлементное сравнение (или копирование) массивов, и поадресное – смотря что ему требуется.

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 будет иметь непосредственный доступ к элементам данного массива *«по адресу»*.

Листинг 69

```
void String30Copy(char[], char[]); // прототип функции
//---------------------------------------------------------------------------
void _tmain(int argc, _TCHAR* argv[])
{
	char current[30], target[30];
	String30Copy(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. 47*). 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` (*type definition* – определение типа). Введенное таким образом *новое* имя типа можно использовать таким же образом, как и имена стандартных типов данных.

```
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 препроцессирования 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 65*, *Листинг 67* и неудачные – *Листинг 68*, *Листинг 69*.

Пример

Рассмотрим пример (*Листинг 70*), в котором при помощи подпрограмм заполняются случайными числами два динамических массива, состоящие из 100 и 7 целых чисел, а также вычисляется минимальный элемент в каждом из них.

Листинг 70

```
#include <iostream>
  //-------------------------------------------------------------------------
  int min_mas(int* D, int N)
  {
    int m = D[0];
    for (int j = 1; j < N; j++)
    {
      if (D[j] < m) // если j-тый элемент массива меньше минимума
      {
        m = D[j];  // значит, это новый минимум
      }
    } return m;
  }
  //-------------------------------------------------------------------------
  void set_mas(int* Z, int N, const char* Name)
  {
    for (int i = 0; i < N; i++)
    {
      Z[i] = rand() % 90 + 10; // от 10 до 100
      std::cout << Name << "[" << i << "]= " << Z[i] << "\n";
    }
  }
//-------------------------------------------------------------------------
int main()
{
    system("chcp 1251");
    srand(time(NULL));
    int* d;
    d = new int[100]; // выделение памяти под динамический массив из 100 чисел
    set_mas(d, 100, "d"); // заполнение и вывод на экран
    // вычисление минимума в массиве d и вывод его на экран
    std::cout << "min_mas(d, 100)= " << min_mas(d, 100) << "\n";
    int* a;
    a = new int[7]; // выделение памяти под динамический массив из 7 чисел
    set_mas(a, 7, "a"); // заполнение и вывод на экран
    // вычисление минимума в массиве a и вывод его на экран
    std::cout << "min_mas(a, 7)= " << min_mas(a, 7) << "\n";
    system("pause");
}
```

Первая подпрограмма `int min_mas(int*, int)`, как можно видеть из её описания, имеет два формальных параметра – указатель на целое число `int* D` и целое число `int N`. В качестве результата эта функция должна возвращать целое число типа `int`. Программа предназначена для нахождения минимального элемента в динамическом массиве, состоящем из `N` целых чисел (адрес которого расположен в указателе `D`).

В самом начале этой подпрограммы задается целая переменная `int m`, в которой будет храниться минимальный элемент массива `D`. Эта переменная сразу же задается начальным (нулевым) элементом массива `D[0]`. Тем самым делается начальное предположение, что минимальный элемент – это `D[0]`. Затем в цикле со счетчиком `for(int j=1; j<N; j++)` выполняются `N-1` раз следующие действия. Каждый элемент массива `D[j]` сравнивается с предположительно минимальным элементом `m`; если очередной элемент меньше минимального `m`, то значит он (а вовсе не `m`) есть новый минимум и, следовательно, переменной `m` нужно присвоить новое значение `D[j]`. В противном случае – если переменная `m` не меньше (больше или равна) j-того элемента массива `D`, значит ничего менять на этом шаге цикла не нужно.

Сравнив в цикле все элементы массива `D` неминуемо получим в переменной `m` значение наименьшего значения среди всех ячеек массива, в конце вернем его при помощи оператора `return`.

Следующая подпрограмма `void set_mas(int* Z, int N, const char* Name)` – это процедура, так как не возвращает никаких значений, в описании подпрограмма имеет тип `void`, а в её теле нет оператора `return`. Она просто заполняет в цикле все ячейки массива, адрес которого передается в подпрограмму через указатель `int* Z`, размерность массива – `int N`, а указатель на константу `const char* Name` содержит адрес постоянной строки символов – имя массива для печати на экран.

Оцените, как лаконично выглядит теперь программа `main()` – сперва выделяется память под динамический массив `d` из `100` целых чисел и массив `a` из `7`. Затем, для каждого из них вызывается подпрограмма `set_mas()` со своим набором параметров: `set_mas(d, 100, "d")` для первого массива и `set_mas(a, 7, "a")` – для второго. После чего вызывается подпрограмма `min_mas()` с фактическими параметрами, соответствующими заданными массивам. Возвращаемые ею значения сразу же выводятся на экран оператором `std::cout <<`. Фрагмент работы этой программы на *Рис. 53*.

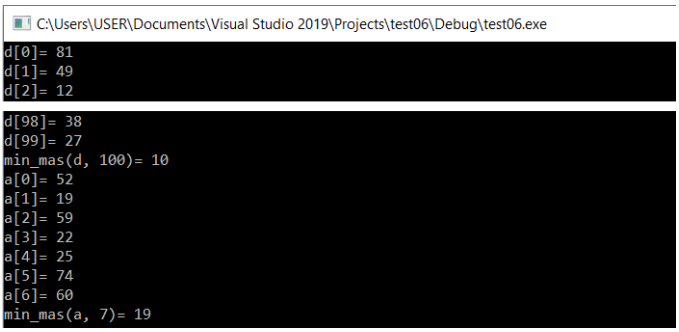

Рис. 53. Результаты работы подпрограмм с массивами

Обратите внимание, как элегантно передаются динамические массивы внутрь подпрограммы – нет нужды копировать все элементы массива (а он может быть огромным) в вызываемую подпрограмму (*дочернюю*) из программы основной (*родительской*), достаточно передать в нее *адрес* динамического массива. Массив `d`, например, занимает в памяти 400 байт, а его адрес – 4 байта.

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?*

6. Двумерные массивы (Матрицы)

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, шифрованием, компрессией и т.п.

6.1. Статический двумерный массив

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

Листинг 71

```
#include "stdafx.h"
#include <iostream>
#include <time.h>
using namespace std;
//--------------------------------------------------------------------------
#define N 5 // макросы размерность массива
#define M 3
//--------------------------------------------------------------------------
// прототипы функций
void Set_Array_NxM_rand(double Q[N][M]);
void Print_Array_NxM(double Q[N][M], const char*);
//--------------------------------------------------------------------------
void _tmain(int argc, char* argv[])
{	system("chcp 1251");  //setlocale(LC_ALL, "Russian");
	double F[N][M]; // статический двумерный массив
	Set_Array_NxM_rand(F); // вызов подпрограмм
	Print_Array_NxM(F, "F");
	system("pause");
}
//--------------------------------------------------------------------------
// заполнение матрицы размерности N*M случайными числами
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;
		}
	}
}
//--------------------------------------------------------------------------
// вывод матрицы размерности N*M на экран
void Print_Array_NxM(double Q[N][M], const 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");
	}
}
```

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. 54*.

При описании списка параметров подпрограммы, в которую необходимо передать статический массив, необходимо указывать размерность этого массива, как в примере выше (*Листинг 71*).

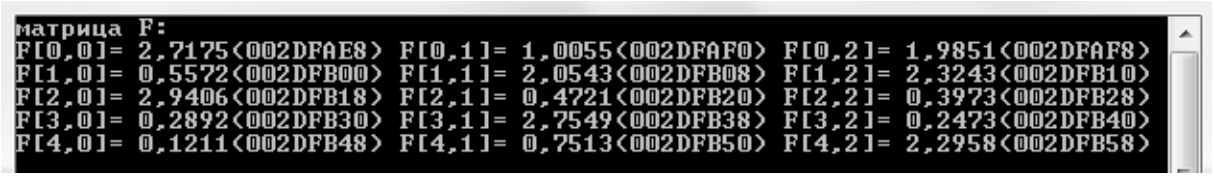

Рис. 54. Адресация элементов двумерной статической матрицы

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 72*) в тело программы, работающей со статическим двумерным массивом `double F[N][M]` (*Листинг 71*).

Листинг 72

```
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]);
```

Рис. 55. Адресация строк двумерной статической матрицы

Сравнив полученные результаты (*Рис. 55*) с предыдущими (*Рис. 54*), можно видеть, что `&F[i]` – адреса строк `F[i]` двумерной матрицы `F` совпадают с адресами начальных элементов этих строк `&F[i][0]`, где `i = 0..N-1`. Кроме того, адрес `F` самой матрицы совпадает с адресом `&F[0]` начальной строки `F[0]` и адресом начального элемента `&F[0][0]`.

Вы, конечно, заметили, как меняются в цикле переменные `int i` и `int j` – счетчики строк и столбцов матрицы. Счетчик строк `i= 0..N-1`, а счетчик столбцов (элементов в строках) `j= 0..M-1`. Напомню, что в языке C++ элементы массивов нумеруются с `0`.

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

6.2. Двумерный динамический массив в виде массива 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 *динамического массива указателей*, каждый из которых содержит адрес одномерного динамического массива – строк (или столбцов) матрицы.

Этот способ выделения памяти под двумерный N×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. 56*), память под которые так же выделяется динамически:

Листинг 73

```
//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];
}
```

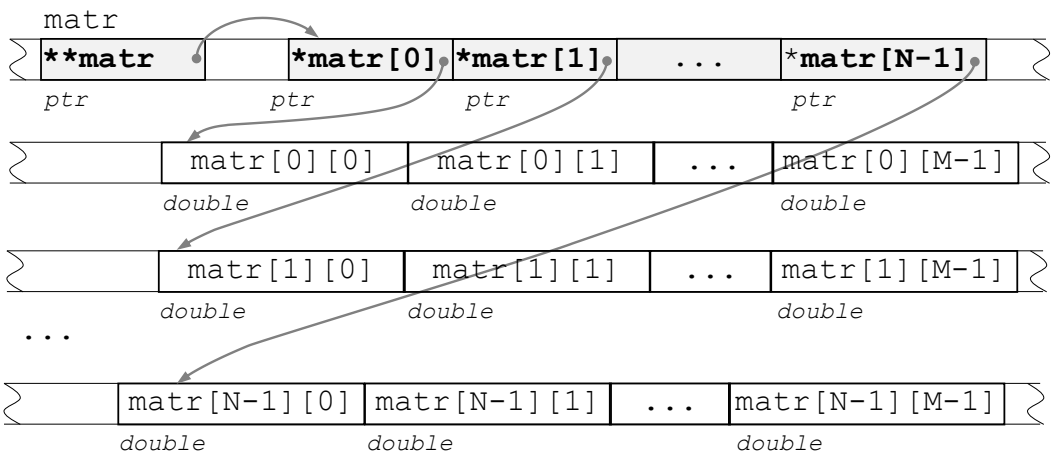

Рис. 56. Расположение двумерного динамического массива в памяти

На рисунке (*Рис. 56*) 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 73*), отдельно под каждую строку. Поэтому строки расположены не обязательно одна за другой – одномерные массивы строк могут располагаться в разных областях памяти.

Листинг 74

```
double** dd; // указатель для массива указателей
int M = 4; // размерность матрицы N*M
int N = 3;
dd = new double*[N]; // выделение памяти под массив указателей
for (int i = 0; i < N; i++)// выделение памяти под N массивов - строк матрицы
        dd[i] = new double[M]; // строка из M элементов
```

При таком способе выделения памяти (см. *Листинг 74*) 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*. В них предполагается хранить адреса одномерных динамических массивов dd[i] (строк матрицы).

На начальном этапе массив указателей не заполнен, поэтому дальше в цикле по счетчику i= 0..N-1 выделяется память под одномерные массивы рациональных значений dd[i] и адреса этих массивов сохраняются в соответствующих ячейках массива указателей dd (*Листинг 74*).

Листинг 75
```
int n, m; // n и m – количество строк и столбцов матрицы
double** dd; // указатель для массива указателей
// выделение динамической памяти под массив указателей
dd = (double**)malloc(n * sizeof(double*));
for (int i = 0; i < n; i++)
      // выделение динамической памяти для n массивов – строк матрицы
        dd[i] = (double*)malloc(m * sizeof(double)); // строка из m элементов
```

Функции `malloc()` и `calloc()` при выделении памяти под массив указателей и под массивы-строки возвращают *нетипизированный* указатель `void*`, таким образом, необходимо выполнять его явное преобразование в указатель нужного типа. Например, в программе (*Листинг 75*) указатель на массив строк типизируется в `double**`, а указатели на строки матрицы – в тип `double*`. Элементы строк – имеют тип `double`.

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_1][ЗначениеИндекса_2];
matr[i][j];
```

Рассмотрим (*Листинг 76*) присвоение значений двумерному массиву – матрице `dd`.

Листинг 76
```
// 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`.

Листинг 77
```
//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 функций `malloc()` и `calloc()` (*Листинг 75*) освобождается при помощи соответствующей им библиотечной функции `free()`:

Листинг 78

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

Рассмотрим пример (*Листинг 79*) конструирования в памяти двумерного динамического массива, заполнения его значениями, вывода на экран и корректного удаления. Основные операции с элементами матрицы – выделение памяти, заполнение случайными элементами `Set_Matrix_NxM()`, вывод на экран `Print_Array_NxM()` и освобождение памяти `Free_Array_NxM()` реализовано при помощи подпрограмм.

Видно, что матрица размещена в памяти аналогично статической, но имеются отличия.

Листинг 79

```
#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[])
{
	system("chcp 1251");
	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);
	system("pause");
}
```

Из рисунка (*Рис. 57*), отображающего результаты работы программы (*Листинг 79*) можно видеть, что адреса, хранящиеся в указателях `R[i]` совпадают с начальными элементами соответствующих строк `&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. 57*) и двумерного статического массива (*Рис. 54*, *Рис. 55*).

```
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. 57. Адресация элементов двумерной динамической матрицы

6.3. Двумерный динамический массив в виде одномерного массива

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 для хранения всех `N*M` элементов в виде одной строки (*Рис. 58*), например, при помощи оператора `new`, или посредством функций `calloc()` или `malloc()`.

```
#include <alloc.h>
float* A;
int n, m;
A = (float*)calloc(N * M, sizeof(float));
```

Описанным 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);
```

В этом случае, все элементы двумерного массива (матрицы) хранятся в одномерном массиве размером `N*M` элементов построчно. Сначала в этом массиве расположена нулевая строка матрицы, затем без разрывов – первая строка, вторая, 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 **`k = i·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 80*, *Рис. 59*) компенсируется неудобством доступа к ее элементам.

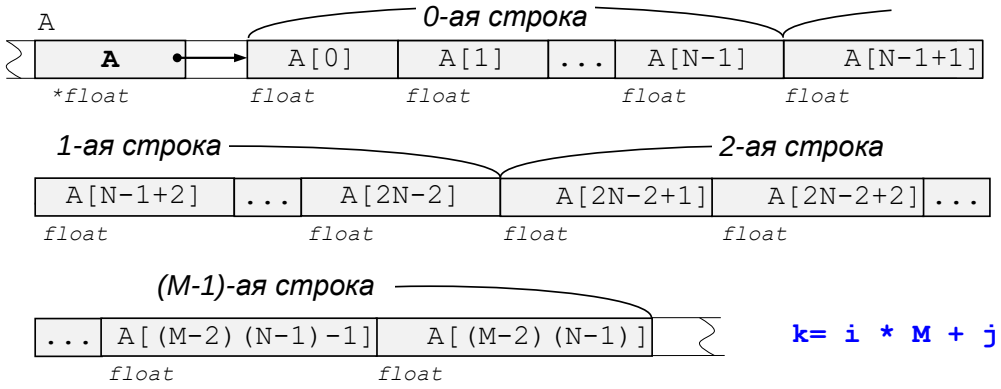

Рис. 58. Распределение в памяти элементов двумерного массива «в строку»

В качестве примера работы с динамическими матрицами рассмотрим следующую задачу (*Листинг 80*): Заданы две матрицы вещественных чисел `A[N*M]` и `B[N*M]`. Вычислить сумму матриц – матрицу `C = A+B`.

Листинг 80

```
#include "stdafx.h"
#include <iostream>
//-----------------------------------------------------------------------------
using namespace std;
void _tmain(int argc, char* argv[])
{
    system("chcp 1251"); // 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;
    //-------------------------------------------------------------------------
    // выделение памяти для матриц
    a = new double[N * M];
    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");
}
```

Рис. 59. Результаты работы программы

Как распределены в памяти элементы матрицы, представленной в виде одномерного массива, можно видеть, если вывести адреса матрицы `C` на экран (*Рис. 60*):

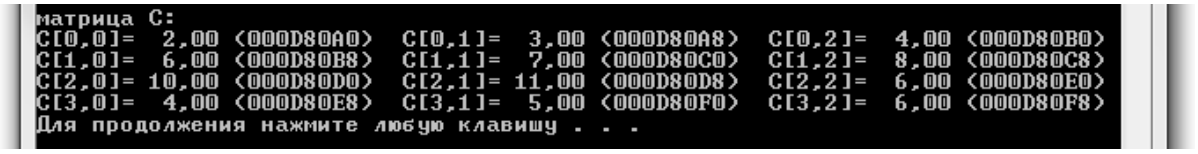

Рис. 60. Распределение в памяти элементов двумерного массива «в строку»

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*».

7. Работа со строками

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

Строки в языке *C++* позволяют работать с символьными данными и текстом [*9, 14*].

7.1. 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'` (невидимая 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 81

```
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.

Аналогичные сложности можно встретить, если попытаться скопировать содержимое одной строки в другую.

Для работы с массивами символов используются подпрограммы, которые будут приведены позднее (*Таблица 11*, *Таблица 12*). Важно рассмотреть, как представлены строки в виде массива символов и как понимать строки с точки зрения указателей.

7.2. Статические и динамические строки

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

Листинг 82

```
#include <iostream>
//-----------------------------------------------------
void _tmain(int argc, char* argv[])
{      // задать статический массив символов – строку
       char str1[21] = "1234567890qwertyuiop";
       std::cout << str1 << endl;
}
```

Выделение памяти под статический массив символов происходит в момент задания: определяется его длина L, выделяется непрерывная область памяти размера L+1,

в первые *L* ячеек массива помещаются символы `"1234567890qwertyuiop"`. В последнюю ячейку помещается *невидимый* символ `'\0'` (конец строки), необходимый только для организации вывода элементов массива.

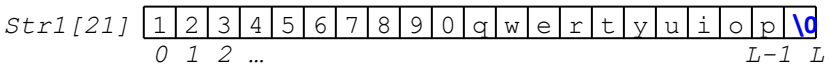

Рис. 61. Строка – это символьный массив

Имя `str1` (имя статического массива символов) представляет собой указатель, как и имя числового массива, это иллюстрирует *Рис. 61*.

Вывод массива символов на экран

При выводе оператор `<<` класса `std::cout` будет выводить все символы массива, начиная от указателя `str1` до символа `'\0'`, сколько бы их не было. В примере, приведенном выше (*Листинг 82*), в 21ой позиции строки стоит символ окончания строки, поэтому все 20 символов выводятся на экран, как показано на *Рис. 62*.

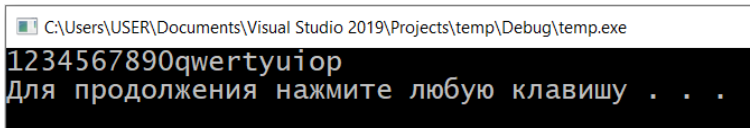

Рис. 62. Вывод на экран строки символов `str1`

Если же символ `'\0'` разместить в середину строки `str1` (*Листинг 83*), то при выводе массива на экран отразятся не все символы строки, а только от начала до первого встретившегося символа `'\0'`. Результат работы этой программы приведен на *Рис. 63* – можно видеть, что в этом случае выводится только 9 символов.

Листинг 83

```
char str1[21] = "1234567890qwertyuiop";
std::cout << str1 << "\n";
str1[9] = '\0'; // помещаем символ '\0' в середину строки
std::cout << str1 << "\n";
```

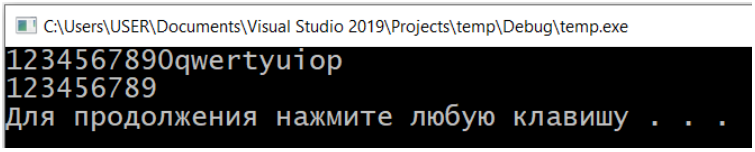

Рис. 63. Вывод на экран строки символов `str1` *с символом* `'\0'` *в середине*

Если напротив символ `'\0'` в конце строки `str1` удалить, то вывод символов не прекратится после окончания строки `str1`. Ячейки памяти, следующие за строкой `str1`, будут интерпретироваться как символы и выводиться на экран, пока не встретится какой-нибудь другой символ `'\0'` – признак окончания строки.

Листинг 84

```
char str1[21] = "1234567890qwertyuiop";
std::cout << str1 << "\n";
str1[20] = '\0'; // заменяем символ '\0' в конце строки символом 'Z'
std::cout << str1 << "\n";
```

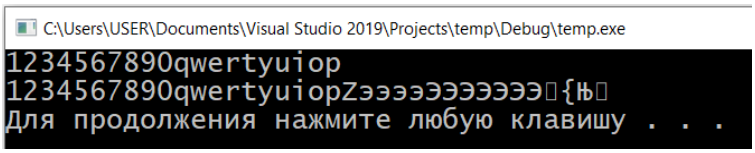

Рис. 64. Вывод на экран строки `str1` *с утраченным символом конца строки*

В примере (*Листинг 84*) символ конца строки был заменен символом `'Z'`. Результат действия этой программы приведен на *Рис. 64*: видно, что после буквы `'Z'` выводится ряд непонятных символов. Это попытка двоичный код, находящийся в ячейках ОЗУ по адресам, следующим за строкой `str1`, интерпретировать как тип `char`.

Динамический массив символов

Статическая строка – это массив символов, следовательно, имя этого массива `str1` можно интерпретировать как указатель, содержащий адрес начальной его ячейки. Иными словами, содержимое указателя `str1` – это содержимое ячейки `str1[0]`.

Исходя из этого несложно создать динамический массив символов (динамическую строку), построенную по принципу выделения динамической памяти указателю `char* pstr`. Выделить память можно известным оператором `new` или функциями `calloc()` и `malloc()`, как показано в программе ниже (*Листинг 85*). Схема на *Рис. 65* иллюстрирует принцип выделения памяти под динамическую строку.

Листинг 85

```
  char* pstr;
  pstr = (char*)malloc(11 * sizeof(char)); // выделить 11 ячеек типа char
  for (int i = 0; i < 10; i++) // заполнить их символами от 'A' до 'J'
    pstr[i] = 'A' + i;
  pstr[10] = '\0'; // в последнюю 11-ю ячейку поместить символ '\0'
  cout << pstr << "\n"; // вывести строку на экран
//-------------------------------------------------------------------------
  cout << "(void*)pstr= " << (void*)pstr << "\n"; // адрес pstr
  cout << "(void*)pstr[0]= " << (void*)&(pstr[0]) << "\n"; // адрес начальной
ячейки
```

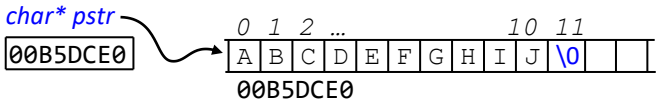

Рис. 65. Создание динамической строки

В данном примере под строку `pstr` при помощи функции `malloc()` выделяется динамическая память – 11 ячеек размера `sizeof(char)`. Однако, поскольку автоматически символ конца строки программой `malloc()` не создается, требуется поместить в последнюю ячейку массива `pstr[10]` символ конца строки `'\0'`.

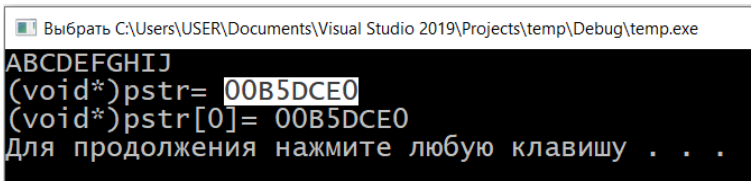

Рис. 66. Создание динамической строки

В последних двух строках фрагмента программы (*Листинг 85*) выводятся 2 адреса – адрес, хранящийся в указателе `pstr` и адрес начальной (нулевой) ячейки динамического массива. Перед выводом на экран эти адреса пришлось типизировать в вид `(void*)`, так как иначе оператор `cout<<` выводит на экран строку. Можно видеть, что это один и тот же адрес – как и для числовых динамических массивов.

Не забывайте в последнюю ячейку символьного массива располагать символ конца строки при создании массива «вручную». В этом случае программист сам должен контролировать размер массива символов и размещение символа `'\0'` в конце строки.

7.3. Операции со строками

Копирование строк

При копировании строк важно понимать, что при прямом присваивании: `str2 = pstr` копируются только указатели. Т.е. адрес одного динамического массива присваивается указателю на другой динамический массив. Верно ли это? Такое присваивание не является ошибкой, так как в каких-то случаях может быть необходимо получить еще один указатель `str2` на тот же самый динамический массив `pstr`. Схематически такая ситуация изображена на *Рис. 67*.

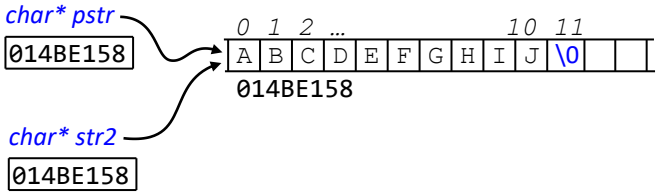

Рис. 67. Копирование адреса динамической строки вместо её содержимого

Но как поступить, если необходимо скопировать содержимое одной динамической строки в другую? Рассмотрим пример (*Листинг 85*), продолжающий программу, начатую в первой части этого листинга.

Листинг 86

```
cout << "строка pstr= " << pstr << "\n"; // вывести строку pstr на экран
cout << "адрес pstr= " << (void*)pstr << "\n\n"; // адрес pstr
//-------------------------------------------------------------------------
char* str2 = pstr; // скопирован только указатель
cout << "строка str2= " << str2 << "\n";
cout << "адрес str2= " << (void*)str2 << "\n\n"; // адрес pstr
//-------------------------------------------------------------------------
char* str3; // создание указателя
int count = strlen(pstr); // длина строки без учета символа '\0'
cout << "длина строки pstr= " << count << "\n";
str3 = (char*)calloc(count+1, sizeof(char)); // выделение памяти под str3
strcpy_s(str3, count+1, pstr); // копирование символов из одной строки в другую
cout << "строка str3= " << str3 << "\n";
cout << "адрес str3= " << (void*)str3 << "\n\n"; // адрес pstr
```

Для корректного копирования строк нужно понимать, что указатели `str2` и `str3` – это только переменные, хранящие адреса. Чтобы скопировать строку необходимо предварительно выделить под неё требуемый объем памяти в ОЗУ. А уже потом переносить в эту область памяти символы.

В примере сначала в переменную `count` заносится размер строки `pstr` при помощи подпрограммы `strlen(pstr)`. Эта подпрограмма выясняет именно количество выводимых на экран символов – от начала строки до конца строки `'\0'`.

Затем функция `calloc()` выделяет область памяти под будущую строку и адрес этой области ОЗУ сохраняется в указателе `str3`. Заметьте что выделяется `count+1` ячейка типа `char`. И, наконец, при помощи функции `strcpy()`, копируется `count+1` символов из динамической строки `pstr` в динамическую строку `str3`.

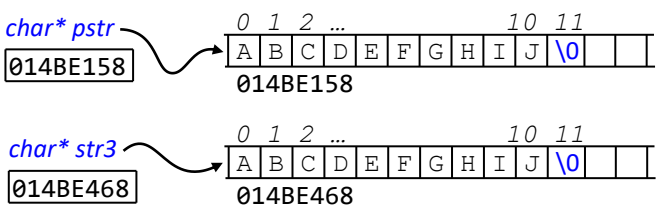

Рис. 68. Копирование символов из одной строки в другую динамическую строку

Результат работы обеих частей программы (*Листинг 85*) приведен ниже на *Рис. 69*. Нужно ясно понимать, что строка – это не только указатель, но и область памяти (динамическая или статическая) в которой лежит сам текст строки.

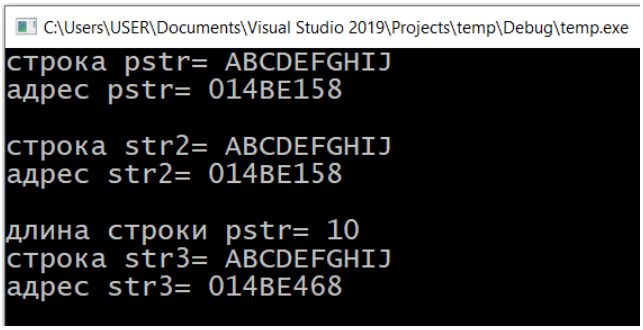

Рис. 69. Результат работы программы (Листинг 85, часть 2)

Здесь копирование `str2 = pstr` не является правильным, так как копируется только указатель на строку, а массив данных не дублируется (*Рис. 67*). Правильное копирование строк – это выделение памяти и поэлементное копирование одного массива в другой – вручную или с использованием подпрограммы `strcpy()`.

Полный список подпрограмм из библиотеки `<string.h>`, предназначенных для работы со строками содержит *Таблица 11*, приведенная ниже.

Оператор `cout <<` выводит во всех трех случаях одинаковую строку (*Рис. 69*), но в первом и третьем случае (`pstr` и `str3`) – это вывод самостоятельных массивов, а при выводе `str2` выводится массив `pstr`, на который указывает адрес `str2`.

Сравнение строк

Таким же образом происходит и сравнение строк. В примере ниже (*Листинг 87*) строка `str2` является полной точной копией строки `str1`, однако, простое сравнение (`str2 == str1`) даст отрицательный результат (см. *Рис. 70*).

Листинг 87

```
char str1[] = "abcdefghijklmnopqrstuvwxyz";
char str2[] = "abcdefghijklmnopqrstuvwxyz";
cout << str1 << "\n";
cout << str2 << "\n";
if (str1 == str2)
  cout << "строки равны\n";
else
  cout << "строки не равны\n";
```

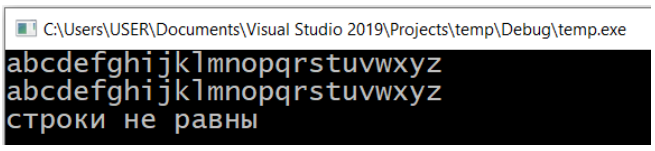

Рис. 70. Результат работы фрагмента программы

В этом случае (так же как и при копировании строк) сравниваются адреса массивов, а не их содержимое. А адреса у них разные.

Посимвольно сравнить содержимое строк можно при помощи встроенной функции `strcmp()`, как в листинге, приведенном ниже (*Листинг 88*).

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 эта подпрограмма всегда возвращает `1(true)` в случае неравенства строк).

Листинг 88

```
char str1[] = "abcdefghijklmnopqrstuvwxyz";
char str2[] = "abcdefghijklmnopqrstuvwxyz";
cout << str1 << "\n";
cout << str2 << "\n";
if (!strcmp(str2, str1)) //          if (!strcmp(str1, str2) == 0)
  cout << "строки равны " << "\n";
else
  cout << "строки не равны " << "\n";
cout << "strcmp(str2, str1)= " << strcmp(str2, str1) << "\n\n";
```

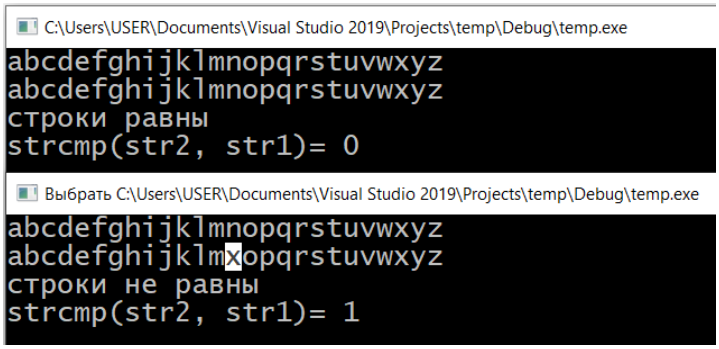

Рис. 71. Результат работы фрагмента программы

Длину строки можно найти с помощью функции `strlen()` (*Таблица 11*).

```
char str[100] = "absdefghij";
cout << strlen(str) << "\n";
cout << sizeof(str) << "\n\n";
```

Определите самостоятельно, в чем отличие подпрограммы `strlen()` и функции `sizeof()` для статических и динамических строк.

Поиск подстроки в строке

Подпрограмма `strstr()` (*Таблица 11*) предназначена для поиска подстроки в строке, как показано в примере ниже (*Листинг 89*) и на *Рис. 73*. Эта функция возвращает адрес той ячейки массива, с которой начинается подстрока и возвращает пустой указатель `NULL`, если такой подстроки нет.

Листинг 89

```
char stroka[] = "Мама мыла раму с мылом.";
char* pos = strstr(stroka, "мыло"); // поиск подстроки
if (pos == NULL)
  cout << "подстрока не найдена" << "\n";
else
  cout << "подстрока лежит по адресу " <<  (void*)pos << "\n";
cout << "pos= " << pos << "\n"; // найденная строка
cout << "stroka= " << stroka << "\n\n"; // исходная строка
// вывод всех символов строки с их адресами
for (int i=0; i<strlen(stroka); i++)
  cout << stroka[i] << " " << (void*)&stroka[i] << " ";
```

Результаты выполнения данного фрагмента программы приведены ниже на *Рис. 72*. В статической строке `stroka` ищется подстрока `"мыло"`. Можно видеть, что искомая подстрока была найдена по адресу `00D5F8E1`. Затем найденная подстрока, адрес которой сохраняется в указателе `pos` выводится на экран вместе с исходной строкой `stroka` (`00D5F8D0`).

И, для того, чтобы убедиться в правильности поиска все символы исходной строки выводятся на экран с их адресами. Единственное сомнение в правильности

поиска может быть связано с тем, что при выводе подстроки pos, на экран вывелось не "мыло", а "мылом.". Это объясняется тем, что подстрока pos – это указатель типа char*, т.е. тоже строка символов. А значит, она выводится на экран начиная с адреса 00D5F8E1 до первого встретившегося символа конца строки '\0'(00D5F8E6).

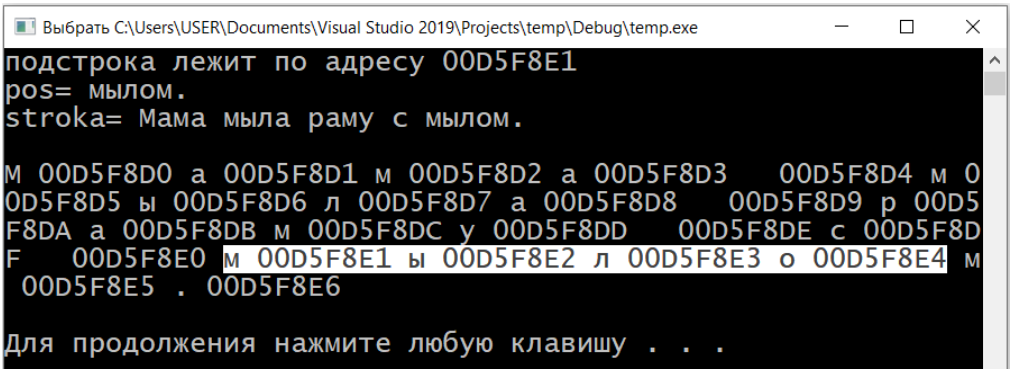

Рис. 72. Поиск подстроки в строке. Результат работы программы

Расположение в памяти строки stroka и указателя pos иллюстрирует схема на *Рис. 73*.

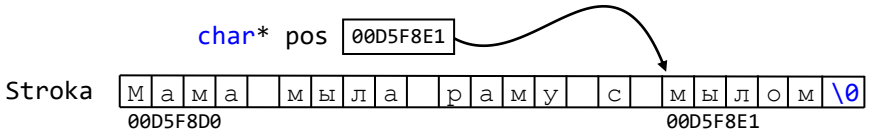

Рис. 73. Поиск подстроки в строке. Схема расположения в памяти

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

```
char* pos2 = (char*)calloc(strlen("мыло")+1, sizeof(char));
strncpy(pos2, pos, strlen("мыло"));
cout << "pos2= " << pos2 << "\n";
```

Таблица 11 иллюстрирует краткий список с встроенных функций языка C++, предназначенных для работы со строками, здесь же дается их краткое описание. *Таблица 12* содержит функции, предназначенные для перевода числовых констант различного типа в строковый вид и наоборот. Более полную информацию можно посмотреть в справочной системе и в литературе [*1-14*].

7.4. 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 11*). 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.

Приведенные в таблице подпрограммы присутствуют в том или ином виде в любом компиляторе языка C++. Однако, в зависимости от версии интерфейс (форма вызова) этих функций может отличаться. Например, вместо функции char* strcpy(char*, const char*) в 19 версии C++ применяется err_no strcpy_s(char*, size_t, char*).

Поэтому всегда нужно использовать справочную документацию или контекстный help, позволяющие проследить, какая версия отладочной среды используется и какие версии программ для работы со строками применяются.

Таблица 11 Функции для работы со строками

Функция	Описание
`strcat`	`char *strcat(char *dest, const char *src);` конкатенация (склеивание) строки `src` со строкой `dest`
`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, записать её в обратном порядке
`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.

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

Ввод строки с клавиатуры

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.

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.

```
char st[21];
gets_s(st, 21);
```

Пример: дублирование строки с выделением памяти:

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

Пример (*Листинг 90*) иллюстрирует действие функции `strtok()`, позволяющей разбивать строку на лексемы:

Листинг 90

```
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 91*), лексемы могут разделяться пробелом, запятой или горизонтальной табуляцией, а прием и синтаксический разбор строк завершается при вводе пустой строки.

Листинг 91

```
#include <iostream>
using namespace std;
// разбор строки на лексемы (предложения на слова)
int main()
{
	system("chcp 1251");
	//------------------------------------------------------------------------
--
	char* bufer = new char[100]; // для хранения текста - предложения с пробелами
	int i; // счетчик слов
```

```
        char* token; // указатель на лексему - "голова"
        char* token_in_bufer; // указатель на оставшуюся часть предложения - "хвост"
        char delimiter[] = "?! ,.\t"; // массив символов разделителей
        gets_s(bufer, 100); // ввод предложения с пробелами
        //std::cout << bufer << "\n";
        //-------------------------------------------------------------------------
--
        i = 0;
        token = strtok_s(bufer, delimiter, &token_in_bufer);
        while (token != NULL) // пока не кончилось предложение
        {
                if (token == NULL) break; // если слово пустое
                printf("%d - %s\n", i, token);
                //printf("Хвост %s\n", token_in_bufer);
                token = strtok_s(NULL, delimiter, &token_in_bufer);
                i++;
        }
        delete[]bufer;
}
```

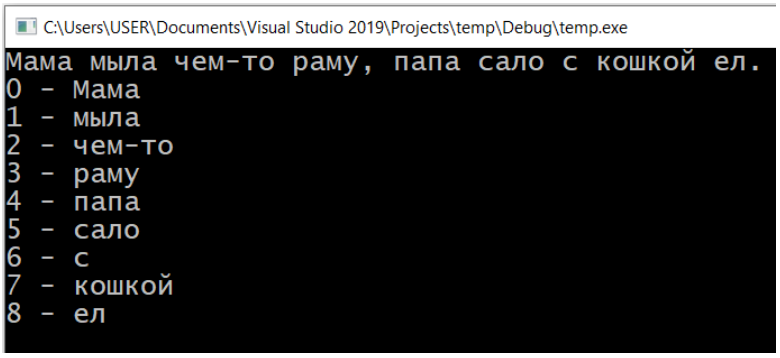

Рис. 74. Разделение строки на лексемы

7.5. Функции преобразования типов

Описанные ниже функции (*Таблица 12*) объявлены в стандартной библиотеке `stdlib.h`. Прототип функции `atof()` содержится, кроме того, в файле `math.h`.

Таблица 12 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.

Функция	Описание
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 92

```
#include <iostream>
using namespace std;
#define string2 "This is test string."
int _tmain(int argc, char* argv[])
{
	system("chcp 1251"); // 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;
}
```

```
Введите ваше имя: Sergey Greatest
Приятно познакомиться, уважаемый Sergey.

Введенная строка содержит 6 символов и занимает 50 ячеек памяти.
Тестовая строка: This is test string.
Тестовая строка содержит 20 символов и занимает 21 ячеек памяти.
```

Рис. 75. Отличие функций sizeof и strlen

Из примера (*Листинг 92*) и рисунка (*Рис. 75*) 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?

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?

8. 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 [*8, 13*]. *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.

8.1. 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("E:\\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.

Обратите внимание на константную строку, описывающую путь к файлу `"c:\\tmp\\test.txt"`. Можно видеть, что символ `'\\'` (слеш) в имени файла повторяется дважды. Это сделано для того, чтобы не путать символы строке – названии файла, стоящие перед символом «слэш» и служебные символы типа `'\n'`, `'\t'` и др.

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 4*).

```
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 93

```
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 )
```

Используя программу `fscanf()` нужно понимать, что она должна *возвращать* в вызывающую программу содержащиеся в файле данные. Естественно, обращение к ним из подпрограммы должно быть выполнено или «*по ссылке*» или «*по указателю*». Разработчиками C++ был выбран второй вариант – «*по указателю*». Поэтому при обращении к этой подпрограмме в неё нужно передавать `список_адресов_переменных`, в которые «по указателю» будут возвращаться считанные из файла данные.

Например, операцию ввода числа из текстового файла `F` в целочисленную переменную `n` правильно записать так:

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

Полезная функция `feof()` возвращает `1` (`true`), если файл, открытый для считывания, закончился. Она возвращает `0` (`false`), если из файла еще можно продолжать ввод.

Так как зачастую предварительно не известно, сколько данных содержит файл, то подпрограмму `feof()`можно применять для поиска конца файла, например, так:

Листинг 94

```
FILE* fi;
fi = fopen("test.txt", "rt"); // rt означает открытие файла для чтения
int n;
while (!feof(fi)) // пока не наступил конец файла
{ // прочитать из файла очередную переменную
  fscanf(fi, "%d", &n);
}
fclose(fi);
```

8.2. 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 *главе 12*, 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»* определяет принадлежность метода `open()` классу `fout`. Метод – это встроенная в класс подпрограмма; подробно о классах и их методах мы будем говорить в *главе 12*. Указанный в круглых скобках 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 13*).

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

Здесь для записи строки в файл используется оператор `fout <<` – передача операнда в поток `fout`. После чего файл закрывается при помощи метода `fout.close( )`.

В одной строке можно создать объект и сразу же связать его с файлом при создании объекта fout, как показано ниже:

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

Чтение данных из файла производится аналогично записи в файл, только используются при этом объекты класса вывода `ifstream`. Параметр режима открытия файла (*Таблица 13*) можно не указывать – он устанавливается по умолчанию.

Листинг 95

```
#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;
  system("PAUSE");
}
```

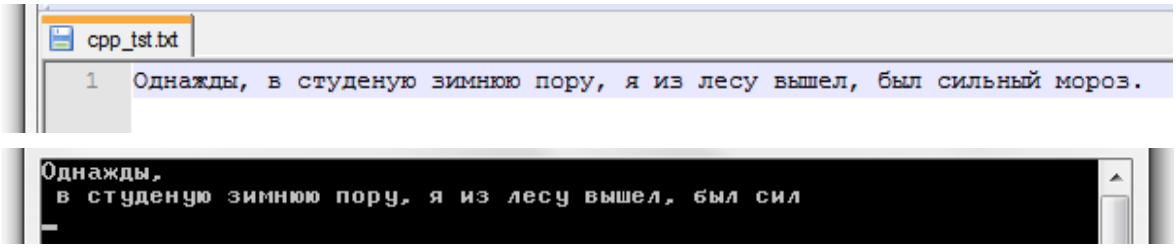

Рис. 76. Содержимое файла cpp_tst.txt (а) и результаты работы программы (б)

В программе (*Листинг 95*) показаны два способа чтения из файла, первый – используя операцию `fin >>` передачи в поток, второй – используя метод `fin.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. 76*), 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` (`true`) – если файл был успешно открыт, `0` (`false`) – если файл открыт не был (*Листинг 95*).

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 13*).

Таблица 13 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 96*) приведена функция производящая подсчет числа строк в текстовом файле.

Листинг 96

```
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;
}
```

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.

9. 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 в главе *5* мы познакомились с массивами – конструкцией из множества *однотипных* ячеек; строки, изученные в главе *7* – это частный случай массива. Настало время познакомиться с типом данных, позволяющим в одной переменной объединять ячейки памяти *разного типа*.

9.1. Структуры (struct)

В отличие от массива, все элементы которого однотипны, такой *тип данных* как *структура* (`struct`) может содержать элементы разных типов. Элементы структуры называются `полями` *структуры* и могут иметь любой тип, описанный ранее. Кроме типа этой же структуры (но могут быть указателями на него).

Программист сам конструирует, какие `поля` каких `типов` 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_переменных;
```

Если отсутствует `имя_структуры`, то должен быть указан `список_переменных`. В этом случае описание структуры служит определением элементов этого списка:

Листинг 97

```
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 97*) 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 полей перечисляют в фигурных скобках в порядке их описания:

Листинг 98

```
struct
{
  char fio[30];   // первое поле – символьная строка
  int date, code; // второе и третье поля – целые числа
  double salary;  // четвертое поле – вещественное число
} worker = { "Абрамов В. В.", 31, 215, 3400.55 };
```

В примере (*Листинг 98*) 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.

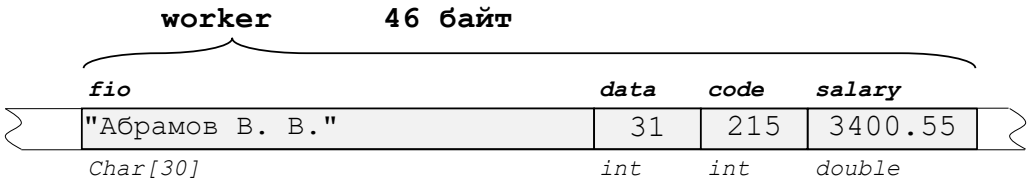

Рис. 77. Размещение переменной worker структурного типа в памяти

Как показано на рисунке (*Рис. 77*), 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 99*), тогда описание структуры определяет *новый тип данных*, имя которого можно использовать в дальнейшем наряду со стандартными типами, например:

Листинг 99

```
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 100

```
  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 101

```
  struct complex
  {
    float real, im;
  };
complex 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 побитовое копирование из одной области памяти в другую без «осмысления» того, как и что размещено в полях структуры. 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 102

```
struct A { int a; double x; };
struct B { A; double x; } x[2];
x[0].a.a = 1;
x[1].x = 0.1;
x[1].a.x = 0.01;
```

Как видно из примера (*Листинг 102*), поля разных структур могут иметь одинаковые имена, у них разная область видимости.

9.2. 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 103*).

Листинг 103

```
struct WorkerType
{
       char fio[30];
       int date, code;
       double salary;
};
WorkerType sw = { "Енишвили 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 = "Горбачус М.С.";
ps->code = 218; (*ps).salary = 0.01;
```

В этом примере `ps` – *указатель* на структуру типа `Worker`, тогда *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 `"*"`. Операторы доступа к полям структуры `"."` и `"->"` вместе с операторами вызова функции `()` и индексами массива `[]` занимают самое высокое положение в иерархии приоритетов операций в языке C++.

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 104*), 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 104*) описываются три подпрограммы для работы со структурой такого типа, реализация которых приводится ниже (*Листинг 104* - *Листинг 106*). Проанализировав главную программу `main()`, можно видеть основной алгоритм работы. Сперва выясняется количество строк в файле данных `weather.txt` при помощи программы, аналогичной *(Листинг 96*), 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 104

```
#include <iostream>
#include <fstream>
using namespace std;
// ----------------------------------------------------------------------------
enum TypeWD { N, NW, NE, W, E, S, SW, SE }; // направление 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;           // направление ветра
  long double WindPower;          // сила ветра
};
// ----------------------------------------------------------------------------
int Read(const char*, TypeWeather*);
void ReadWeather(ifstream&, TypeWeather&);
string StringWeather(TypeWeather&);
// ----------------------------------------------------------------------------
int main()
{
    system("chcp 1251");
    int size = string_count("weather.txt");
    TypeWeather* ArrayWeather = new TypeWeather[size];
    int rsize = Read("weather.txt", ArrayWeather);
    while (rsize)
    {
        cout << StringWeather(ArrayWeather[--rsize]) << endl;
    }
    system("pause");
}
```

Заполнение массива ArrayWeather осуществляется при помощи функции int Read(const char*, TypeWeather*), описанной в листинге (*Листинг 105*). Можно заметить, что параметрами этой функции выступает константная строка – имя файла и указатель TypeWeather*, смысл которого – передача адреса массива ArrayWeather в функцию «*по указателю*».

Листинг 105

```
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, подпрограмму `void ReadWeather(ifstream&, TypeWeather&)`, описанную в листинге (*Листинг 105*), 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 106

```
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(x.Date.Day) + "."
	+ to_string(x.Date.Month) + "." + to_string(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 (*Листинг 106*). В этой подпрограмме использован способ передачи объекта структурного типа в функцию *«по ссылке»*.

9.3. 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 (*Листинг 107*).

Листинг 107

```
  struct STR112
  {
    int a;
    char s[100]; // статическая строка
    // заготовка для хранения динамической строки - указатель на char
    char* d;
    char c;
};
```

Создадим новый тип данных – структуру для работы со строками переменной длины. Для хранения динамической строки `d` будем использовать указатель `char*`, статическая строка `s` пусть задается сразу в виде статического массива из 100 элементов типа `char`. Таким образом будет видна разница между использованием статических и динамических строк в составе переменной структурного типа.

Приведенный ниже листинг (*Листинг 108*) 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 108

```
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. 78*) 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.

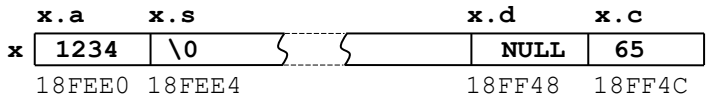

Рис. 78. Структура, содержащая в себе статическую строку

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 32-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 109*) 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 109

```
strcpy(x.s, "привет!"); // заполнение статической строки
// выделение памяти под динамическую строку
x.d = (char*)malloc(8 * 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 размера динамической строки `8*sizeof(char) = 8 байт`;

– выделение 8 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. 79*.

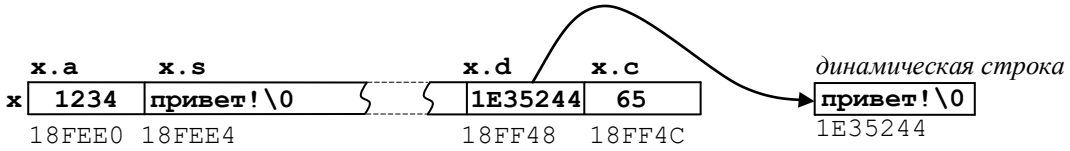

Рис. 79. Структура, содержащая в себе статическую и динамическую строки

Напишем подпрограмму для выделения памяти под динамическую строку структуры `STR112` и заполнения ее полей (*Листинг 110*). 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 110

```
// 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. 79*) 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 111*):

Листинг 111

```
// 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 113*) иллюстрирует создание *статической* переменной `x` структурного типа `STR112`. Результаты выполнения этой программы приведены на *Рис. 80*.

Листинг 112

```
STR112 x; // задана статическая переменная типа STR12
set_STR112(&x); // заполнение статической переменной
print_STR112(&x); // вывод на экран статической переменной
```

```
a= 0
s= static string s
d= dynamic string d
c= A

addresses:
addr_STR112= 0018FEE0
addr_a= 0018FEE0
addr_static_string_s= 0018FEE4
z->s= 0018FEE4
addr_dynamic_string_d= 0018FF48
z->d= 01F05244
addr_c= 0018FF4C

sizeof(s)= 100
sizeof(d)= 4
strlen(s)= 15
strlen(d)= 16
sizeof(STR112)= 112
```

Рис. 80. Размещение в памяти статической переменной типа `STR112`

Дополнив эту программу строками из листинга ниже (*Листинг 113*), 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. 81*).

Листинг 113

```
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
```

```
a= 0
s= static string s
d= dynamic string d
c= A

addresses:
addr_STR112= 01F05A80
addr_a= 01F05A80
addr_static_string_s= 01F05A84
z->s= 01F05A84
addr_dynamic_string_d= 01F05AE8
z->d= 01F05AF4
addr_c= 01F05AEC

sizeof(s)= 100
sizeof(d)= 4
strlen(s)= 15
strlen(d)= 16
sizeof(STR112)= 112
```

Рис. 81. Размещение в памяти динамической переменной типа `STR112`

По адресам, приведенным на *Рис. 81*, можно видеть, что в отличие от *Рис. 80*, динамическая переменная *y* размещается в динамической области памяти – *«куче»*. Однако принцип расположения полей не изменился, что естественно.

Пример

Задание: Создать динамический массив, состоящий из структур (`struct`), каждая из которых содержит по два динамических массива символов (`char*`). Написать подпрограммы запись такого массива в текстовый файл и чтение из него. Реализовать корректное освобождение памяти от динамических массивов.

Пусть массив структур содержит информацию о студентах. В составе структуры `struct Student` содержатся два указателя `char* FirstName` и `char* SecondName`, предназначенные для хранения адресов строк (*динамических массив символов*) – имени и фамилии студента соответственно.

Листинг 114

```
// структура для хранения информации о студенте
struct Student
{
  unsigned int ID;
  char *FirstName; // указатель на динамический массив символов
  char *SecondName; // указатель на динамический массив символов
  bool Gender;
  unsigned int YearOfBirth;
  unsigned int GroupID;
};
```

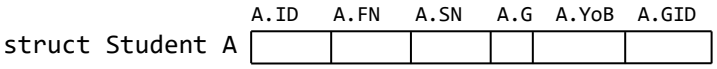

Рис. 82. Структура. Размещение в памяти

Листинг 115

```
// выделение памяти под динамические массивы символов
void SetStudent( Student* X,
                 unsigned int _ID,
                 const char* _FN,
                 const char* _SN,
                 bool _G,
                 unsigned int _YOB,
                 unsigned int _GID)
{
  X->ID = _ID;
  X->FirstName = new char[strlen(_FN) + 1];
  strcpy_s(X->FirstName, strlen(_FN) + 1, _FN);
  X->SecondName = new char[strlen(_SN) + 1];
  strcpy_s(X->SN, strlen(_SN) + 1, _SN);
  X->Gender = _G;
  X->YearOfBirth = _YOB;
  X->GroupID = _GID;
}
//---------------------------------------------------------------------------
// освобождение памяти из-под динамических массивов символов
void FreeStudent(Student* X)
{
  delete[](X->FirstName);
X->FirstName = nullptr;
  delete[](X->SecondName);
X->SecondName = nullptr;
}
```

Подпрограмма `SetStudent()` предназначена для заполнения полей динамической структурной переменной `Student* X`. Предполагается, что динамическая переменная `X` уже создана, а в подпрограмму передается её адрес.

В этой же подпрограмме выделяется память под динамические массивы символов. При выделении памяти под динамические массивы символов `char* FirstName` и `char* SecondName` у ОС запрашивается *минимальный* необходимый объем памяти. Размер ячейки вычисляется при помощи функции `strlen(_FN)+1`. Затем в выделенный динамический массив копируются символы константных строк `const char* _FN` и `const char* _SN` при помощи функции `strcpy_s()`.

Подпрограмма `FreeStudent()` освобождает память из-под динамических строк `X->FirstName` и `X->SecondName` структурной переменной `X`.

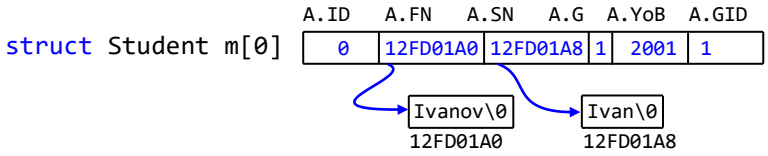

Рис. 83. Структура с двумя динамическими строками

Подпрограмма `PrintInLine()` производит вывод на экран полей массива `Student* XM` переменных структурного типа. В подпрограмму передается `XM` – адрес начального элемента массива (неважно, статического или динамического) и количество `int count` элементов в массиве.

Листинг 116

```
//-----------------------------------------------------------------------------
// вывод массива структур (Student) на экран
void PrintInLine(Student* XM, int count)
{
  std::cout << count << "\n"; // количество записей в массиве
  for (int i = 0; i < count; i++)
  {
    std::cout << XM[i].ID << " " << XM[i].FirstName << " " << XM[i].SecondName;
    if (XM[i].Gender)
      std::cout << ", male, ";
    else
      std::cout << ", female, ";
    std::cout << XM[i].YearOfBirth << " YoB, ";
    if (XM[i].GroupID == 1)
      std::cout << " TUSUR, 361-1 group.\n";
    if (XM[i].GroupID == 2)
      std::cout << " TUSUR, 361-2 group.\n";
    if (XM[i].GroupID == 3)
      std::cout << " TUSUR, 361-3 group.\n";
  }
}
```

Для работы с файлами созданы две подпрограммы `PrintInFile()` и `ReadFromFileInDynamicArray()`.

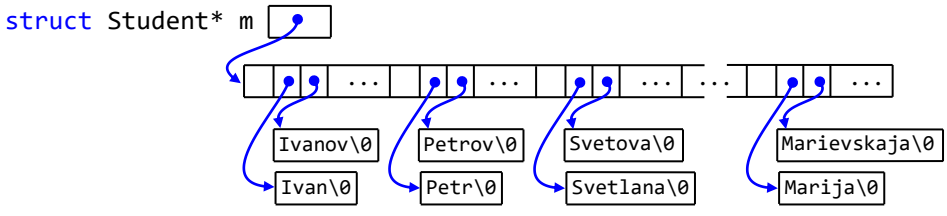

Рис. 84. Динамический массив, состоящий из структур, содержащих в себе по две строки (динамических массива) символов

Подпрограмма `PrintInFile()` открывает файл `const char* FileName`, проверяет, успешно ли осуществилось открытие. В первую строку файла заносится количество `count` элементов в массиве `Student* XM`. Затем в цикле построчно в файл записываются значения полей всех элементов `XM[i]`. Подпрограмма возвращает признак успешной (или неуспешной) записи в файл.

Листинг 117

```
//---------------------------------------------------------------------------
// Запись count элементов из массива структур (Student) в файл FileName
bool PrintInFile(Student* XM, int count, const char* FileName)
{
  std::ofstream fout(FileName, std::ios_base::out); // ios_base::app
  if (!fout.is_open()) // Проверка открытия файла
  {
    std::cout << "файл открыть не удалось\n";
    system("PAUSE");
    return 0; // флаг: ошибка открытия файла
  }
  fout << count << "\n"; // запись количества элементов в массиве
  for (int i = 0; i < count; i++) // запись полей структуры Student в файл
  {
    fout << XM[i].ID << "\n";
    fout << XM[i].FirstName << "\n";
    fout << XM[i].SecondName << "\n";
    fout << XM[i].Gender << "\n";
    fout << XM[i].YearOfBirth << "\n";
    fout << XM[i].GroupID << "\n";
  }
  fout.close(); // закрытие файла
  return 1; // флаг: данные записаны корректно
}
//---------------------------------------------------------------------------
// Чтение из файла и сохранение в динамическом массиве структур данных (Student)
Student* ReadFromFileInDynamicArray(const char* FileName, int& count, Student* DM)
{
  std::ifstream finp(FileName); // открыть файл
  if (!finp.is_open()) // проверка открытия файла
  {
    std::cout << "файл открыть не удалось\n";
    system("PAUSE");
    return nullptr;
  }
  finp >> count; // чтение количества записей в файле
  if ((count < 1) || (count > 100)) // ограничения на размер динамического массива
    return nullptr;
  DM = new Student[count]; // создание динамического массива структур
  char buf[100];
  for (int i = 0; i < count; i++) // заполнение массива структур из файла
  {
    finp >> DM[i].ID;
    finp >> buf;
    DM[i].FirstName = new char[strlen(buf) + 1];
    strcpy_s(DM[i].FirstName, strlen(buf) + 1, buf);
    finp >> buf;
    DM[i].SecondName = new char[strlen(buf) + 1];
    strcpy_s(DM[i].SecondName, strlen(buf) + 1, buf);
    finp >> DM[i].Gender;
    finp >> DM[i].YearOfBirth;
    finp >> DM[i].GroupID;
  }
  finp.close(); // закрытие файла
  return DM; // возврат адреса динамического массива структур
}
```

Подпрограмма `ReadFromFileInDynamicArray()` предназначена для чтения из файла `const char* FileName` динамического массива `Student* DM`, состоящего из элементов структурного типа `Student`. Важно: память под динамический массив `DM` выделяется внутри подпрограммы, в качестве параметра в нее передается пустой указатель `DM`, программа возвращает адрес выделенной области памяти.

Память под массив `DM` выделяется внутри подпрограммы чтения из файлов потому, что заранее не известно количество элементов этого массива. Прочтение первой строки из файла `FileName` заполняет ссылочную переменную `int& count`, определяющую количество записей в файле. После этого запрашивается память под `count` элементов динамического массива `DM` и организуется цикл, в котором производится чтение `count` записей из файла и заполнение `count` элементов динамического массива `DM`.

При заполнении каждого *i*-того элемента массива `DM[i]` создаются два динамических массива символов `DM[i].FirstName` и `DM[i].SecondName` под имя и фамилию студента. Эти массивы заполняются аналогично программе (*Листинг 114*).

Программа `Student* ReadFromFileInDynamicArray()` возвращает типизированный адрес выделенного динамического массива.

Листинг 118

```
//---------------------------------------------------------------------------
int main()
{
  system("chcp 1251"); // setlocale(LC_ALL, "Russian");
  Student *m = nullptr; // Указатель для динамического массива структур
  m = new Student[5];
  SetStudent(&m[0], 0, "Ivan", "Ivanov", 1, 2001, 1);
  SetStudent(&m[1], 1, "Petr", "Petrov", 1, 2000, 1);
  SetStudent(&m[2], 2, "Svetlana", "Svetlova", 0, 2001, 1);
  SetStudent(&m[3], 3, "Spiridon", "Spiridonov", 1, 2000, 2);
  SetStudent(&m[4], 4, "Marija", "Marievskaja", 0, 2001, 2);
  //
  // размер структуры без учета динамических массивов FirstName и SecondName
  std::cout << "sizeof(Student)= " << sizeof(Student) << "\n";
  //
  std::cout << "\nвывод динамического массива структур до записи в файл\n";
  PrintInLine(m, 5);
  //
  if (!PrintInFile(m, 5, "E:\\test.txt")) // если записать не удалось
    return 0;
  //
  for (int i = 0; i < 5; i++) // освобождение памяти
    FreeStudent(&m[i]);
  delete[]m;
  //---------------------------------------------------------------------------
  Student *dm = nullptr;
  int cnt = 0; // счетчик количества элементов в динамическом массиве структур
  dm = ReadFromFileInDynamicArray("E:\\test.txt", cnt, dm); // заполнение массива
  if (dm != nullptr) // если прочитать файл удалось
  {
    std::cout << "\nвывод динамического массива после его прочтения из файла\n";
    PrintInLine(dm, cnt); // вывод на экран прочитанного массива
    //
    for (int i = 0; i < 5; i++) // освобождение памяти
      FreeStudent(&dm[i]);
    delete[]dm;
  }
  system("PAUSE");
}
```

В программе `main()` создается и вручную заполняется динамический массив m, состоящий из `5` элементов типа `Student`. Оценивается размер каждого элемента массива – 24 байта (но при этом не учитываются размеры динамических строк символов `m[i].FirstName` и `m[i].SecondName` – память под них выделяется отдельно).

Элементы массива `m` выводятся на экран подпрограммой `PrintInLine(m, 5)` и в файл – подпрограммой `PrintInFile(m, 5, "E:\\test.txt")`.

Затем производится освобождение памяти и удаление массива `m`. При этом для каждого элемента массива вызывается подпрограмма `FreeStudent(&m[i])`, корректно освобождающая память из-под динамических строк `m[i].FirstName` и `m[i].SecondName`. И только после этого удаляется сам массив `m`.

Во второй части программы производится чтение динамического массива `dm` из файла при помощи подпрограммы `ReadFromFileInDynamicArray("E:\\test.txt", cnt, dm)`. Поскольку заранее не известно число записей в файле, то используется переменная `int cnt` для хранения размера массива. Затем прочитанный массив `dm` выводится на экран после чего корректно уничтожается.

```
C:\Users\USER\Documents\Visual Studio 2019\Projects\test_20220312_08_02\Debug\test_20220312_08_02.exe
sizeof(Student)= 24

вывод динамического массива структур до записи в файл
5
0 Ivan Ivanov, male, 2001 YoB,  TUSUR, 361-1 group.
1 Petr Petrov, male, 2000 YoB,  TUSUR, 361-1 group.
2 Svetlana Svetlova, female, 2001 YoB,  TUSUR, 361-1 group.
3 Spiridon Spiridonov, male, 2000 YoB,  TUSUR, 361-2 group.
4 Marija Marievskaja, female, 2001 YoB,  TUSUR, 361-2 group.

вывод динамического массива структур после его прочтения из файла
5
0 Ivan Ivanov, male, 2001 YoB,  TUSUR, 361-1 group.
1 Petr Petrov, male, 2000 YoB,  TUSUR, 361-1 group.
2 Svetlana Svetlova, female, 2001 YoB,  TUSUR, 361-1 group.
3 Spiridon Spiridonov, male, 2000 YoB,  TUSUR, 361-2 group.
4 Marija Marievskaja, female, 2001 YoB,  TUSUR, 361-2 group.
Для продолжения нажмите любую клавишу . . .
```

Рис. 85. Динамический массив, состоящий из структур, содержащих в себе по два динамических массива символов (строки). Результаты работы программы – запись в файл и чтение из него

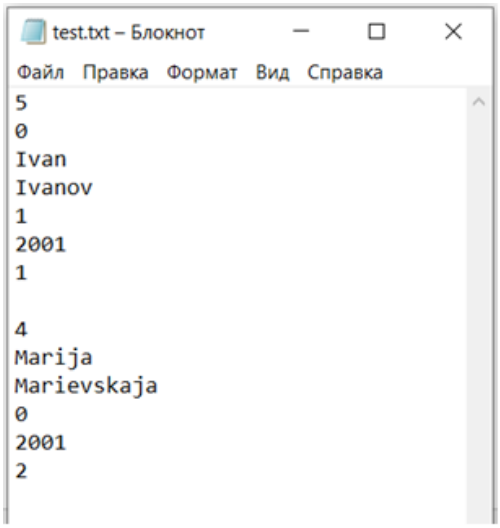

Рис. 86. Сохранение массива структур в текстовом файле

На рисунке ниже (*Рис. 86*) изображен текстовый файл `"E:\test.txt"`, в который производится запись элементов массива `m` и из которого производится чтение структур – элементов массива `dm`.

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?

6. Что такое указатель на структуру, как с ним работать, как обращаться к полям структурной переменной «по адресу?»

8. Как осуществляется обращение к полям динамической переменной структурного типа?

9. Как передается в подпрограмму статическая переменная структурного типа?

10. Специальные структурные типы данных

10.1. 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 119*.

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. 87*) переменная MyOption дополнена до 2 байт.

Листинг 119

```
#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;
}
```

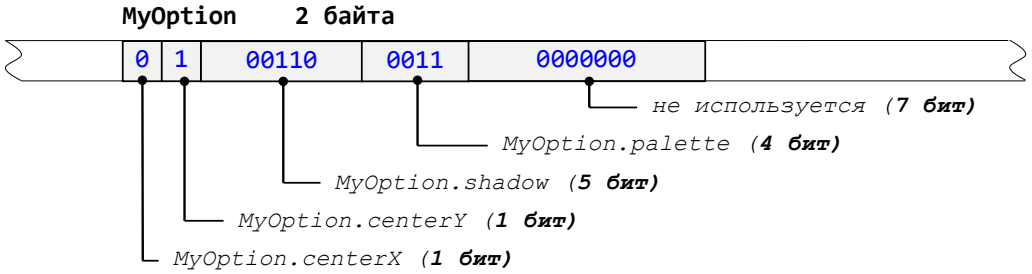

Рис. 87. Размещение переменной MyOption структурного типа Options в памяти

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 120

```
int main()
{
  system("chcp 1251");
  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);
  system("pause");
}
```

Важно знать, что компилятор не отслеживает выхода за границы битового поля – ответственность за это полностью лежит на программисте. Например, в задаче (*Листинг 119*) 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!
```

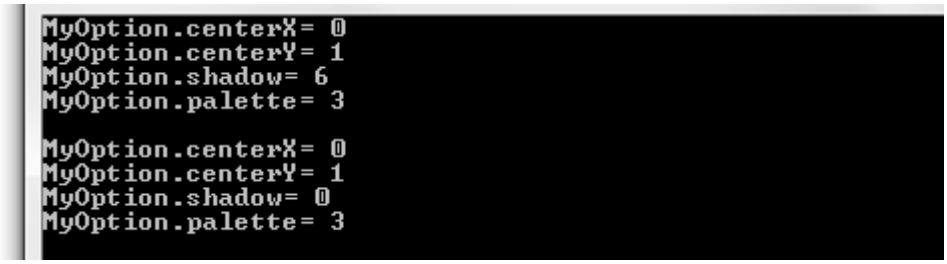

Рис. 88. Переполнение разряда 5-битного поля MyOption.shadow

Переполнение разрядов в полях битовой карты

Рассмотрим пример, который иллюстрирует работу с битовыми полями при переполнении каких-то полей битовой карты. Для этого в программе (*Листинг 121*) задан новый тип данных – битовая карта `BitMapType` с полями различной длины. Подпрограмма `show(BitMapType&)`, получающая ссылку на переменную типа `BitMapType`, предназначена для вывода на экран размера переменной, её адреса и значений, хранящихся в полях ссылочной переменной.

Листинг 121

```
#include <iostream>
using namespace std;
//---------------------------------------------------------------------------
struct BitMapType
{
  unsigned short int a : 3;  // только 3 бита
  unsigned short int b : 2;  // только 2 бита
  unsigned short int c : 1;  // только 1 бит
  unsigned short int d : 4;  // только 4 бита
  unsigned short int e : 6;  // только 6 битов
};
//---------------------------------------------------------------------------
void show(BitMapType& Z)
{
  cout << "размер Z= " << sizeof(Z) << " байта, адрес Z= " << &Z << "\n";
  cout << "Z.a= " << Z.a << "  Z.b= " << Z.b << "  Z.c= " << Z.c << "  Z.d= "
  << Z.d << "  Z.e= " << Z.e << "\n\n";
}
//---------------------------------------------------------------------------
int main()
{
  system("chcp 1251");
  unsigned short int temp = 0;
  cin >> temp;
  BitMapType Z1;
  Z1.a = temp; // 0..7
```

```
  Z1.b = temp; // 0..3
  Z1.c = temp; // 0 или 1
  Z1.d = temp; // 0..15
  Z1.e = temp; // 0..63
  show(Z1);
  system("pause");
}
```

В теле программы `main()` задается переменная `unsigned short int temp`, её значение вводится с клавиатуры. Кроме того, задается переменная `Z1` исследуемого типа `BitMapType`. После делается попытка присвоить полям переменной `Z1` значения переменной `temp`. Переменная `temp` занимает 2 байта в ОЗУ, а размеры поля переменной `Z1` ограничены в соответствии с форматом битовой карты `BitMapType`. Поэтому обязательно произойдет переполнение каких-то её полей. Будем задавать различные значения переменной `temp` и проследим, как происходит переполнение полей. Какие при этом значения принимают соответствующие поля и влияет ли переполнение на биты соседних полей.

Поле `Z1.a` занимает 3 бита, поэтому может принимать значения от 000_2 до 111_2, т.е. от 0_{10} до 7_{10}. Поле `Z1.b` занимает 2 бита, поэтому может принимать значения от 00_2 до 11_2, т.е. от 0_{10} до 3_{10}. Поле `Z1.c` занимает 1 бит, поэтому может принимать значения только `0` или `1`. Поле `Z1.d` занимает 4 бита, поэтому может принимать значения от 0000_2 до 1111_2, т.е. от 0_{10} до 15_{10}. Поле `Z1.e` занимает 6 битов, поэтому может принимать значения от 000000_2 до 111111_2, т.е. от 0_{10} до 63_{10}.

На рисунках ниже приводятся результаты работы программы (*Листинг 121*) для различных значений введенной переменной `temp` и схемы расположения двоичного кода в полях переменной `Z1` после присваивания.

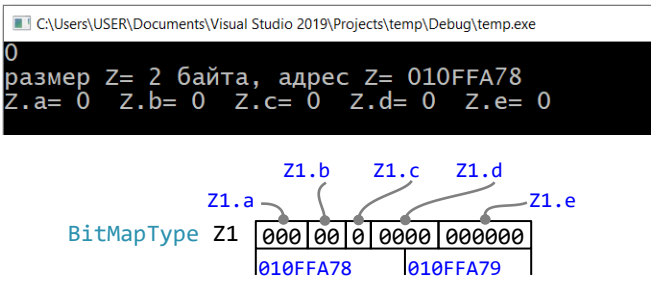

Рис. 89. Состояние битов структуры `BitMapType` *при* `temp=0`

Из схемы (*Рис. 89*) видно, что переменная `Z1` занимает в памяти 2 байта и её поля обнуляются корректно.

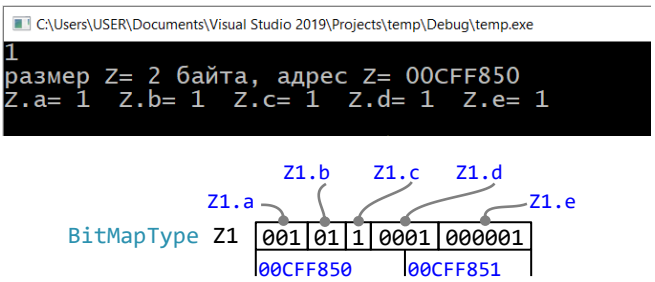

Рис. 90. Состояние битов структуры `BitMapType` *при* `temp=1`

При `temp==1` схема (*Рис. 90*) показывает, как в полях переменной `Z1` размещается двоичный код. Видно, что границы байтов внутри битовой карты не играют роли: в частности, поле `Z1.d` располагается в обоих байтах. Но это не мешает ему быть корректно заполненным. Пока никаких переполнений нет.

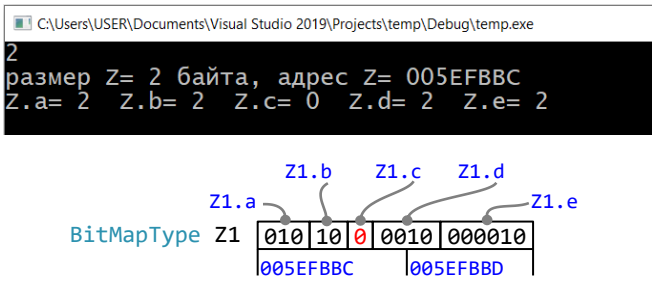

Рис. 91. Состояние битов структуры `BitMapType` *при* `temp=2`

При `temp == 2` (*Рис. 91*) появилось переполнение поля `Z1.c`. В программе этому полю присваивается `2`, а в нем оказался `0`. Как это произошло? Схема расположения байтов в полях переменной `Z1` поясняет это. Верхний бит числа 2_{10} `==` 01_2 не может разместиться в однобитовом поле `Z1.c` и отбрасывается, «обрезается». Обратите внимание, что число `01` именно обрезается, а не переходит в соседний разряд! Это потому, что соседний разряд принадлежит уже другому полю – `Z1.b` и недоступен для поля `Z1.c`. При выводе на экран видно, что значения соседних полей `Z1.b` и `Z1.d` введены корректно. Значит, никакого перехода через разряд в соседнее поле битовой карты не происходит.

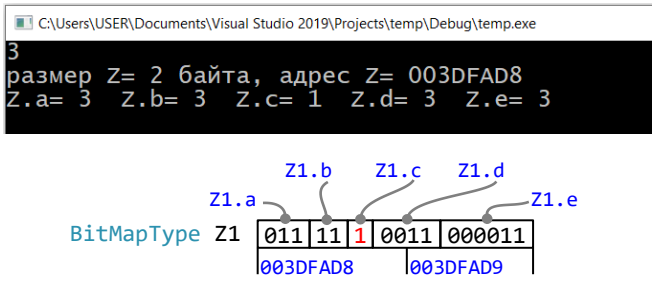

Рис. 92. Состояние битов структуры `BitMapType` *при* `temp=3`

Выводимые на экран данные и схема (*Рис. 92*) показывают при `temp == 3` как происходит отбрасывание не поместившихся в разряд битов. Отбрасываются старшие биты. В этом можно убедиться, если обратить внимание на переполняемое поле `Z1.c`. Этому полю присваивалась `3`, а в нем оказалась `1`. Это, очевидно, единица младшего разряда. Остальные поля присвоены корректно.

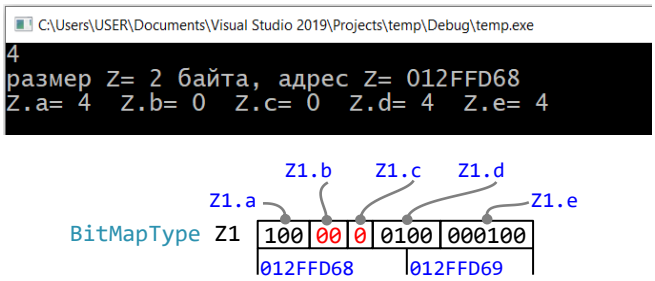

Рис. 93. Состояние битов структуры `BitMapType` *при* `temp=4`

А вот при `temp == 4` переполненными оказываются уже 2 поля – `Z1.c` и `Z1.b`. Так как 4_{10} `==` 100_2 , то двух битов поля `Z1.b`. уже не хватает для размещения в нем значения переменной `temp`. Как и для `Z1.c`. Не поместившиеся биты отбрасываются начиная со старших разрядов. Поэтому в полях `Z1.b` и `Z1.c` лежат `00` и `0` соответственно Остальные поля имеют размер больше двух битов, поэтому присвоены корректно (*Рис. 93*).

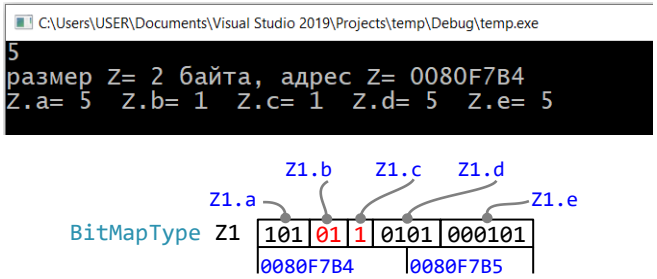

Рис. 94. Состояние битов структуры `BitMapType` *при* `temp=5`

Итак, по индукции можно сделать вывод о том, как происходит присваивание полей битовой карты при их переполнении:

- во-первых, не поместившиеся биты отбрасываются, начиная с верхних разрядов;
- во-вторых, переполнение одних полей битовой карты не влияет на остальные её поля.

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 и времени работы программы.

10.2. 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 122

```
#include <iostream>
//---------------------------------------------------------------------------
enum paytype { CARD, CHECK };
union payment
{
  char card[20];
  long check;
};
//---------------------------------------------------------------------------
int main()
{
  system("chcp 1251");
  payment info;
  paytype ptype = CARD;
  switch (ptype) // с каким полем объединения info работаем?
  {
  case CARD:
    strcpy_s(info.card, 20, "1122 2234 3201 0030");
    cout << "Оплата по карте: " << info.card << "\n"; break;
  case CHECK:
    info.check = 123456;
    cout << "Оплата чеком: " << info.check << "\n"; break;
  }
}
```

Объединение часто используют в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Например, в программе (*Листинг 122*) переменная `ptype` типа `enum paytype` используется как раз для того, чтобы определять, в каком случае переменная-объединение `info` типа `union payment` используется как поле `info.card`, а в каком случае – как `info.check`.

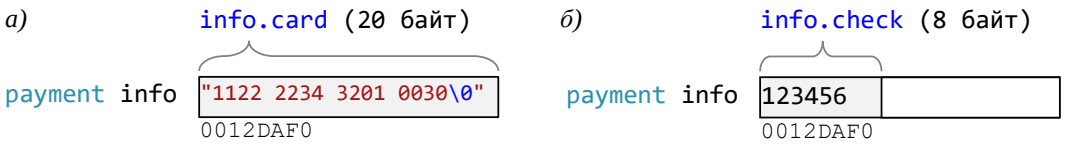

Рис. 95. Размещение переменной `info` в памяти:
а) при обращении к полю `info.card`; б) при обращении к полю `info.check`

Если все поля объединения хранятся по одному адресу, то как же данные, хранящиеся в полях объединения, не смешиваются? А они ***смешиваются***! При изменении значения поля `info.check` у переменной `info` изменится и поле `info.card` и наоборот. Потому что это одна и та же ячейка памяти, в нашем примере – с адресом `0012DAF0`.

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 123*) рассмотрим работу со структурой, содержащей битовые поля:

Листинг 123

```
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;
  } bits; // 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 (*битовая карта*) `bits`. Поле `bits` содержит 8 однобитовых полей `bits.b0 – bits.b7`.

Листинг 124

```
void show(uByteType* B)
{
  printf("B.byte= %d B.bits= (%d%d%d%d%d%d%d%d)", B->byte, B->bits.b7, B->bits.b6,
    B->bits.b5, B->bits.b4, B->bits.b3, B->bits.b2, B->bits.b1, B->bits.b0);
  printf("\n&B= %p &B.byte= %p &B.bit.b0= %p\n\n", B, &(B->byte), &(B->bits));
}
```

На примере программы `void show(uByteType*)` удобно рассмотреть гибкость полученной конструкции. Когда нам удобно обращаться к переменной типа `uByteType` как к десятичному числу или символу, мы будем писать `B->byte` и данная однобайтовая конструкция будет интерпретироваться как `unsigned char`. Если же нам требуется

получить доступ к конкретному биту этой ячейки памяти, то мы воспользуемся побитовым обращением `B->bit.b7`, ..., `B->bit.b0`.

Листинг 125

```
int main()
{
  system("chcp 1251");
    uByteType B;
    B.byte = 204; // обращение к числу
    show(&B);
    B.bits.b3 = 0; // обращение к 3му биту
    show(&B);
    B.byte = 128; // обращение к числу
    show(&B);
  system("pause");
}
```

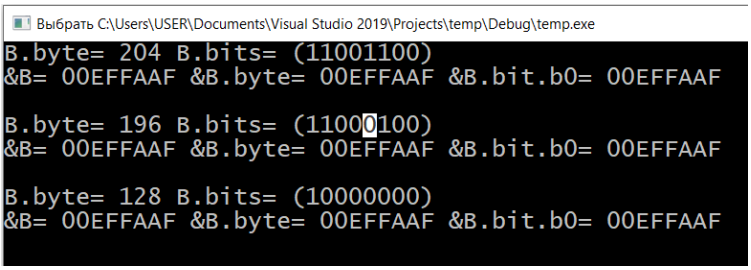

Рис. 96. Размещение в памяти переменной типа union

Можно видеть (*Рис. 96*) 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. 97*) иллюстрирует расположение ячейки `B` типа `uByteType` в памяти. В листинге (*Листинг 124*) показывается, что можно обращаться ко всему байту целиком: `B.byte=204`, при изменении отдельного бита `B.bit.b3=0`, изменяется весь байт и становится равным `196`.

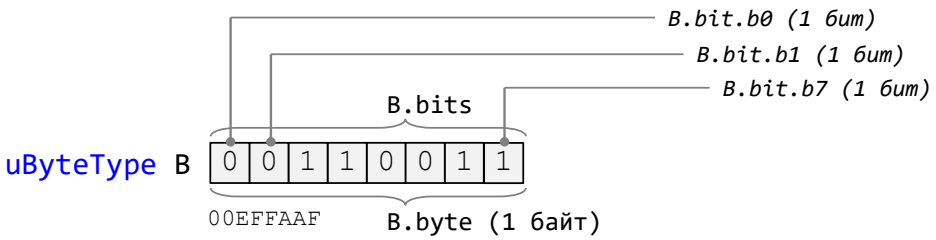

Рис. 97. Размещение в памяти переменной типа union

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. 12*);
- объединение не может входить в иерархию классов (*см. гл. 12*).

Контрольные вопросы:

1. Что такое объединение (`union`), как размещаются поля в переменных такого типа данных?

2. Как размещаются в переменной объединения (`union`) поля структуры?

3. Можно ли указывать, сколько бит выделять под каждое поле структуры?

4. Как происходит заполнение полей структуры битовая карта (`bitmap`) при переполнении какого-то из полей?

5. В чем отличия при обращении к полям статической и динамической переменных типа битовая карта (`bitmap`)?

6. Чем отличается обращение к полям переменной структурного типа (`struct`) и к полям переменной объединения (`union`)?

7. Отличается ли обращение к полям статической и динамической переменных типа объединение (`union`)?

8. Какие адреса у полей переменных типа объединение (`union`)?

11. Операции с разрядами

Программистов на похоронах выносят
младшим байтом вперёд

В организации оперативной памяти всех вычислительных машин – начиная от первых 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».

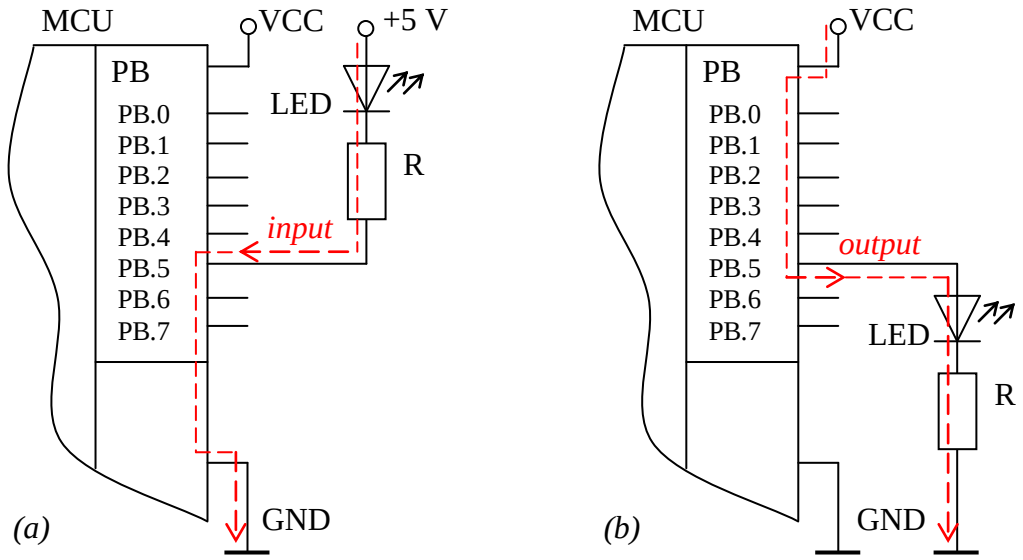

Рис. 98. Варианты схем принципиальных подключения светодиодного индикатора к микроконтроллеру

На *Рис. 98* 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 126

```
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 126*) 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. 98 (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`. Фактически, микроконтроллер включает и выключает все 8 выходов порта `PB`. Однако, для случая, когда необходимо изменить, например, *только пятый разряд* регистров `DDRB` и `PORTB`, не изменяя остальные разряды, не существует операции 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. 11.1, 11.2*), выполняющихся над битами байта. Во-вторых, конструируемая форма данных, объединяющая *битовое поле* в *объединение* (*union*), дающая доступ к разрядам переменной (*см. п. 10.1, 10.2*).

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 $0x19_{16}$.

11.1. Поразрядные логические операции

Четыре операции производят действия над данными, относящимися ко всем целым типам данных, включая `unsigned char`. Они называются «*поразрядными*» потому что выполняются отдельно над каждым разрядом *независимо*.

Поразрядное отрицание

Поразрядное отрицание или дополнение до единицы – это унарная операция изменяет побитно каждую `1` на `0`, а `0` на `1`. Обозначается значком «*тильда*» `"~"`.

```
~(10011010₂) == 01100101₂
```

Поразрядное И

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен `1(true)`, если только оба соответствующих разряда операндов равны `1(true)`. Результат получается истинным, если только каждый из двух одноразрядных операндов является истинным. Обозначается символом «*амперсент*» `"&"`.

```
10010011₂ & 00000100₂ == 00000000₂
10010011₂ & 00000001₂ == 00000001₂
10010011₂ & 00111101₂ == 00010001₂
```

Поразрядное ИЛИ

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен `1(true)`, если любой из соответствующих разрядов операндов равен `1(true)`. Результат получается истинным, если один из двух (или оба) одноразрядных операндов является истинным. Обозначается символом «*вертикальная черта*» `"|"`.

```
10010011₂ | 11111111₂ == 11111111₂
10010011₂ | 00000000₂ == 10010011₂
10010011₂ | 00111101₂ == 10111111₂
```

Поразрядное исключающее ИЛИ

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен `1(true)`, если соответствующие разряды операндов не совпали и `0(false)` – если совпали. Результат получается истинным, если один из двух (но не оба) одноразрядных операнда является истинным. Обозначается символом «*стрелка вверх*» `"^"`.

```
10010011₂ ^ 00000000₂ == 10010011₂
10010011₂ ^ 11111111₂ == 01101100₂
10010011₂ ^ 00111101₂ == 10101110₂
```

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 создали макроопределение `mask` в директиве `#define` равным `2`, т. е. двоичному значению 00000010_2, имеющему ненулевое значение только в первом разряде.

```
#define MASK 2 // 00000010 // выделяем для операции первый разряд
```

Тогда оператор «Поразрядное И»

```
flags = flags & MASK; // обнуляет все биты кроме первого, первый не меняется
```

установит все разряды байта `flags` (кроме первого) в `0`, потому что `A&0 == 0` для любого `A`. Первый разряд останется неизменным, так как `A&1 ==A` для любого `A`.

Аналогично оператор «Поразрядное ИЛИ»

```
flags = flags | MASK; // первый разряд делает 1, остальные не меняются
```

установит первый разряд в `1`, потому что `A|1 ==1` для любого `A`. Остальные разряды останутся неизменными, так как `A&1 ==A` для любого `A`.

Оператор «Поразрядное исключающее ИЛИ» в комбинации с маской он инвертирует первый разряды байта `flag`, и оставит все остальные биты неизменными.

```
flags = flags ^ MASK; // инвертирует первый разряд, остальные не меняются
```

Комбинируя описанные выше битовые (поразрядные) операции и маски различного вида можно работать непосредственно с теми разрядами, которые необходимы, не изменяя остальные биты в байте. Или наоборот, изменяя как нужно все разряды кроме выделенных маской.

Рассмотрим пример (*Листинг 127*). В нем, задавая маску 00000110_2, выделяющую два бита – первый и второй, решается ряд задач по преобразованию *выделенной группы битов*. При этом остальные биты однобайтовой переменной остаются неизменными.

```
                                                                    Листинг 127
#define MASK 6 // 00000110 // выделяем 1 и 2 разряды
//---------------------------------------------------------------------------
```

```
int main()
{
  unsigned char flags = 0;
  cout << "MASK\t"; show_byte(MASK);
  cout << "~MASK\t"; show_byte(~MASK); //----------------------------------------
  cout << "задача: инвертировать разряды, выделенные маской, не меняя остальные\n";
  flags = 0;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags ^ MASK);
  flags = 255;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags ^ MASK);
  flags = 170;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags ^ MASK); //---------------------------------
  cout << "задача: обнулить разряды, выделенные маской, не меняя остальные\n";
  flags = 0;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags & (~MASK));
  flags = 255;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags & (~MASK));
  flags = 170;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags & (~MASK)); //-------------------------------
  cout << "задача: сделать 1 разряды, выделенные маской, не меняя остальные\n";
  flags = 0;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags | MASK);
  flags = 255;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags | MASK);
  flags = 170;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags | MASK);
}
```

Не обращайте пока внимания на подпрограмму `show_byte()`, она рассматривается в следующем параграфе (*Листинг 131*) и предназначена для вывода на экран однобайтовой переменной в десятичном и двоичном форматах.

```
C:\Users\USER\Documents\Visual Studio 2019\Projects\temp\Debug\temp.exe
MASK    6       00000110
~MASK   249     11111001
задача: инвертировать разряды, выделенные маской, не меняя остальные
 было= 0        00000000
 стало= 6       00000110
 было= 255      11111111
 стало= 249     11111001
 было= 170      10101010
 стало= 172     10101100
задача: обнулить разряды, выделенные маской, не меняя остальные
 было= 0        00000000
 стало= 0       00000000
 было= 255      11111111
 стало= 249     11111001
 было= 170      10101010
 стало= 168     10101000
задача: сделать единичками разряды, выделенные маской, не меняя остальные
 было= 0        00000000
 стало= 6       00000110
 было= 255      11111111
 стало= 255     11111111
 было= 170      10101010
 стало= 174     10101110
```

Рис. 99. Результаты выполнения программы по преобразованию битов при помощи маски

Рассмотрим альтернативную программу (*Листинг 128*). При той же маске 00000110_2 решается ряд задач по преобразованию всех разрядов однобайтного числа *кроме выделенной группы битов*.

Листинг 128

```
#define MASK 6 // 00000110 // выделяем 1 и 2 разряды
//---------------------------------------------------------------------------
int main()
{
  unsigned char flags = 0;
  cout << "MASK\t"; show_byte(MASK);
  cout << "~MASK\t"; show_byte(~MASK); //-----------------------------------------
  cout << "задача: не меняя разряды, выделенные маской, сделать 1 остальные\n";
  flags = 0;
  cout << " было= \t";  show_byte(flags);
  cout << " стало= ";  show_byte(flags | (~MASK));
  flags = 255;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags | (~MASK));
  flags = 170;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags | (~MASK)); //-----------------------------
  cout << "задача: не меняя разряды, выделенные маской, обнулить остальные\n";
  flags = 0;
  cout << " было= \t";  show_byte(flags);
  cout << " стало= ";  show_byte(flags & MASK);
  flags = 255;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags & MASK);
  flags = 170;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags & MASK); //---------------------------------
  cout << "задача: не меняя разряды, выделенные маской, инвертировать остальные\n";
  flags = 0;
  cout << " было= \t";  show_byte(flags);
  cout << " стало= ";  show_byte(flags ^ (~MASK));
  flags = 255;
  cout << " было= ";  show_byte(flags);
  cout << " стало= ";  show_byte(flags ^ (~MASK));
}
```

```
C:\Users\USER\Documents\Visual Studio 2019\Projects\temp\Debug\temp.exe
MASK    6       00000110
~MASK   249     11111001
задача: не меняя разряды, выделенные маской, сделать единичками остальные
 было=  0       00000000
 стало= 249     11111001
 было= 255      11111111
 стало= 255     11111111
 было= 170      10101010
 стало= 251     11111011
задача: не меняя разряды, выделенные маской, обнулить остальные
 было=  0       00000000
 стало= 0       00000000
 было= 255      11111111
 стало= 6       00000110
 было= 170      10101010
 стало= 2       00000010
задача: не меняя разряды, выделенные маской, инвертировать остальные
 было=  0       00000000
 стало= 249     11111001
 было= 255      11111111
 стало= 6       00000110
 было= 170      10101010
 стало= 83      01010011
```

Рис. 100. Результаты выполнения программы по преобразованию битов

11.2. Поразрядные операции сдвига

Операторы, которые мы используем для ввода и вывода данных на консоль – операторы сдвига "<<" и ">>" на самом деле изначально создавались не для этого, а для *побитовых операций сдвига*. Эти операции сдвигают разряды влево или вправо. Рассмотрим их более подробно.

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 (отбрасываются).

$$10001010_2 << 2 == 00101000_2$$

Сдвиг вправо

Эта операция ">>" сдвигает разряды левого операнда вправо на число позиций, указанное правым операндом. Разряды, сдвигаемые за правый предел левого операнда, теряются (отбрасываются). 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).

$$10001010_2 >> 2 == 00100010_2$$

Операции выполняют сдвиг влево или вправо, что также является эффективным способом умножения и деления на степени 2 (*Листинг 129*). Но, разумеется, для целых, беззнаковых чисел.

Листинг 129

```
unsigned char number;
number = 7;
show_byte(number);
show_byte(number << 3); // умножает number на 8 (на 2 в третьей степени)
number= 39;
show_byte(number);
show_byte(number >> 2); // делит number нацело на 4 (на 2 в квадрате)
```

Это действие аналогично соответствующему алгоритму для десятичной системы счисления, обеспечивающему сдвиг десятичной точки при умножении или делении на степень числа 10. А для двоичной системы счисления – на степень числа 2.

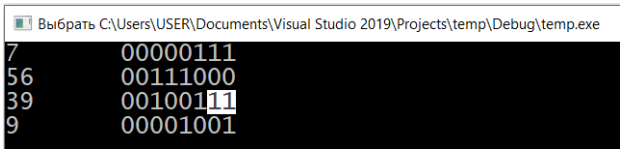

Рис. 101. Результаты выполнения операций сдвига

Очевидно, что сдвинутые вправо и отброшенные биты (аналог целочисленного деления на 2 в степени), на *Рис. 101* они выделены белым цветом, представляют собой *остаток от деления*. Видимо, так и реализована операция "%".

В листинге, приведенном ниже (*Листинг 130*), показано, как при помощи операции сдвига организовать вывод *разрядов* – байтов, содержащих единицу только в одном соответствующем разряде. На *Рис. 102* отражен результат – в цикле выводятся степени числа 2, как результат сдвига единицы в i-*тую* позицию: (1 << i). Фактически, это просто единица в соответствующем разряде: 10000000_2, 01000000_2, ..., 00000001_2.

Листинг 130

```
  for (int i = 7; i >= 0; i--)
    cout << (1 << i) << " ";
```

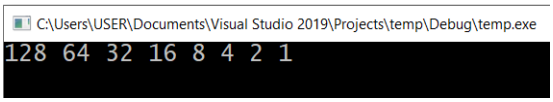

Рис. 102. Результаты выполнения операций сдвига единицы влево

Сравнивая побитно, используя `поразрядное И`, единицу соответствующего разряда `(1<<i)` и бит этого же разряда из произвольной переменной `b` (как в подпрограмме ниже) можно определить, какой бит переменной `b` размещен в данном разряде. И вывести на экран значение переменной в двоичном формате (*Листинг 131*):

Листинг 131

```
void show_byte(unsigned char b) // 1 байт
{
  printf("\n%d ", b);
  for (int i = 7; i >= 0; i--)
    if (b & (1 << i)) cout << "1";
    else cout << "0";
}
```

Иными словами, в этом листинге (*Листинг 131*) каждый разряд числа `b` сравнивается `b & (1<<i)` 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 132*):

Листинг 132

```
void show_2byte(short int b) // 2 байта
{
  cout << b << "\n";
  for (int i = 15; i >= 0; i--)
    if (b & (1 << i)) cout << "1";
    else cout << "0";
}
```

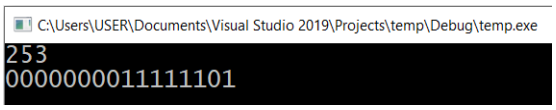

Рис. 103. Вывод на экран двухбайтного числа в десятичном и двоичном форматах

11.3. Обращение к разрядам при помощи битовых полей

Второй способ манипуляции разрядами заключается в использовании *битовых полей и объединений* – то, что мы изучали в *Главе 10*. Повторим: 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 структурной переменной `prnt` типа `BitMap` можно использовать для оперирования значениями отдельных битовых полей:

```
  cout << "size= " << sizeof(prnt) << "byte" << endl;
  prnt.b0 = 0;
  prnt.b3 = 1;
  prnt.b4 = false;
  prnt.b5 = true;
  cout << prnt.b5 << endl;
```

Поскольку каждое поле состоит только из одного разряда, мы можем использовать для присваивания лишь значение `0(false)` или `1(true)`. Переменная `prnt` размещается в ячейке памяти, имеющей размер 1 байт.

Теперь для интерпретации данных зададим 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; // для оперирование с байтом
    BitMap bm; // для обращения к битам
  };
```

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`.

Листинг 133

```
int main()
{
  system("chcp 1251");
    uBitMap X;
    X.ch = 'A';
    printf("X= %c\n", X.ch);
    cout << "sizeof(X)= " << sizeof(X) << " byte" << "\n"; // размер
    printf("в формате char X= %c\n", X.ch);
    printf("в шестнадцатеричном формате X= %X\n", X.ch);
    printf("в десятичном формате X= %d\n", X.ch);
    printf("в восьмеричном формате X= %o\n", X.ch);
    cout << "в двоичном формате X= " << X.bm.b7 << X.bm.b6 << X.bm.b5 << X.bm.b4
                                << X.bm.b3 << X.bm.b2 << X.bm.b1 << X.bm.b0 << "\n";
}
```

```
C:\Users\USER\Documents\Visual Studio 2019\Projects\temp\Debug\temp.exe
X= A
sizeof(X)= 1 byte
в формате char X= A
в шестнадцатеричном формате X= 41
в десятичном формате X= 65
в восьмеричном формате X= 101
в двоичном формате X= 01000001
```

Рис. 104. Вывод на экран однобайтного числа в десятичном и двоичном форматах

Для работы с двухбайтными (четырехбайтными, восьмибайтными и др.) целыми беззнаковыми числами можно организовать аналогичную конструкцию типа данных. В

ней создать битовую карту с соответствующим числом однобитных полей – 16, 32 или 64 и объединить (union) их с целым беззнаковым числом типа short int, int или long. В остальном отличий, по существу, нет.

Можно также объединить с помощью структуры union целое беззнаковое число с массивом нужного количества битовых карт, как это сделано в (*Листинг 134*). Пример использования приведен на *Рис. 105*.

Листинг 134

```
union u2Byte
{
  short int word; // обращение к двухбайтному числу целиком
  BitMap bm[2]; // массив из двух битовых полей
};
//---------------------------------------------------------------------------
u2Byte Y;
Y.word = 2456329;
cout << "Y= "
  << Y.bm[1].b7 << Y.bm[1].b6 << Y.bm[1].b5 << Y.bm[1].b4 // старший байт
  << Y.bm[1].b3 << Y.bm[1].b2 << Y.bm[1].b1 << Y.bm[1].b0 << " "
  << Y.bm[0].b7 << Y.bm[0].b6 << Y.bm[0].b5 << Y.bm[0].b4 // младший байт
  << Y.bm[0].b3 << Y.bm[0].b2 << Y.bm[0].b1 << Y.bm[0].b0;
```

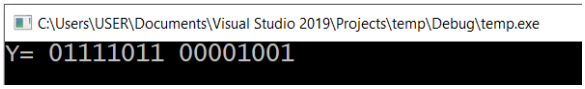

Рис. 105. Вывод на экран двухбайтного числа в двоичном формате

Контрольные вопросы:

1. Как запрограммировать побитное обращение к переменным целых типов.
2. Как работает побитная операция «сдвиг», что получится в результате 23<<3?
3. Как работает двоичная побитная операция & «и», что получится в результате вычисления 23&126?
4. Как размещаются в памяти поля переменной типа union?
5. Как работает двоичная побитная операция | «или», что получится в результате вычисления 23|112?
6. Как работает двоичная побитная операция ^ «исключающее или», что получится в результате 103^112?
7. Объединение содержит три поля unsigned short X, double Z и char Y, какой размер будет занимать переменная этого типа?
8. Как работает побитная операция «сдвиг», что получится в результате 23>>2?
9. Что такое битовые поля и как с ними работать?

12. Введение в классы

Ложась спать, программист ставит рядом на столик 2 стакана: один с водой — если захочет пить, второй пустой — если не захочет.

12.1. Класс

Классы и объекты в *C++* являются основными концепциями *объектно-ориентированного программирования – ООП* [*1-27*].

Классы в C++ это тип данных, описывающий *методы* (встроенные подпрограммы) и *свойства* (поля, тип ячеек памяти) объектов. Иными словами, класс объединяет в себе структурированные данные и подпрограммы для работы с ними.

Объекты – это переменные такого типа, называемые также *экземплярами* данного класса.

В объектно-ориентированном программировании существует четыре основных принципа построения классов:

1. *Абстракция* – это принцип выделения в моделируемом процессе только тех его свойств, которые понадобятся для решения конкретной задачи, и формализация их в виде класса.
2. *Инкапсуляция* – это свойство, позволяющее объединить в классе и данные (структуры), и методы (встроенные подпрограммы), работающие с ними и скрыть детали реализации от пользователя.
3. *Наследование* – это механизм, позволяющий создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.
4. *Полиморфизм* – свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

ООП является основой современного программирования, оно реализовано во всех современных языках высокого уровня, в том числе и в C++. ООП базируется на понятии классов и их объектов.

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 последнем параграфе *главы*, где подробно будет рассмотрен механизм наследования.

Зачем вообще разделять уровни доступа? Этот механизм организован для защиты данных. Как для предупреждения непреднамеренной их порчи, так и из соображений безопасности. Ниже будет показано, как этот механизм реализуется на языке C++.

Спецификаторы доступа и порядок их использования определяются программистом по необходимости. Рассмотрим простейший пример использования класса (*Листинг 135*). Похоже на структуру, не правда ли?

Листинг 135

```
  #include <iostream>
  using namespace std;
//---------------------------------------------------------------------------
  class date // имя класса
  {
  private: // доступ ко всем последующим полям ограничен
    int day;   // день
    int month; // месяц
    int year;  // год
  public: // доступ ко всем последующим полям открыт
    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 метода date::message() класса date
    X.day = 32;   // ошибка: доступ запрещен
    X.month = 13; // ошибка: доступ запрещен
    system("pause");
  }
```

В данном примере определен класс с именем `date`, содержащий в разделе `private` три поля `int day`, `int month` и `int year`. Раздел `public` включает в себя метод `void message()`.

В теле программы `main()` объявлена переменная `X` типа `date` (говорят: объявлен объект `X` класса `date`). После того как объект класса объявлен, можно пользоваться его полями и 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 `message()` не может быть вызвана, к ней можно обратиться только из какого-нибудь объекта класса `date`. Принадлежность метода классу обозначается оператором `"::"`. В нашей программе не существует подпрограммы `message()`, но существует метод `date::message()`.

В теле класса этот метод объявлен после спецификатора доступа `public`, поэтому в главной функции `main()` доступ к методу `X.message()` открыт. В отличии от доступа к полям `int day, month, year` – к ним доступ запрещен из-за спецификатора `private`.

12.2. 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-`*методы*, с помощью которых можно манипулировать элементами данных. `set-`*методы* служат для инициализации полей объекта, а `get-`методы позволяют выдать значения полей объекта.

Обратили внимание, что в приведенном выше примере (*Листинг 135*) делается попытка присвоить полю `X.day` число `32`, а полю `X.month` – числа `13`? Ясно ведь, что не бывает 32 числа ни в одном месяце, и самих месяцев – не больше 12. В этом суть ООП – программное описание природного предмета (факта, процесса), например – даты, должно соответствовать его естественным свойствам. Как же обеспечить это соответствие программно? Ведь кому-то из пользователей нашего класса может взбрести в голову присваивать `X.day=32`?

Для этого служат `set-` и `get-`*методы*, в теле которых можно провести требуемую предварительную проверку на правильность вводимых данных.

Доработаем класс `date` так, чтобы в нём присутствовали `set-` и `get-`*методы*, с помощью которых можно было организовать контролируемый, управляемый доступ к данным из раздела `private` (*Листинг 136*).

Листинг 136

```
#include <iostream>
using namespace std;
//------------------------------------------------------------------------
class date
{
private:
  int day;   // день
  int month; // месяц
  int year;  // год
public:
  void message()
  { // отобразить текущую дату
    cout << "date: " << day << "." << month << "." << year << endl;
  }
  //---------------------------------------- get-методы
  int get_day() { return day;  }
  int get_month() { return month; }
  int get_year() { return year; }
  //---------------------------------------- set-методы
  void set(int n_day, int n_month, int n_year)
  { set_year(n_year);
    set_month(n_month);
    set_day(n_day);
  }
  void set_year(int n_year) { year = n_year; }
  void set_month(int n_month);
  void set_day(int n_day);
};
//------------------------------------------------------------------------
int main()
{
  system("chcp 1251");
    date X; // объявление объекта
    X.set(34, 23, 2024);
    cout << X.get_day() << "\n";
    X.set_month(X.get_month() + 1);
    X.message(); // вызов функции message() принадлежащей классу date
  system("pause");
}
```

Теперь, используя методы `X.set()`, `X.get_month()`, `X.set_month()` и другие, можно получать доступ к внутренним (`private`) данным объекта `X` нашего класса `date`.

Обратите внимание, что реализация не всех методов может быть прописана прямо в теле описания класса. В нашем примере методы `set_month()` и `set_day()` присутствуют только в виде прототипов. А где же их реализация? Она может располагаться теперь в любом другом месте программы.

Листинг 137

```
//-------------------------------------------------------------------------
void date::set_month(int n_month)
{
  if (n_month <= 0)
    month = 1;
  else
    if (n_month >= 13)
      month = 12;
    else
      month = n_month;
}
//-------------------------------------------------------------------------
void date::set_day(int n_day)
{
  day = n_day;
  if (n_day <= 0)
    day = 1;
  else
  {
    if ((n_day >= 32) && (month == 1 || month == 3 || month == 5 ||
      month == 7 || month == 8 || month == 10 || month == 12))
      day = 31;
    if ((n_day >= 31) && (month == 4 || month == 6 || month == 9 || month == 11))
      day = 30;
    if ((n_day >= 30) && (month == 2) && (year % 4 == 0)) // февраль високосного
года
      day = 29;
    if ((n_day >= 29) && (month == 2) && (year % 4 != 0)) // февраль не високосного
года
      day = 28;
  }
}
```

В этом примере (*Листинг 137*) из-за спецификатора доступа *private* к переменным `day`, `month` и `year`, получают доступ только методы класса. Причем, управляемый, контролируемый доступ. Функции, не принадлежащие классу, не могут обращаться к этим переменным.

Заметьте также, что для методов, описываемых внутри класса, могут опускаться модификаторы принадлежности `date::`. Но для методов, чья реализация производится вне тела класса, этот модификатор обязателен, как в примере (*Листинг 137*).

Листинг 138

```
int main()
{
  int day, month, year; // вспомогательные переменные
  cout << "день: ";    cin >> day;
  cout << "месяц: ";   cin >> month;
  cout << "год: ";     cin >> year;
  date X; // объявление статического объекта класса date
  date* Y = &X; // объявление указателя на X
  Y->set(day, month, year);
  Y->message();
  //cout << "год= " << Y->year; << endl; // ошибка: доступа к полю private
```

```
  cout << "год= " << Y->get_year() << "\n"; // доступ по методу get_year()
}
```

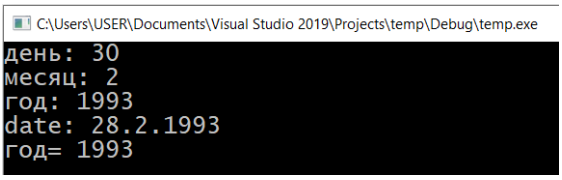

Рис. 106. Работа с закрытыми свойствами класса через set- и get-методы

Рассмотрим более внимательно пример (*Листинг 138*). В нем задается указатель `date* Y` и ему сразу же присваивается адрес `Y = &X`. Дальше мы пользуемся данными, хранящимися в экземпляре `X` через указатель `Y`, пользуясь оператором `"->"`. Но между объявлением переменной `X` и вызовом метода `Y->set(day, month, year)`, инициирующего объект класса `date` начальными значениями, проделывается ряд операций (см *Листинг 138*). Получается, мы работали с неинициированной переменной и её адресом, не задав её значений! А если бы полях переменной класса содержались какие-то динамические объекты? Это без сомнения привело бы к ошибке!

Вспомним пример со структурой `struct STR112`, содержащей динамическую строку (*Листинг 107 – Листинг 112*). В том примере нам пришлось вручную создавать динамический массив, а его адрес хранить в одном из полей структуры `STR112`. Нами были написаны специальные подпрограммы для корректного выделения и освобождения памяти из-под динамического массива.

Логично предположить, что в классах эта проблема как-то решается. Тем более, что у них есть такой инструмент, как встроенные в каждую переменную подпрограммы. Можно ли сделать так, чтобы определенные методы вызывались автоматически при создании и удалении объекта класса? Есть такие методы.

12.3. Конструктор и деструктор

При создании объектов класса, можно сразу же проинициализировать их поля, а при уничтожении – совершить обратные действия. Выполняют эту функцию *конструкторы* и *деструкторы*.

Конструктор – это метод, вызываемый безусловно при создании объекта класса. Обычно он выполняет начальную инициализацию элементов данных и выделяет память под динамические объекты. Имя конструктора обязательно должно совпадать с именем класса.

Важным отличием конструктора от остальных методов 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 пустой 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 136*), добавив к нему конструктор `date()` и деструктор `~date()`.

```
                                                                Листинг 139
class date
```

```
{
private:
  int day;   // день
  int month; // месяц
  int year;  // год
public: //-------------------------------------------------------------------------
  date(int n_day, int n_month, int n_year) // конструктор с параметрами
  { set(n_day, n_month, n_year); }
  date() { set(1, 1, 2000); } // конструктор по умолчанию – без параметров
  ~date() { } // пустой деструктор
//-------------------------------------------------------------------------------
  void message();
  int get_day();
  int get_month();
  int get_year();
  void set(int n_day, int n_month, int n_year);
  void set_year(int n_year);
  void set_month(int n_month);
  void set_day(int n_day);
};
```

Конструкторов может быть несколько (*Листинг 139*) – лишь бы они отличались списком параметров (имена-то у них должны быть одинаковые). В нашем примере первый конструктор имеет три параметра, через которые он получает информацию о дате. В теле этого конструктора вызывается `set-`*метод* для установки даты. Второй конструктор вызывается при объявлении (и создании) объекта класса в том случае, если параметры не указаны В этом конструкторе выполняется начальная инициализация закрытых полей класса датой `01.01.2000`. Ниже приведены примеры вызовов конструкторов при создании объектов класса.

```
  date X; // вызывается конструктор по умолчанию
  date* Y; // это только указатель: конструктор не вызывается
  Y = new date(1, 2, 2023); // вызывается конструктор с параметрами
  date Z(12, 11, 7); // вызывается конструктор с параметрами
  delete Y; // вызывается деструктор
  date R(1.02, 2021); // ошибка: отсутствует конструктор с таким списком аргументов
```

Обратите внимание, что при создании указателя `date*Y`, конструктор не вызывается. Потому как это только указатель, а будет ли он использован для создания нового динамического объекта или для адресации уже существующих статических или динамических переменных – не известно. А вот в момент создания динамической переменной `Y=new date(1,2,2023)` вызывается конструктор. При удалении динамической переменной `delete Y` вызывается деструктор. Деструктор для статических переменных тоже вызывается, но это происходит автоматически в момент закрытия программы.

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). Деструктор вызывается в конце области существования статического объекта класса, обычно – при завершении программы.

Создание и уничтожение динамического объекта явно задается и реализуется программистом. Оператор `new` и функции `calloc(), malloc()` создают объект, вызывая его конструктор. А оператор `delete` и функция `free()` уничтожают объект класса, при этом вызывая его деструктор. 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 137*) задается динамический объект `Y` класса `date`. Для его создания используется оператор `new`, вызывающий конструктор `Y = new date(1, 2, 2023)` для инициализации полей динамического объекта `Y`. Соответственно, оператор `delete` вызывает деструктор `~date()`, предназначенный для освобождения динамических полей объекта класса, если таковые имеются.

Листинг 140

```
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.

Разберем пример (*Листинг 141*), 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 её в деструкторе.

Листинг 141

```
class VectorType
{ private:
        int count;              // размер массива
        char* name;             // имя массива
        long double* vect;      // указатель на массив переменной длины
public:
      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
};
//---------------------------------------------------------------------------
VectorType::VectorType() // конструктор по умолчанию
{     count = 5 + rand() % 16; // случайный размер массива от 5 до 20
      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) / 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 141*) 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()`.

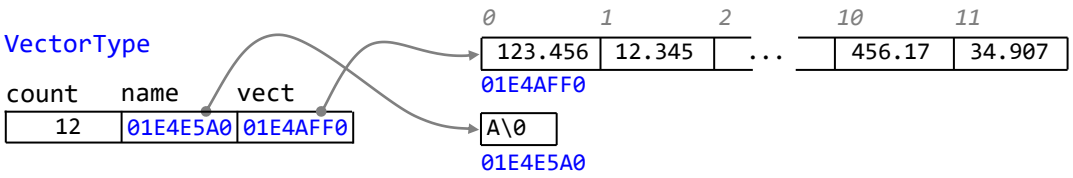

Рис. 107. Структура объекта класса `VectorType`

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`. Они служат только для того, чтобы проиллюстрировать тот момент выполнения программы, в который будут работать конструктор и деструктор для динамической и статической переменных.

Листинг 142

```
void VectorType::Show(const char* annot)
{  cout << annot << " " << name << " = { ";
   for (int i = 0; i < count; i++)
       cout << vect[i] << " ";
   cout << "}" << endl;
}
//---------------------------------------------------------------------------
int main()
{  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
   system("pause");
}
```

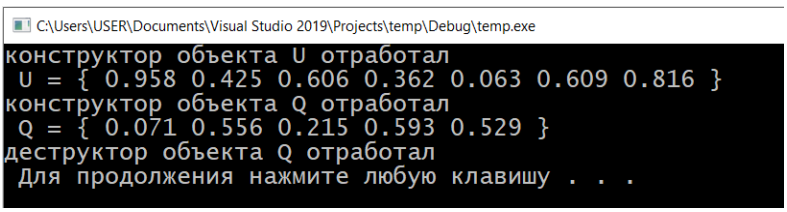

Рис. 108. Результаты работы программы

Из рисунка (Рис. 108) видно, что конструкторы и деструкторы объектов класса `VectorType` корректно запускаются, выделяют и освобождают память при работе с

динамическими полями класса. Обратите внимание, что деструктор статического объекта `U` еще не отработал. Он запустится сразу после того, как завершит свою работу команда `system("pause")`. Попробуйте разглядеть сообщение от него.

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 143*).

Листинг 143

```
VectorType::VectorType(const char* n, int c) // конструктор с параметрами
{	count = c; // копирование параметра c
	name = new char[strlen(n) + 1];
	strcpy_s(name, strlen(n) + 1, 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_s(name, strlen(s) + 1, 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
}
//-------------------------------------------------------------------------
int main()
{
	system("chcp 1251");
	srand(time(NULL));
	VectorType X;
	X.Show("по умолчанию");
	VectorType* Y;
	Y = new VectorType;
	Y->Show("по умолчанию");
	delete Y;
	VectorType* Z = new VectorType("массив Z", 7);
	Z->Show("с параметрами");
	delete Z;
	VectorType A("E:\\array.txt");
	A.Show("считан из файла");
	system("pause");
}
```

Результаты работы программы, приведенные на Рис. 109 иллюстрируют работу конструктора с параметрами и конструктора чтения для статических и динамических объектов класса. Обратите внимание (*Листинг 143*) на то, как отработает конструктор чтения в случае, если файл открыть не удастся.

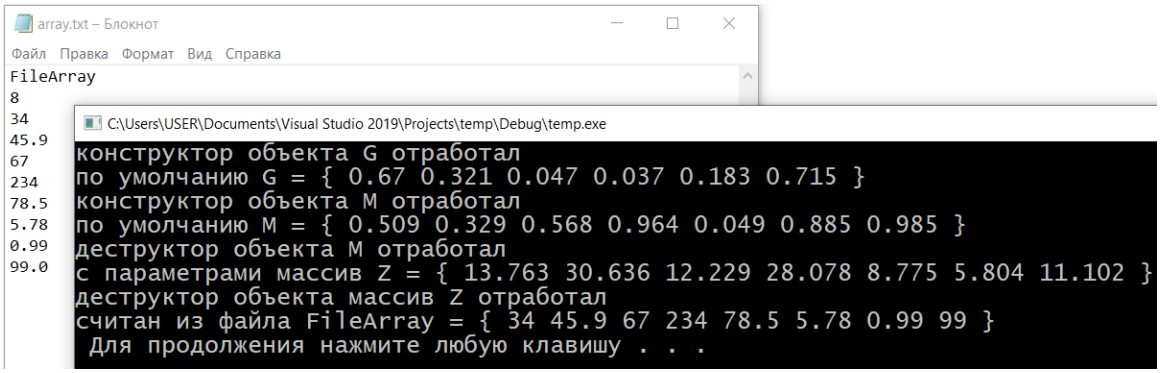

Рис. 109. Результаты работы программы

12.4. 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.), называемые *левый операнд* и *правый операнд*, соответственно.

Мы научились проектировать новые типы данных – классы, логично иметь возможность создавать для своих классов необходимые операции. В примере ниже (*Листинг 144*) 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 144

```
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);
  // оператор индексации элементов класса VectorType
  long double& operator [] (int indx);
  // оператор присваивания объектов класса VectorType
  VectorType& operator = (const VectorType&);
..// дружественная функция, оператор сравнения объектов класса VectorType
  friend bool operator == (const VectorType&, const VectorType&);
};
//---------------------------------------------------------------------------
long double LD_MIN = -1.7976931348623158e+308; // самое маленькое число
//---------------------------------------------------------------------------
long double& VectorType::operator [] (int indx)
{   // оператор индексации элементов класса VectorType
```

```
    if ((indx < 0) || (indx > count - 1)) // если ошибка диапазона
      return LD_MIN;
    return vect[indx];
}
//---------------------------------------------------------------------------
VectorType& VectorType::operator = (const VectorType& V2)
{ // оператор присваивания объектов класса VectorType
  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 число типа `long double` – константу `LD_MIN= -1.7976931348623158e+308`.

Рассмотрим, как вызываются созданные операторы в основной программе:

Листинг 145

```
int main()
{   srand(time(NULL));
    VectorType X("X", 8);
    X.Show("");
    cout << "X[0]= " << X[0] << endl; // оператор индексирования X.operator[](0);
    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.operator=(X);
    Y.Show("Y=X\n");
    system("pause");
}
```

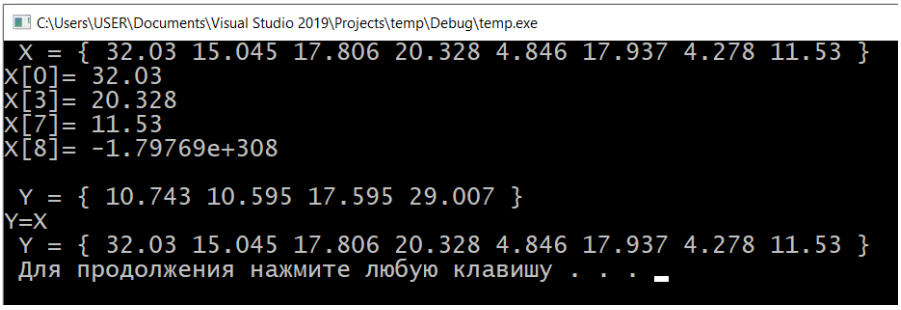

Рис. 110. Результаты работы программы

Результаты работы программы (*Рис. 110*) иллюстрируют использование операторов индексации и присваивания. Можно видеть, как вызывается оператор индексации `X[0]` (*Листинг 145*). Формально, оператор индексации нужно записывать так: `X.operator[](0)`. Левым операндом здесь выступает ссылка на объект `X` класса `VectorType`, иначе – указатель на самого себя `this`. В качестве правого операнда – целое

число, в данном случае – `0`. Оператор возвращает ссылку `VectorType&`. На примере `X[8]` видно, как отрабатывает оператор индексации выход индекса за границы диапазона.

Оператор присваивания `VectorType& operator = (const VectorType&)`, на самом деле правильно должен быть записан так `Y.operator=(X)`. Левым операндом здесь выступает ссылка на тот объект `Y` класса `VectorType`, которому производится присваивание значений полей. Правым операндом служит ссылка на `X` – тот объект класса `VectorType`, из которого берутся присваиваемые значения.

Оператор `operator=()` 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 144*), 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 `operator=()`. 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.

12.5. *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 144*) 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 146

```
bool operator == (const VectorType& V1, const VectorType& V2)
{ // оператор сравнения
	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;
}
```

```
//---------------------------------------------------------------------------
int main()
{
  system("chcp 1251");
  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 не равны" << 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;
  system("pause");
}
```

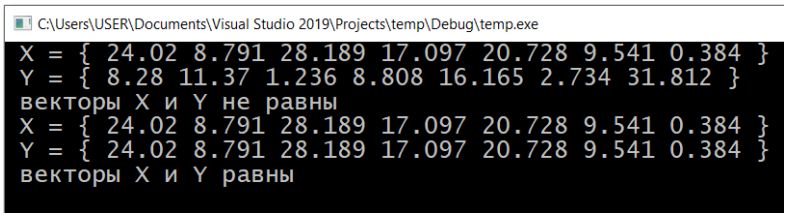

Рис. 111. Результаты работы программы

Проиллюстрировать работу операторов сравнения и присваивания можно на примере динамических объектов `Y` и `X` класса `VectorType` (*Листинг 146*). 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` что значит «*содержимому указателя X присвоить содержимое указателя Y*» и `*Y==*X`, «*содержимое указателя Y сравнивается с содержимым указателя X*».

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 144*), а делалась бы попытка вызвать типовые операторы *сравнения указателей* и *присваивания указателей*, реализованные в стандарте языка *C++*. Что скорее всего привело бы к ошибке.

12.6. 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 137*). 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 `main()` в файле вашего проекта, например, `test04.cpp` (*см.* Рис. 112).

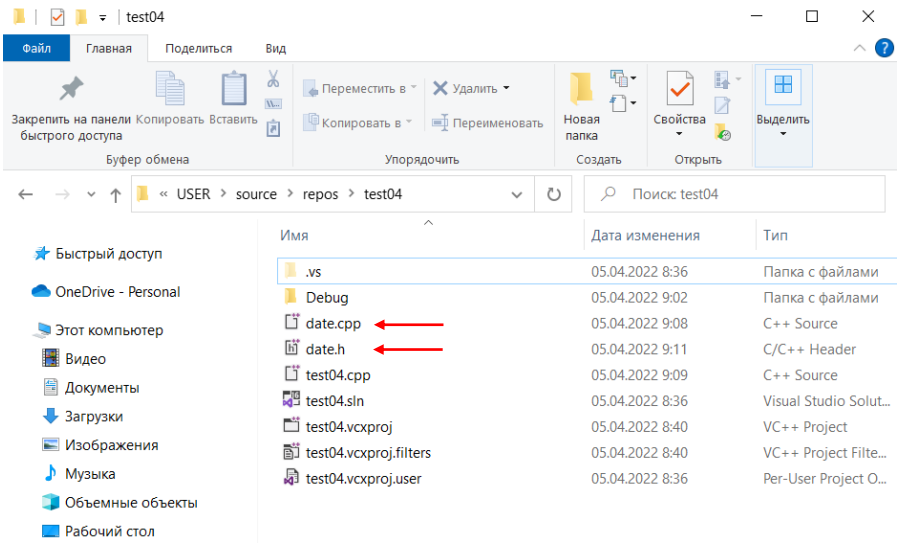

Рис. 112. Архитектура Visual C++ проекта test04

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` (в папке *Debug*) или специальным образом скомпонованные файлы библиотек `date.lib` или `date.dll` – *см. Рис. 113*.

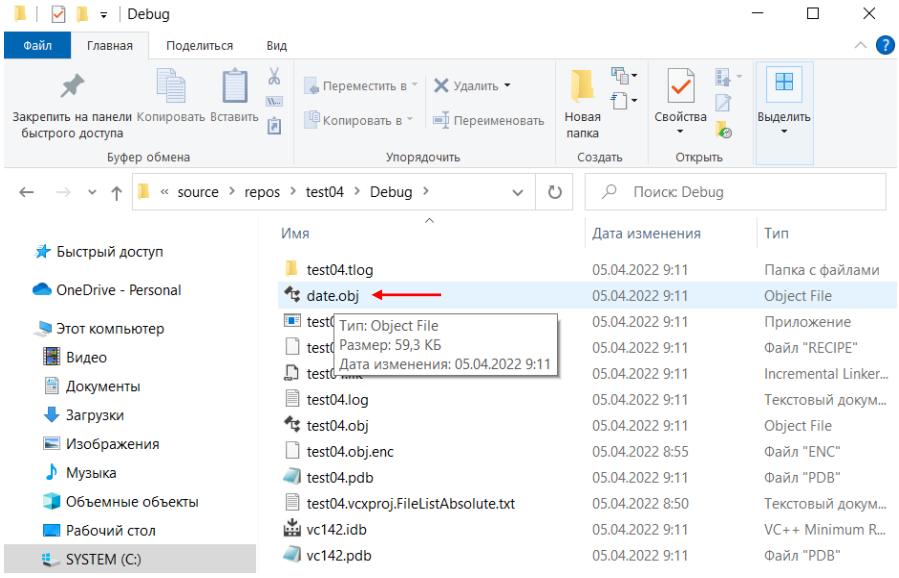

Рис. 113. Откомпилированный файл date.obj в папке проекта test04

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 147*).

Листинг 147

```
// 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
	void show_str();
};
#endif DATE_H // если имя DATE_H уже определено, повторно не определять
```

Заголовочный файл `date.h` предваряется и завершается директивами компилятору `#ifndef`, `#define` и `#endif`. Эти директивы называются «*препроцессорная обертка*», поговорим о ней позже.

Рассмотрим процесс добавления нового класса в проект в виде отдельно компилируемого файла (*Рис. 114*).

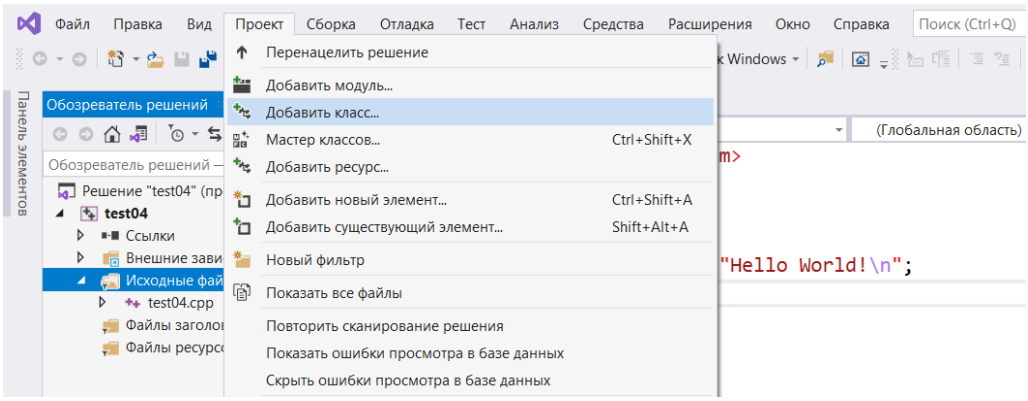

Рис. 114. Добавление нового класса в проект в виде отдельного файла

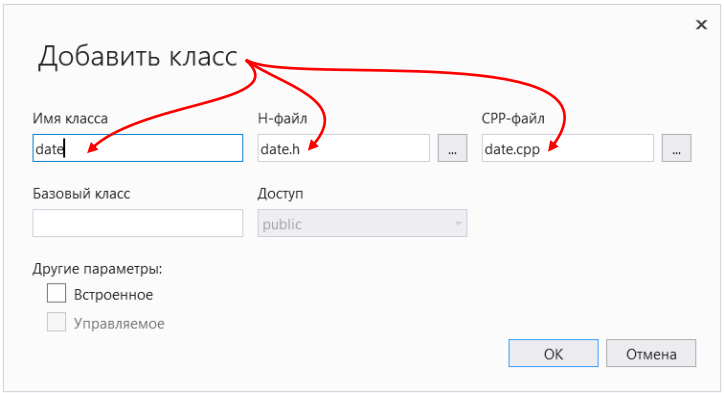

Рис. 115. Интерфейс меню добавления нового класса в проект

В меню добавления нового класса в проект (*Рис. 115*) предлагается выбрать имя класса и автоматически предлагается создать файлы для этого класса. Т.е. по умолчанию предполагается размещение классов отдельно от проекта.

Можно видеть, что в проекте создаются и добавляются два файла, которые теперь используются для проекта (*Рис. 116*). Отметим, что эти файлы можно создать вручную, а не при помощи меню создания класса (*Рис. 115*) и добавить в проект, выбрав в меню вкладки «*Проект/Добавить существующий элемент*» или *<Shift>+<Alt>+<A>*.

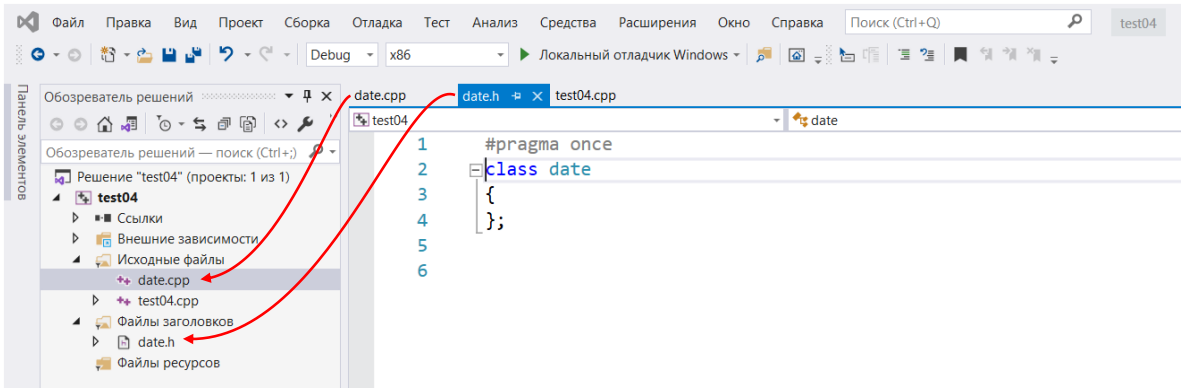

Рис. 116. Добавленные в проект файлы нового класса

Интерфейс класса

Интерфейс класса – это описание класса, определяющее его методы и свойства, (названия типов данных полей и прототипы методов) не раскрывающее алгоритмов. Располагается в файле `имя_класса.h`.

Реализация класса – это способ осуществления алгоритмов работы класса – определение/задание тел методов, ответ на вопрос «как работают те или иные методы?». Располагается в файле `имя_класса.cpp`.

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. 112*);
2. определить интерфейс класса в заголовочном файле (*Листинг 147*);
3. добавить в проект исполняемый файл *.cpp (*Рис. 112*);
4. в исполняемом файле выполнить реализацию класса (*Листинг 148*);
5. подключить заголовочный файл к программе, если он не подключился автоматически;
6. убрать файл реализации и заменить его откомпилированным файлом (*Рис. 120*).

Теперь рассмотрим содержимое файла реализации методов класса `date` (*Листинг 148*). 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 *оператор принадлежности* `"::"`, эта операция привязывает метод, объявленный извне, к классу.

Проверьте, что в файле реализации класса `date.cpp` присутствует подключение заголовочного файла `#include "date.h"`. Иначе компилятору не понятно, откуда брать прототипы методов, которые в этом файле реализуются.

```
                                                                Листинг 148
                                                  // date.cpp файл реализации
#include "date.h" // подключить интерфейс класса
#include <iostream>
using namespace std;
// ------------------------------------------------------------------------
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; }
void date::show_str()
{
	const char* month_name[12] = { "января", "февраля", "марта", "апреля",
	"мая", "июня", "июля", "августа", "сентября", "октября", "ноября",
	"декабря" };
	std::cout << day << " " << month_name[month-1] << " " << year << " года\n";
}
```

Итак, интерфейс класса определён, методы класса объявлены, осталось подключить заголовочный файл в исполняемом файле проекта – в программе `main()`. Чтобы главная функция увидела созданный нами класс и смогла его использовать, необходимо включить определение класса `#include "date.h"` в исполняемом файле.

Листинг 149

```
// test04.cpp главный файл проекта
#include <iostream>
#include "date.h" // подключить интерфейс класса
int main()
{
  system("chcp 1251");
  date* Y = new date(5, 8, 2015);
  Y->show_str();
  delete Y;
  system("pause");
}
```

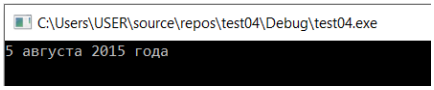

Рис. 117. Результаты работы программы

После компиляции в папке Debug нашего проекта появится файл с откомпилированными методами нового класса `date` – `date.obj` (*Рис. 113*). Теперь мы можем подключать наш класс `date` к другим проектам не передавая в них файл с исходными кодами нашего класса `date.cpp`, а передавая только файл откомпилированной библиотеки методов `date.obj` и заголовочный файл `date.h`. Рассмотрим, как это делается.

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 избыточное, лишнее 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.

Подключение класса к новому проекту

Теперь наш класс подключим к новому проекту. Перенесем текст из программы `main()` в созданный заново проект. Простой перенос текста сразу же выявит ошибки, связанные с тем, что библиотека `#include "date.h"` в этом новом проекте отсутствует. Следовательно, класс `date` новой программе не известен. Скопируем в папку нового проекта файл `date.h` и вручную подключим его в проект, выбрав в меню вкладки «*Проект/Добавить существующий элемент*» или <*Shift*>+<*Alt*>+<*A*>.

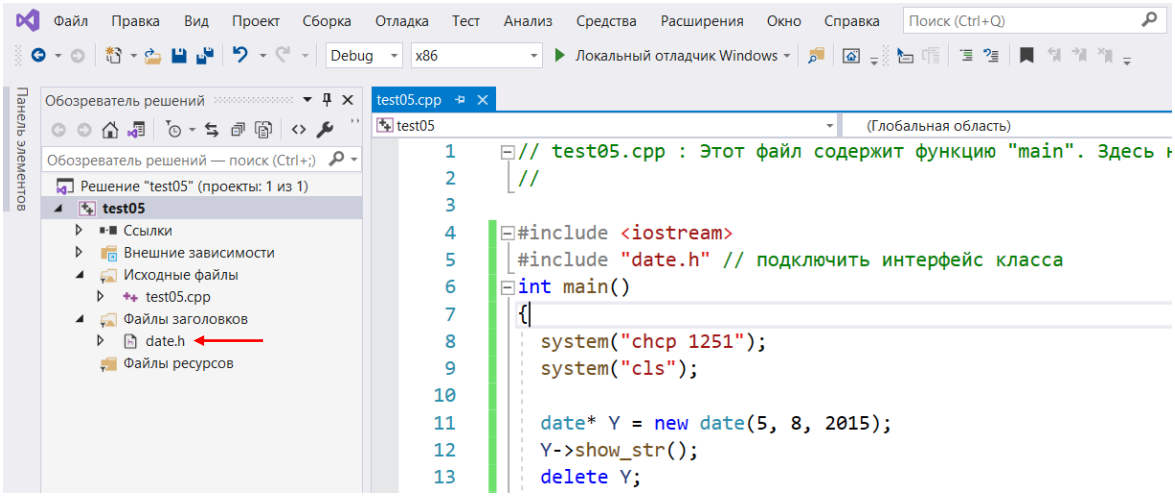

Рис. 118. Подключение заголовочного файла date.h к новому проекту

Теперь (*Рис. 118*) сообщения об ошибках относительно объектов и методов класса `date` исчезли. Т.е. компилятор знает, что в проекте имеется описание этого класса и понимает его структуру. Более того, предполагается, что где-то в проекте имеется и реализация этих методов.

Но на самом деле реализации нет, ведь мы её еще не добавляли. Если сейчас запустить отладку и исполнение нашего проекта, то он на этапе линковки (сборки) проекта обнаружит ошибку – отсутствуют конструктор, деструктор и метод `show_str()` класса `date` (сообщения об ошибках такого рода приведены на *Рис. 119*).

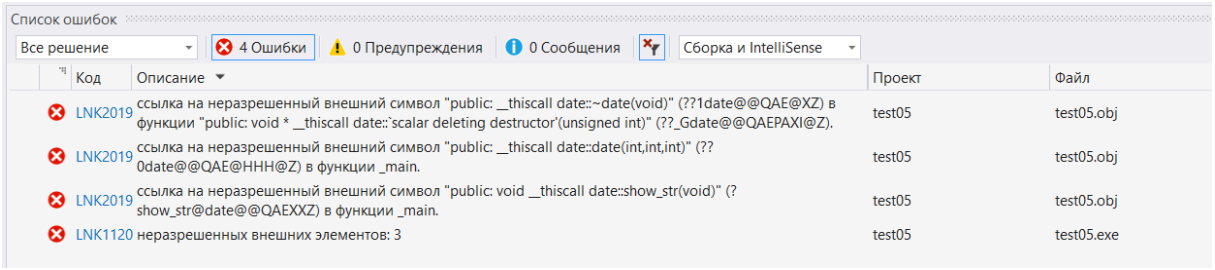

Рис. 119. Ошибки: в проекте отсутствует реализация методов класса, описанных в заголовочном файле

Исправить это просто, нужно добавить в проект файл `date.obj` откомпилированной библиотеки класса `date`, выбрав в меню вкладки «*Проект/Добавить существующий элемент*» или <*Shift*>+<*Alt*>+<*A*> (*Рис. 120*).

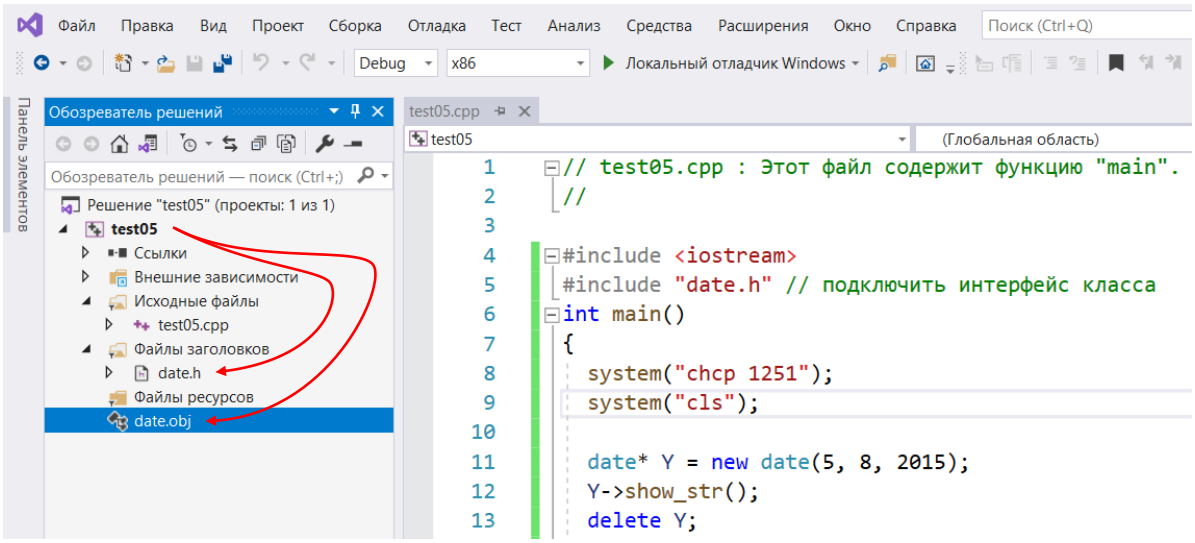

Рис. 120. Подключение к проекту файла откомпилированного класса

Обратите внимание, что исполняемого файла `date.cpp` в этом проекте нет (сравните с *Рис. 116*), а работа с классом `date` успешно реализуется.

12.7. Наследование

Наследование – это механизм создания нового класса на основе ранее созданного. При этом (наследующие) классы-потомки приобретают все `public` и `protected` поля и методы класса-предка (наследуемого). Наследование имеет смысл, если множество разнородных объектов имеют общие характеристики или функции.

Рассмотрим пример классов для работы со студентами (*Листинг 150*). Целесообразно выделить в отдельный (базовый) класс `THuman` информацию о фамилии, имени, дате рождения и пр. (информация о студенте как о человеке). На основе этого класса создавать дочерний класс `TStudent` (для описания студента как учащегося), добавив информацию об учебном заведении, факультете и оценках.

Листинг 150

```
#pragma once
#ifndef THUMAN_H
#define THUMAN_H
//---------------------------------------------------------------------------
class THuman // базовый класс
{ private:
    char* first_name = nullptr;
    char* last_name = nullptr;
    bool gender = 1;
```

```
  protected:
    int year_of_birth = 0;
    int get_age() { return 2022 - year_of_birth; }
  public:
    THuman() {};
    THuman(const char*, const char*, int, bool);
    ~THuman();
    void show();
};
//-------------------------------------------------------------------------
class TStudent : public THuman // наследуемый класс от класса THuman
{ protected:
  private:
    char* university = nullptr;
    char* faculty = nullptr;
    int  GPA = 0;
  public:
    TStudent() {};
    TStudent(const char*, const char*, int, bool,
             const char*, const char*, int);
    ~TStudent();
    void show();
    void show_age();
};
#endif THUMAN_H
```

При описании дочернего класса необходимо указать базовый класс (от которого наследуются поля и методы). Это можно сделать, поставив после имени дочернего класса `TStudent` символ `":"`, а затем пишется имя базового класса `public THuman` (с указанием уровня доступа). Вот так: `class TStudent : public Thuman`.

Уровень наследования `public` определяет доступ класса `TStudent` ко всем полям и методам класса `THuman`, которые не являются `private`. Т.е. в нашем примере (*Листинг 150*) доступ к полям `ID, first_name, second_name` и `gender` для методов класса `TStudent` все же закрыт. Для примера поле `year_of_birth` разместим в разделе `protected` для того, чтобы понять разницу между спецификаторами доступа.

Ниже (*Листинг 151*) приведена реализация методов класса `THuman`.

Листинг 151

```
#include <iostream>
#include "THuman.h"
using namespace std;
//-------------------------------------------------------------------------
THuman::THuman(const char* FN, const char* LN, int YB, bool G)
{
  first_name = new char[strlen(FN) + 1];
  strcpy_s(first_name, strlen(FN) + 1, FN);
  last_name = new char[strlen(LN) + 1];
  strcpy_s(last_name, strlen(LN) + 1, LN);
  gender = G;
  if (YB>0)  year_of_birth = YB;
}
//-------------------------------------------------------------------------
THuman::~THuman()
{
  if (first_name)
    delete[] first_name;
  first_name = nullptr;
  if (last_name)
    delete[] last_name;
  last_name = nullptr;
}
```

```
//---------------------------------------------------------------------------
void THuman::show()
{
  if ((first_name && last_name))
    cout << first_name << " " << last_name << " ";
  cout << gender << " " << year_of_birth << "\n";
}
```

Доступ к методам класса `THuman`, у программ класса `TStudent` имеется, в виду того, что они имеют спецификатор доступа `public`, это иллюстрирует пример (*Листинг 152*). Здесь представлены реализации некоторых методов рассматриваемых классов и показаны возможности применения наследуемых методов, в том числе, конструктора и деструктора.

Листинг 152

```
//---------------------------------------------------------------------------
TStudent::TStudent(const char* FN, const char* LN, int YB, bool G,
                   const char* U, const char* F, int gpa)
{   this->THuman::THuman(FN, LN, YB, G); // конструктор базового класса
    university = new char[strlen(U) + 1];
    strcpy_s(university, strlen(U) + 1, U);
    faculty = new char[strlen(F) + 1];
    strcpy_s(faculty, strlen(F) + 1, F);
    GPA = gpa;
}
//---------------------------------------------------------------------------
TStudent::~TStudent()
{  if (university)
      delete[] university;
   university = nullptr;
   if (faculty)
      delete[] faculty;
   faculty = nullptr;
   this->THuman::~THuman();// деструктор базового класса
}
//---------------------------------------------------------------------------
void TStudent::show()
{  THuman::show(); // вызов метода show() базового класса
   if ((university && faculty))
      cout << university << " " << faculty << " ";
   cout << GPA << "\n";
}
```

Программа ниже (*Листинг 153*) иллюстрирует применение базового класса и дочернего класса. После неё приведены результаты её выполнения (*Рис. 121*).

Листинг 153

```
#include <iostream>
#include "THuman.h"
int main()
{   system("chcp 1251");
   THuman X("Иван", "Иванов", 2001, 1);
   X.show();
   TStudent Z("Петр", "Петров", 2002, 1, "TUSUR", "FET", 4);
   Z.show();
}
```

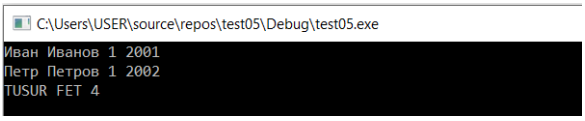

Рис. 121. Результат использования базового и дочернего классов

Стоит обратить внимание на вызов одноименных методов `THuman::show()` и `TStudent::show()`. Метод `THuman::show()` замаскирован в наследуемом классе `TStudent` новым одноименным методом `TStudent::show( )`. Однако, в зависимости от того, объектом какого класса вызывается этот метод, такой и используется.

Обратите внимание на уровень доступа `protected`, предоставленный классом `THuman` своему методу `get_age()` и полю `year_of_birth` (*Листинг 150*).

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 154*) вызова метода `get_age()` базового класса `THuman` из метода `TStudent::show_age()` наследуемого класса. Доступ разрешен, также к полю `THuman::year_of_birth`, поскольку `protected`. Там же приводится пример неправильного обращения к закрытому (`private`) полю и `THuman::first_name`.

Листинг 154

```
void TStudent::show_age()
{
  year_of_birth = 2000;
  cout << "age= " << this->get_age() << "\n" << year_of_birth << "\n";
  cout << first_name; // ошибка: доступ в наследуемых классах запрещен (protected)
}
```

Рассмотрим пример (*Листинг 155*) 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()` и поле `year_of_birth` – для всей программы `main()` они недоступны.

Листинг 155

```
#include <iostream>
#include "THuman.h"
int main()
{
  system("chcp 1251");
  THuman X("Иван", "Иванов", 2001, 1);
  X.show();
  X.year_of_birth = 2002; // ошибка: прямой доступ запрещен (protected)
  TStudent Z("Петр", "Петров", 2002, 1, "TUSUR", "FET", 4);
  Z.show();
  Z.year_of_birth = 2001; // ошибка: прямой доступ запрещен (protected)
  std::cout << Z.get_age(); // ошибка: прямой доступ запрещен (protected)
  Z.show_age();
  system("pause");
}
```

А вот доступ из метода `show_age()` (*Рис. 122*) имеется. Из главной программы `main()` можно спокойно вызывать методы (`public`) класса `TStudent` в теле которых использовать поля и методы родительского класса `THuman`, имеющие доступ `protected`. Таким образом осуществляется контролируемый доступ к `protected`-элементам в методах классов-наследников.

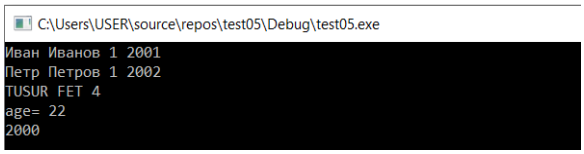

Рис. 122. Доступ к protected-элементам класса через наследование

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?

13. Графический интерфейс пользователя

Существует три способа работать с ресурсами операционной системы: из командной строки, посредством графического интерфейса и при помощи запущенного ранее приложения. Нужно понимать, что все объекты, присутствующие на экране операционной системы – окна, иконки, функциональные кнопки, объекты меню и т.д. выполнены в виде изображений. Пользователь, вызывая какое-либо приложение, запуская воспроизведение аудио или видео файла, обращаясь к ресурсам Интернет при помощи браузера, работая с таблицами, рисунками или текстом, взаимодействует с приложением посредством ***графического интерфейса***.

GUI (*Graphical User Interface* или *графический интерфейс пользователя*) – это вид пользовательского интерфейса, элементы которого выполнены в виде графических изображений. Эта глава посвящена изучению основ разработки графического интерфейса для разрабатываемых на C++ приложений.

Преимущества и недостатки GUI

GUI считается наиболее «дружественным» для новичков, только знакомящихся с ПК в целом или определенным программным обеспечением в частности. В программах для обработки изображений, любых графических элементов или видео GUI — единственное возможное решение.

GUI как правило интуитивно понятен, в нем реализуется концепция ***DWIM*** (*Do What I Mean* или дословно «делай то, что имеется в виду»). То есть система должна функционировать предсказуемо, чтобы пользователь интуитивно понимал, что произойдет после его определенного действия.

В то же время, GUI более требователен к памяти ПК, если его сравнить с текстовым интерфейсом командной строки. С ним труднее организовать полноценную удаленную работу, трудно автоматизировать, если это не было заложено по умолчанию разработчиком ПО.

13.1. Создание проекта Windows Forms в Visual Studio на C++

Для реализации GUI для операционной системы Windows традиционно использовались ресурсы Win32 API, однако, эта технология довольно запутана и не слишком универсальна. На ее основе разработаны готовые визуальные GUI компоненты, реализованные в виде дерева наследуемых классов и называемые ***Windows Forms***. Эта технология адаптируется IDE под нужную версию ОС, потому значительно удобнее и понятнее.

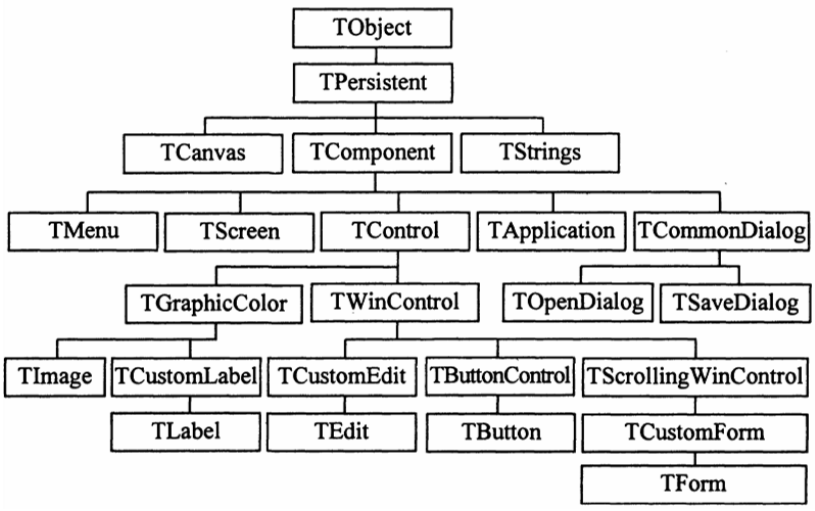

Рис. 123. Архитектура графических примитивов GUI

Создание *Проекта Windows Forms* на C++ в IDE Visual Studio начиная с 2012 года отсутствует в списке проектов, так что проект такого типа требуется создавать вручную.

В меню создания проекта (*File→Create→Project*) необходимо выбрать *тип проекта*. Требуется в подразделе *CLR* задать пункт *Пустой проект CLR* (*Рис. 124*) (*New Project→ Visual C++→ CLR→ CLR Empty Project*).

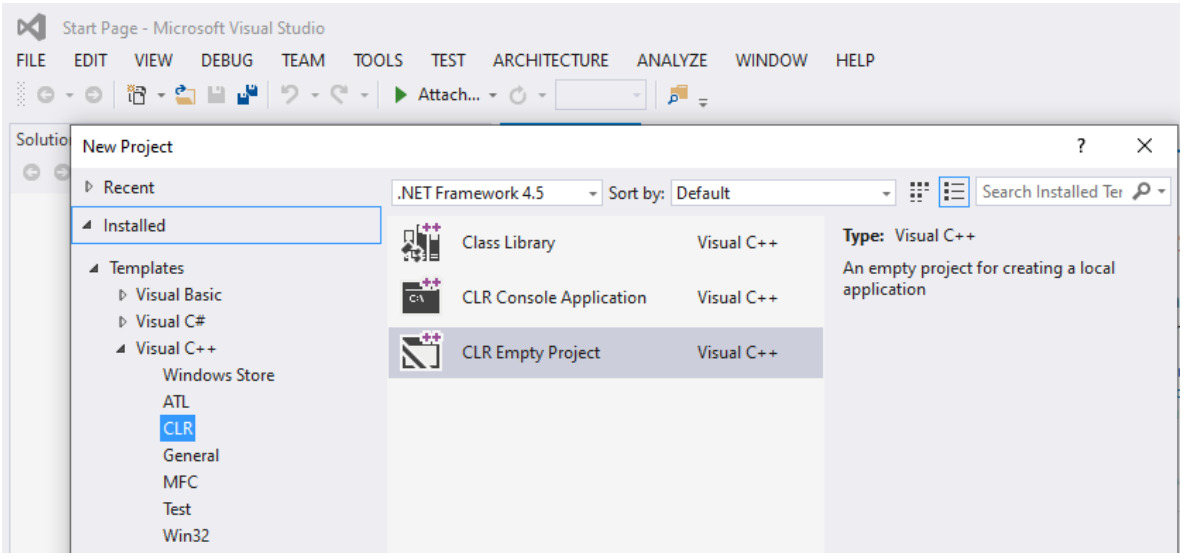

Рис. 124. Создание пустого проекта Visual Studio

Все файлы созданного проекта размещаются в папке рабочих документов пользователя в директории *C:\Users\User\Documents\Visual Studio\Projects* (если сам пользователь при установке и настройке IDE не изменил рабочую папку).

Теперь в созданный проект нужно добавить GUI-элемент `Form`, для этого в обозревателе решений (*Solution Explorer*) нужно кликнуть правой кнопкой мыши по созданному проекту и в открывшемся контекстном меню последовательно выбираем *добавить* (*Add*) и *создать новый элемент* (*New Item*), см. *Рис. 125*.

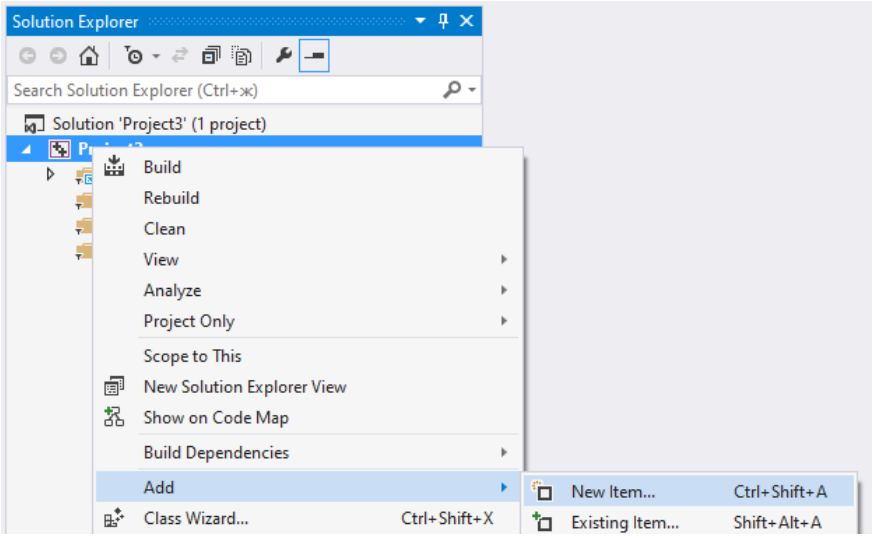

Рис. 125. Добавление к проекту нового элемента

В открывшемся меню в разделе *UI* выбираем GUI-класс *Форма* (`Windows Form`) – см. *Рис. 126* (*Add→ New Item→ UI→ Windows Form*).

В этот момент форма будет добавлена к проекту, в обозревателе решений (*Solution Explorer*) к проекту добавится два новых файла – файл исходного кода `MyForm.cpp` и заголовочный файл `MyForm.h`, кроме того в обозревателе *Конструктора формы* (`MyForm.h` [*Design*]) будет представлен внешний вид созданной формы (*Рис. 127*).

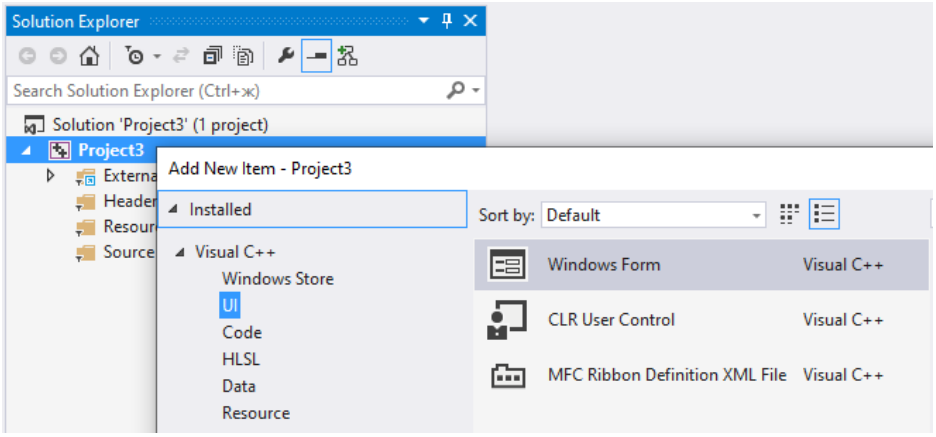

Рис. 126. Добавление к проекту Windows Form

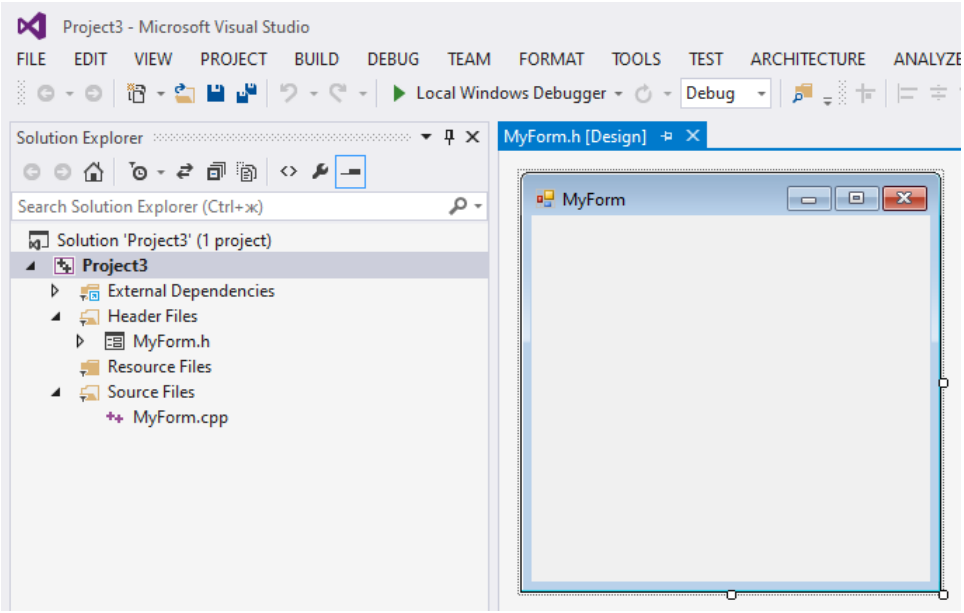

Рис. 127. К проекту Project3 добавлена форма Windows Form

В файле исходного кода `MyForm.cpp` содержится единственная строчка кода:

```
#include "MyForm.h"
```

В этот файл необходимо добавить следующий код:

Листинг 156

```
using namespace System;
using namespace System::Windows::Forms;
[STAThreadAttribute]
void Main(array<String^>^ args)
{
	Application::EnableVisualStyles();
	Application::SetCompatibleTextRenderingDefault(false);
	Project3::MyForm form;
	Application::Run(% form);

}
```

В приведенном примере проект называется `Project3`, а в разрабатываемом вами решении он может называться по-другому, поэтому в текст программы (*Листинг 156*) необходимо внести нужное имя проекта.

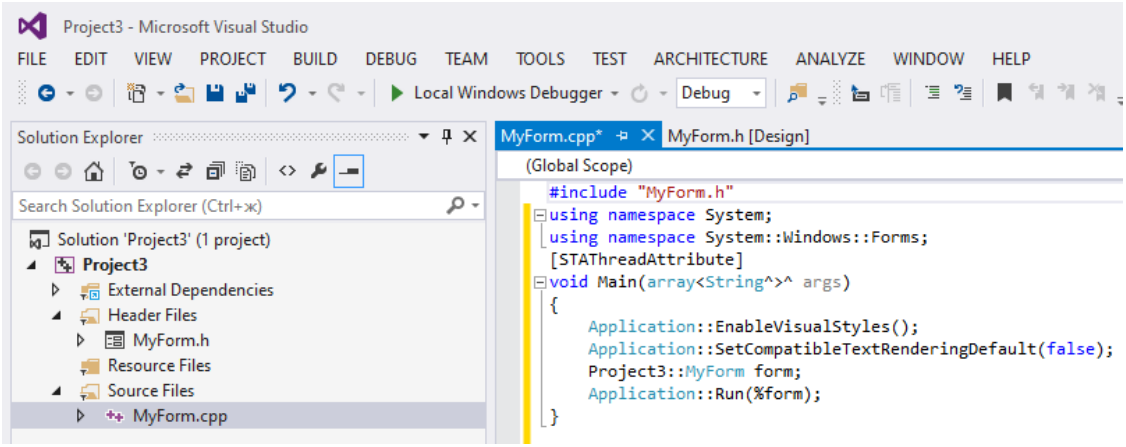

Рис. 128. Добавление стартовой программы `Main()` *с настройками* `Windows Form`

После этого в *Свойствах Проекта* (*Project3→ Properties*) нужно выбрать подраздел *Система* раздела *Компоновщик* (*Properties→ Linker→ System*) и в строке *Подсистема* (*SubSystem*) из выпадающего меню выбрать Windows (/SUBSYSTEM:WINDOWS) – как на *Рис. 129* – и нажать кнопку *Применить*.

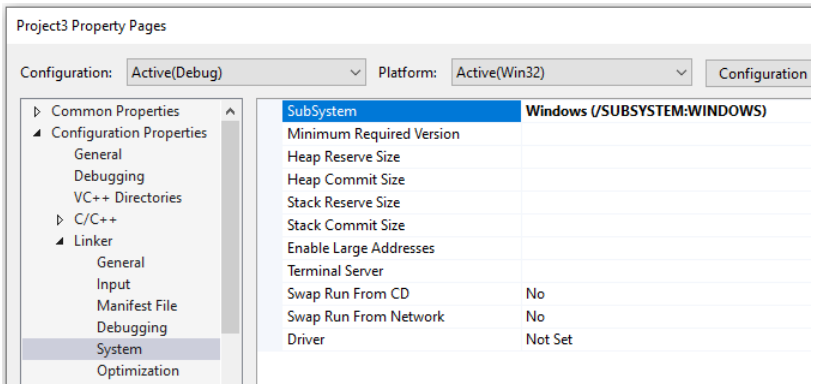

Рис. 129. Настройка параметров Компоновщика (Linker) проекта

Не закрывая окно свойств проекта, необходимо перейти в подраздел *Дополнительно* (*Advanced*) и в строке *Точка входа* (*Entry Point*) прописать имя главной функции `Main` и после этого нажать клавишу *OK*. (см. *Рис. 130*). Таким образом задается точка входа – подпрограмма `Main()`, заданная нами в файле `MyForm.cpp`, с которой начнется выполнение исходного кода.

Шаблон проекта *Windows Forms* в Visual Studio на C++ готов. Рекомендуется сохранить все файлы проекта (*FILE→ Save All*), откомпилировать решение (*BUILD→ Bild Solution*) и запустить его на выполнение (*DEBUG→ Start Debugging*).

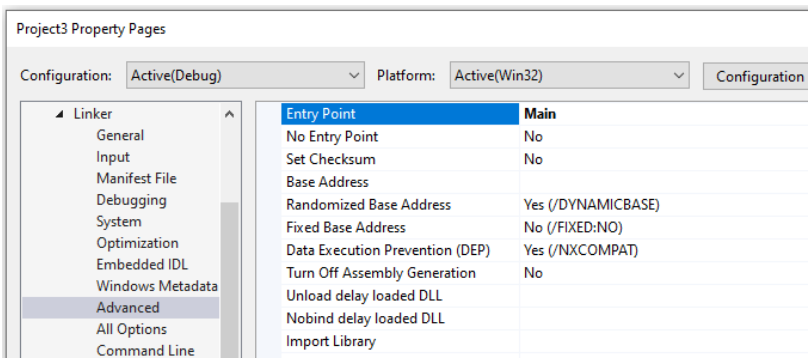

Рис. 130. Настройка точки входа проекта

На этом настройки проекта заканчиваются. Чтобы не проделывать описанную выше последовательность действий каждый раз при создании нового проекта, уместно

создать на базе только что созданного решения ***шаблон проекта*** и в дальнейшем пользоваться им при входе в *Visual Studio*.

13.2. Создание шаблона проекта

Для этого необходимо во вкладке *Файл* выбрать пункт *Экспорт шаблона* (*FILE→ Export Template*) см. *Рис. 131*.

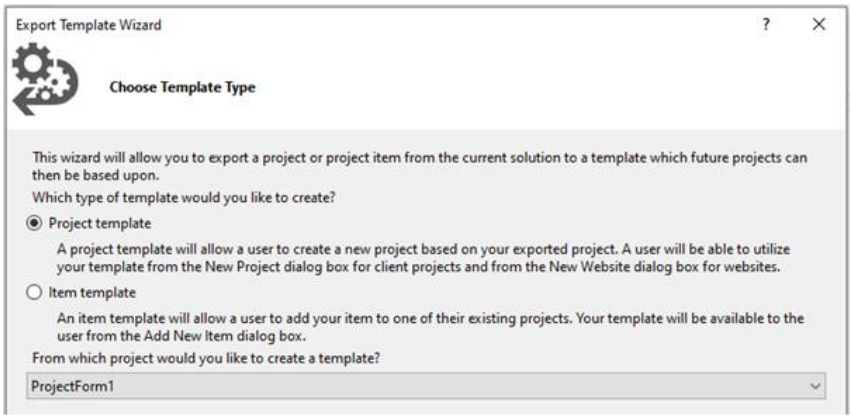

Рис. 131. Экспорт шаблона

В появившемся мастере шаблонов важно отследить, чтобы стояла галочка «*Автоматический импорт шаблонов в Visual Studio*» (*Automatically import the template into Visual Studio)* см. *Рис. 132*.

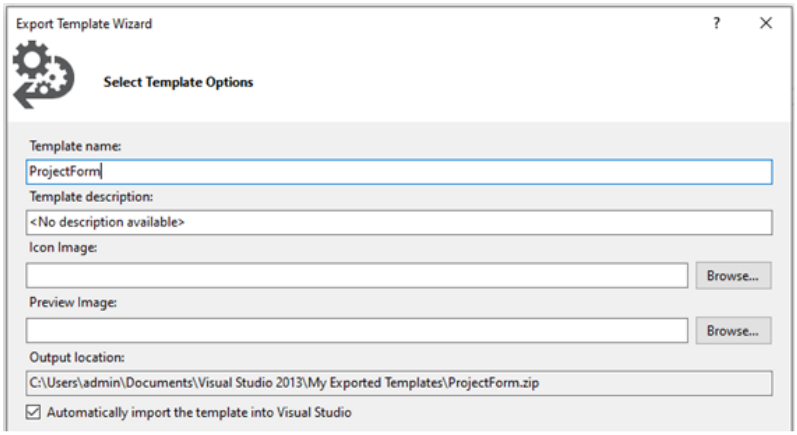

Рис. 132. Настройка параметров шаблона

Созданный шаблон будет размещен в папке *Шаблонов* (*My Exported Templates*) в рабочей папке проектов *Visual Studio*, также в перечне шаблонов проектов, на базе которых можно реализовывать новое решение, появится еще один вариант пользовательского шаблона.

Создавая новый проект в *IDE Visual Studio* теперь можно выбрать готовый проект, содержащий форму. Создав проект на базе этого нового шаблона, проверьте настройки компоновщика в части подсистемы компоновщика (*SubSystem*) и входной точки (`Main`).

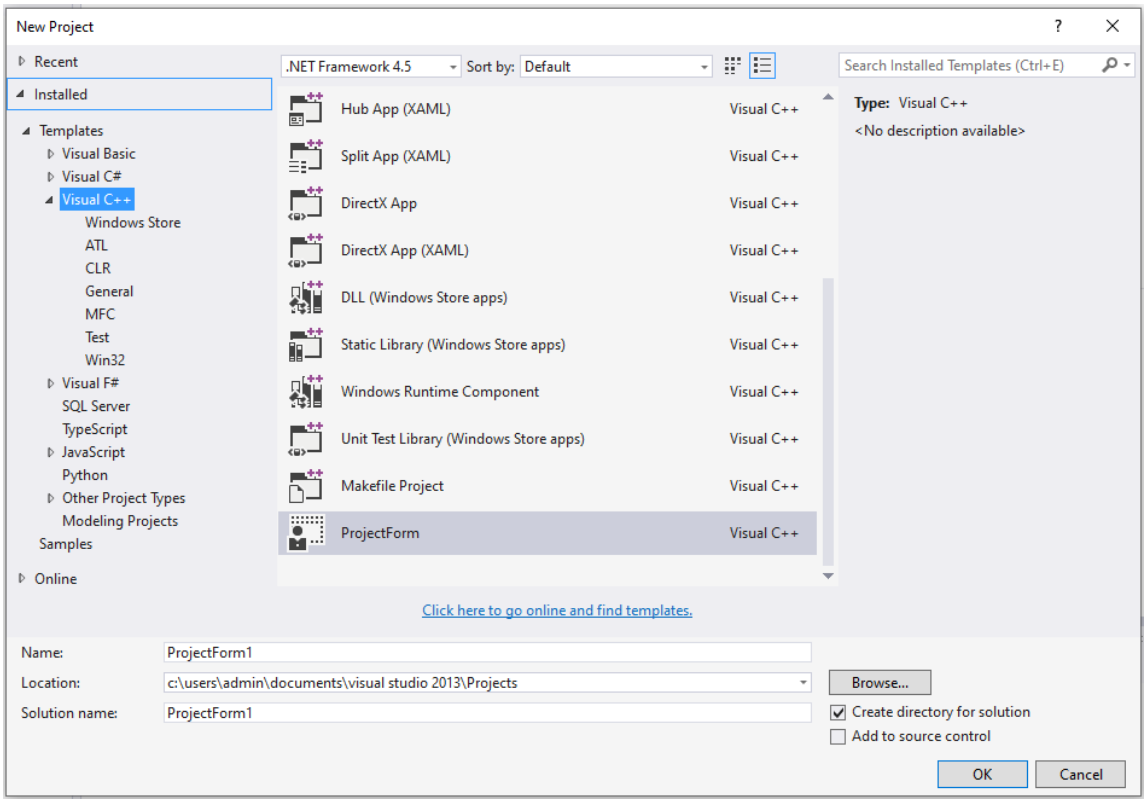

Рис. 133. Создание нового проекта на базе созданного шаблона проекта

13.3. Работа с визуальными объектами

Вернемся к исходному проекту и продолжим работу с формой `form`. Для редактирования внешнего вида формы, можно перейти во вкладку *Конструктора формы* (`MyForm.h [Design]`), кликнув дважды по файлу `MyForm.h` в обозревателе решений.

Для понимания структуры проекта и принципа работы создаваемого GUI-приложения нужно проанализировать файл `MyForm.cpp` (см. *Листинг 156*). Из листинга видно, что программа `Main()` всего лишь в приложении `Application` вызывает две подпрограммы настройки параметров `EnableVisualStyles()` и `SetCompatibleTextRenderingDefault(false)` и создает объект `form` класса `MyForm`, после чего «запускает» форму `Application::Run(%form)`.

Таким образом, приложение работает с одним динамическим объектом – ***экземпляром класса*** `MyForm`. Что же он из себя представляет? Это можно видеть, изучив заголовочный файл `MyForm.h`.

Листинг 157

```
public ref class MyForm : public System::Windows::Forms::Form
{
public:
	MyForm(void)
	{
		InitializeComponent();
		//
		//TODO: Add the constructor code here
		//
	}
protected:
	~MyForm()
	{
		if (components)
		{
```

```
                delete components;
            }
        }
private:
        System::ComponentModel::Container^ components;
#pragma region Windows Form Designer generated code
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        void InitializeComponent(void)
        {
            this->SuspendLayout();
            // MyForm
            this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
            this->AutoScaleMode =
                System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(284, 261);
            this->Name = L"MyForm";
            this->Text = L"MyForm";
            this->ResumeLayout(false);
        }
#pragma endregion
};
```

Итак, наш класс `MyForm` является потомком стандартного класса `System::Windows::Forms::Form`, таким образом наследуя все свойства, методы и дружественные функции стандартного класса `Form`.

Помимо этого, реализовано три метода – конструктор, деструктор и метод `InitializeComponent()`, запускаемый из конструктора, инициирующий начальные параметры (`Properties`) объекта. Больше ничего в этом классе пока нет. Как же работает программа?

Наше приложение вызывает наследуемые от предков методы - *обработчики событий* (`Events`), отслеживаемые операционной системой. Если в фокусе визуального компонента происходит какое-то событие (щелчок мыши, нажатие клавиши, движение пера, перерисовка экрана и т.п.), то операционная система реагирует на это, сообщая нашему приложению, что данное событие произошло. Визуальный объект (сама форма `form` или визуальные компоненты на ней расположенные) вызывает обработчик этого события (свой или своего предка) и выполняет его.

Продемонстрируем это. Нажатие правой кнопкой мыши на визуальном объекте (у нас пока это только форма `form` – объект класса `MyForm`) позволяет вызвать список свойств (`Properties`) этого объекта. Меняя эти свойства, мы будем менять вид визуального компонента (*Рис. 134*).

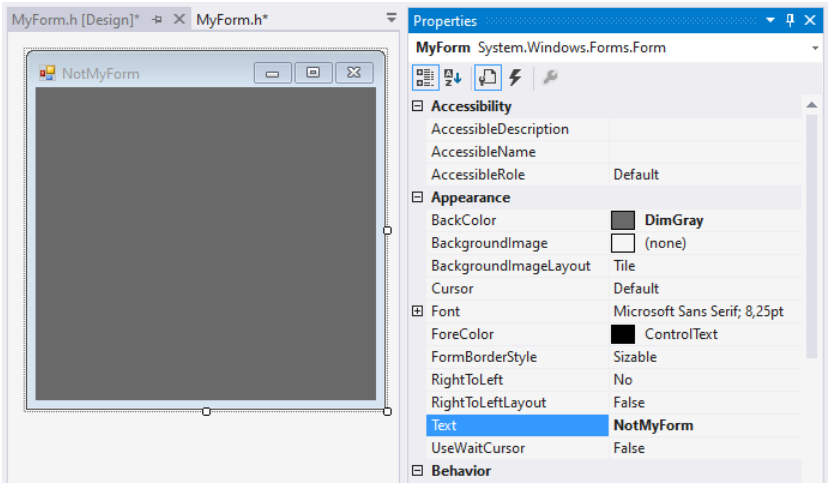

Рис. 134. Список параметров (Properties) формы

Например, на рисунке выше я изменил цвет фона `BackColor` нашей формы на более темный – `DimGray` и текст, выводимый на форме изменил на `"NotMyForm"`. В этом же окне редактора свойств визуального компонента (`Properties`), нажав на вкладку с символом черной молнии, можно переключиться на список событий (`Events`), которые будет обрабатывать наша форма (*Рис. 135*).

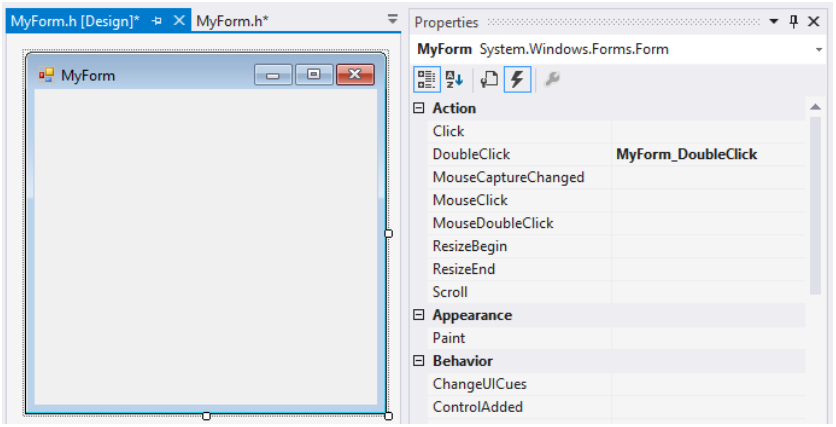

Рис. 135. Список событий формы

Изначально список событий пуст – наша форма ничего не делает, однако, если нажать мышкой на пустое место справа от требуемого события, автоматически будет сгенерирован метод класса `MyForm` – обработчик события и вставлен в листинг программы в части описания формы в файле `MyForm.h`. В нашем примере я нажал на событие `DoubleClick` и Visual C++ сгенерировал обработчик этого события `MyForm_DoubleClick()`, как видно на *Рис. 135*, *Рис. 136*.

Если теперь переключиться на листинг файла `MyForm.h`, то можно видеть (см. *Рис. 136*), что в нижней части описания методов добавилась строка:

```
this->DoubleClick += gcnew System::EventHandler(this, &MyForm::MyForm_DoubleClick);
```

и пустой пока шаблон обработчика события.

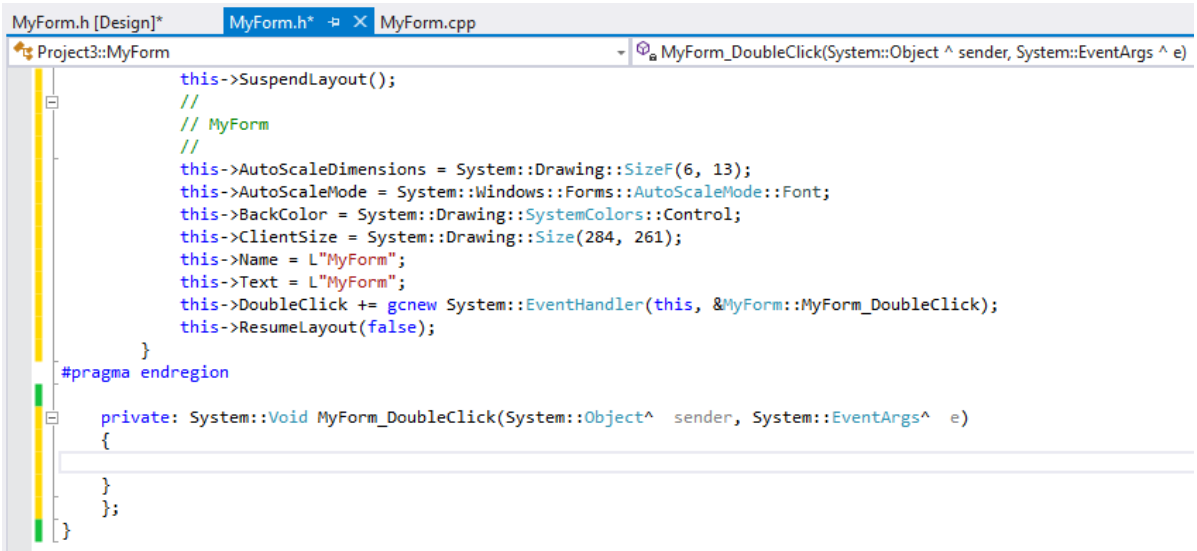

Рис. 136. Список событий (Events) формы

В этом пустом методе можно написать алгоритм тех действий, которые будут выполнены в ответ на появление в фокусе нашей формы `form` события `DoubleClick`, т.е. двойного щелчка мышкой. Пусть это будет изменение цвета фона формы, надписи на ней и ее размера, как показано на примере (*Листинг 158*).

Листинг 158

```
private: System::void MyForm_DoubleClick(System::Object^ sender, System::EventArgs^)
{
  this->Text = L"Hellow, World!";
  this->ClientSize = System::Drawing::Size(284, 400);
  this->BackColor = System::Drawing::SystemColors::ActiveCaption;
}
```

Проверьте как работает программный код из *Листинг 158* самостоятельно подставив его в нужное место программы.

13.4. Добавление новых визуальных компонентов

Для того, чтобы добавить на нашу форму новые элементы, понадобится панель *Инструменты* (*Toolbox*), см. *Рис. 137*, в которой содержатся все доступные классы визуальных компонентов GUI. Их можно располагать на нашей форме.

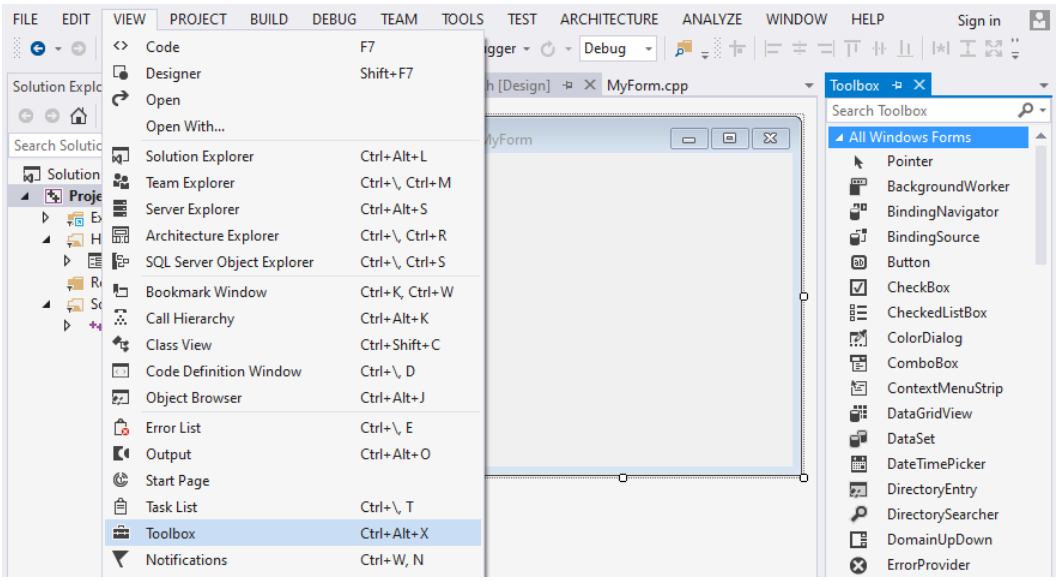

Рис. 137. Добавление визуального объекта на форму

Самостоятельно проследите, какой текст будет автоматически добавлен в описание класса `MyForm` в файле `MyForm.h`. после добавления на форму, например, кнопки (`Button`).

Видно, что в процедуру инициализации `InitializeComponent()` конструктора добавилась строка создания динамического объекта – кнопки (`Button`):

```
this->button1 = gcnew System::Windows::Forms::Button()
```

Здесь оператор gcnew это некоторый аналог оператора new, создающий динамический объект класса, выделяющий память под него, возвращающий его адрес и запускающий соответствующий конструктор.

Созданный `button1` – это тоже объект визуального класса `System::Windows::Forms::Button`, поэтому у него есть свой набор свойств, методов и обработчиков событий. В эту же процедуру инициализации (в конструктор) добавлен блок первоначальной настройки параметров объекта `button1`. Для изменения свойств только что созданного элемента интерфейса можно точно так же щёлкнуть на нём правой кнопкой и в контекстном меню выбрать, соответственно, `Properties`. Но это будут свойства объекта `button1`.

Испытаем кнопку в работе (*Листинг 159*): сделаем так, чтобы при её нажатии появлялось окно с приветственным текстом (см. *Рис. 138*). Для этого создадим

обработчик события – нажатие на кнопку – `button1_Click()`, и добавим в него следующий текст, выводящий сообщение:

Листинг 159

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
  MessageBox::Show("Hello World", "My heading",
                   MessageBoxButtons::OKCancel, MessageBoxIcon::Asterisk);
}
```

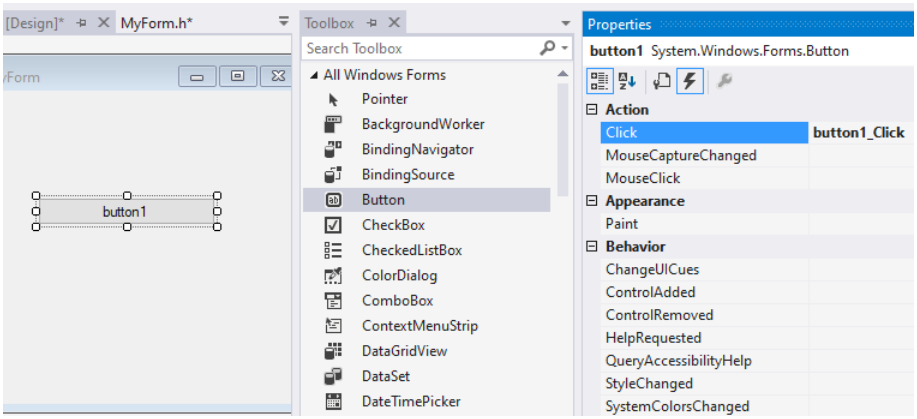

Рис. 138. Обработчик события Click визуального объекта button1

На рисунке (*Рис. 139*) приведен результат работы события – нажатие на кнопку `button1`. При этом срабатывает обработчик события `button1_Click()`, а старый обработчик события `MyForm_DoubleClick()` – не запускается, если кликать строго по кнопке, но на двойной щелчок мышкой по другим частям формы он реагирует как положено.

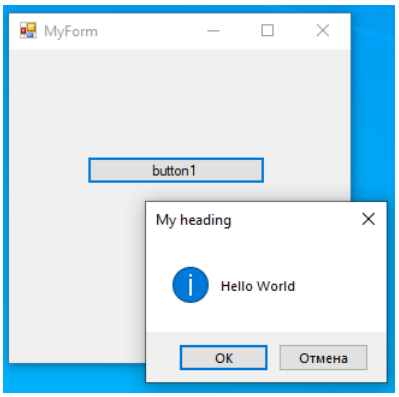

Рис. 139. Реакция приложения на событие Click по кнопке button1

13.5. Обзор основных элементов Windows Forms и их свойств

`Windows Forms` предоставляет широкий спектр элементов, которые условно можно разделить на два типа: интерфейсные – те которые видны пользователю и с которыми он может работать непосредственно (разного рода кнопки, панели, таблицы) и служебные – те, что выполняют определенные задачи и вызываются путем взаимодействия пользователя с интерфейсными элементами (диалоги, таймеры, адаптеры).

Перед использованием элемент должен быть помещен на форму (обычно методом *drag&drop* или как его называют «*перетаскивание*»), за исключением самого элемента формы.

Все элементы `Windows Forms` находятся на панели *Toolbox* (*Рис. 140*).

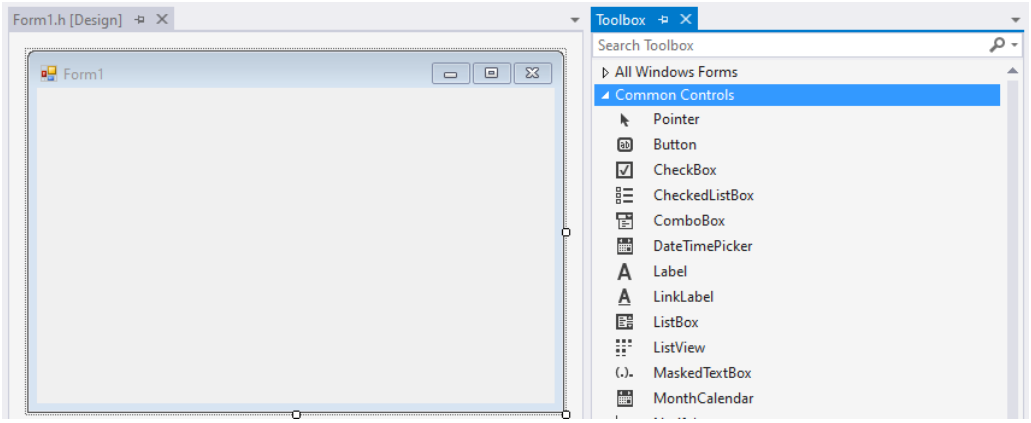

Рис. 140. Панель элементов (Toolbox)

Каждый элемент представляет собой *класс*, наследуемый от единого корня, содержит свой набор *свойств* (`Properties`) и *событий* (`Events`). Большинство из них повторяются для всех элементов (являются универсальными).

Свойство в объектно-ориентированном программировании – способ доступа к внутреннему состоянию объекта, имитирующий переменную некоторого типа.

Событие в объектно-ориентированном программировании – это сообщение операционной системы, которое может возникнуть в различных точках исполняемого кода при выполнении определённых условий.

Обработчики событий – это методы класса ООП (или дружественные функции), предназначенные для того, чтобы иметь возможность предусмотреть реакцию программного обеспечения на события ОС.

Все свойства элемента находятся на панели `Properties` и меняются вручную или выбираются из списка (*Рис. 141, а*), а события на панели `Properties`, вкладка `Events` (*Рис. 141, б*).

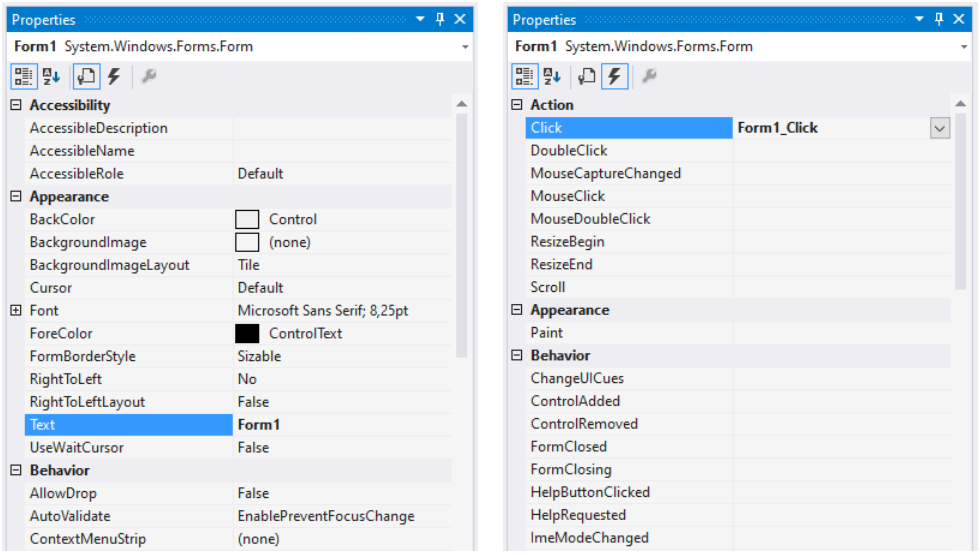

Рис. 141. Свойства элемента (а), события элемента (б)

Элемент Form

Представляет окно или диалоговое окно, которое составляет пользовательский интерфейс приложения (*Рис. 134, Рис. 135*). Приложение может содержать и несколько форм, но какой-то из них должна принадлежать стартовая точка – программа `Main()`, и это – основная форма.

Ниже в таблице (*Таблица 14*) приводятся основные свойства элемента `Form` и его основные события (*Таблица 15*). Понятно, что и свойств, и событий значительно больше.

Полный их список можно увидеть в документации на объект Form и здесь приводить его нет смысла.

Таблица 14 Основные свойства элемента Form

Text	Пользовательское название. Текст, выводимый в заголовке формы
Visible	Получает или задает значение, указывающее, видимый ли этот элемент управления и все его дочерние элементы управления.
WindowState	Возвращает или задаёт значение, указывающее на состояние формы: развёрнутое, свернутое или обычное.
FormBorderStyle	Получает или задает стиль границы формы. Может принимать значения: None – форма не имеет границ; FixedSingle – обычная форма без возможности масштабирования; Sizable – обычная форма с возможностью масштабирования.
MinimizeBox	Возвращает или задает значение, указывающее, отображается ли кнопка свертывания в строке заголовка формы.
MaximizeBox	Возвращает или задает значение, указывающее, отображается ли кнопка развертывания в строке заголовка формы.

Таблица 15 Основные события элемента Form

Click	Происходит при щелчке элемента управления.
Load	Происходит перед первоначальным отображением формы.
Activated	Переключение с какого-то другого визуального объекта на данную форму
Deactivate	Переключение с данной формы на какой-то другой объект, не на ней расположенный

При наведении мышки на какое-то из свойств или событий в окне Properties, в нижней части экрана, как правило, дается краткий комментарий данным свойствам или событиям. Более детально, можно ознакомиться с ними в окне помощи (Help) или в документации.

Элемент Button

Элемент управления Windows *Кнопка*. В наследуемых свойствах этого объекта содержатся возможности визуального и анимированного представления нажатия, отристовки кнопок Windows. Находится в группе элементов Common Controls (*Рис. 142*).

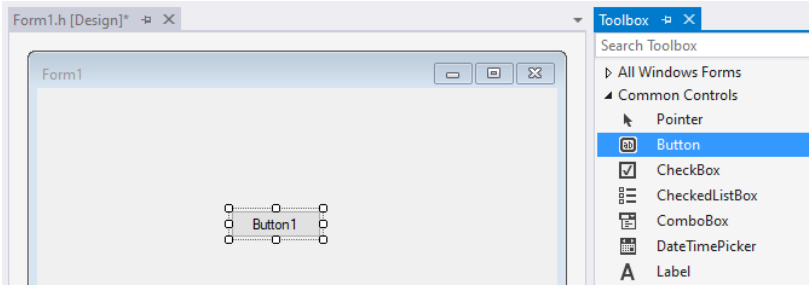

Рис. 142. Элемент Button

Таблица 16 Основные свойства элемента Button

Text	Получает или задает текст, сопоставленный с этим элементом управления.
TextAlign	Получает или задает выравнивание текста в элементе управления.
Image	Возвращает или задаёт изображение, отображаемое на элементе управления.

Visible	Получает или задает значение, указывающее, отображается ли элемент управления.
Enabled	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
Name	Возвращает или задает имя элемента управления. Стандарты те же что и для переменных C++.

Таблица 17 Основные события элемента Button

Click	Происходит при щелчке элемента управления.
DoubleClick	Происходит при двойном щелчке мышью элемента управления Button.

Элемент Label

Элемент *Метка* (Label) представляет стандартную надпись Windows – любой текст или изображение, выводимые на форму. Этот элемент позволяет настраивать практически все свойства текста, отдельных символов и изображений, присущие Windows. Находится в группе элементов Common Controls (*Рис. 143*).

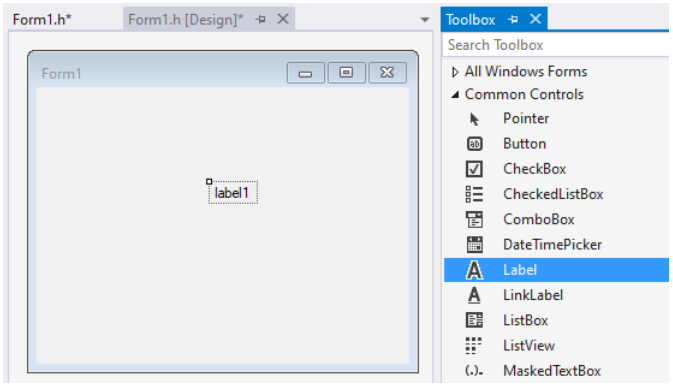

Рис. 143. Элемент Метка (Label)

Таблица 18 Основные свойства элемента Label

Text	Получает или задает текст, сопоставленный с этим элементом управления.
TextAlign	Получает или задает выравнивание текста в элементе управления Label.
Image	Возвращает или задаёт изображение, отображаемое на элементе управления Label.
Visible	Получает или задает значение, указывающее, отображается ли элемент управления.
Enabled	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
Name	Возвращает или задает имя элемента управления. Стандарты те же что и для переменных C++.

Элемент TextBox

Элемент TextBox отображает элемент управления текстовым окном для ввода данных пользователем. Находится в группе элементов Common Controls (*Рис. 144*).

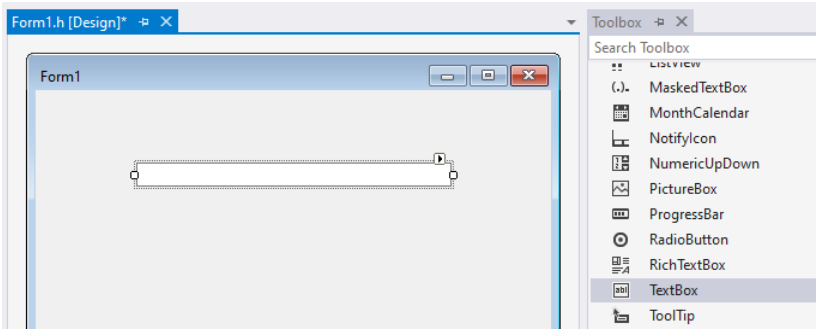

Рис. 144. Элемент TextBox

Таблица 19 Основные свойства элемента TextBox

`Text`	Получает или задает текст, сопоставленный с этим элементом управления.
`TextAlign`	Получает или задает выравнивание текста в элементе управления `TextBox`.
`MaxLength`	Получает или задает максимальное разрешенное число знаков в текстовом окне (по умолчанию `32 767` символов).
`Multiline`	Определяет поддерживает ли элемент многострочный текст.
`Enabled`	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
`Name`	Возвращает или задает имя элемента управления. Стандарты те же что и для переменных C++.
`Visible`	Получает или задает значение, указывающее, отображается ли элемент управления.
`ReadOnly`	Возвращает или задает значение, определяющее возможность изменения содержимого элемента управления `TextBox`.
`Rows`	Возвращает или задает число строк, отображаемых в многострочном текстовом окне.
`Columns`	Возвращает или задает ширину отображаемого текстового окна в знаках.

Таблица 20 Основные события элемента TextBox

`Click`	Происходит при щелчке элемента управления.
`DoubleClick`	Происходит при двойном щелчке мышью элемента управления `TextBox`.
`KeyDown`	Происходит при нажатии клавиши на клавиатуре, когда фокус находится на элементе `TextBox`.
`KeyPress`	Происходит при вжатом положении клавиши на клавиатуре, когда фокус находится на элементе `TextBox`.
`KeyUp`	Происходит при отпускании клавиши на клавиатуре, когда фокус находится на элементе `TextBox`.
`TextChanged`	Происходит при изменении содержимого текстового окна `TextBox`.

Во фрагменте программного кода, приведенном ниже (*Листинг 160*) на форме расположены два объекта – метка `label1` и окно для ввода текста – `textBox1`. В методе обработчике события «*текст в окне* `textBox1` *изменен*» – `System::Void textBox1_TextChanged()` происходит присваивание введенного в окне `textBox1` текста содержимому метки `label1`.

Листинг 160

```
private: System::Void textBox1_TextChanged(System::Object^ sender,
System::EventArgs^ e)
{
       label1->Text = textBox1->Text;

}
```

Таким образом, одна строчка программного кода позволяет мгновенно отображать на метке `label1` тот текст, который вводится в окне `textBox1` (*Рис. 145*). Проверьте работу обработчика событий (*Листинг 160*) самостоятельно.

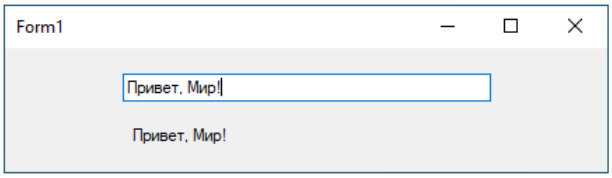

Рис. 145. Элемент Label отображает текст, введенный в окне элемента TextBox

Элемент RichTextBox

Элемент `TextBox` служит для ввода и редактирования коротких сообщений – не длиннее 32 тысяч символов – одной строки. Как же работать с более серьезным текстом? Для этого существует элемент `RichTextBox`.

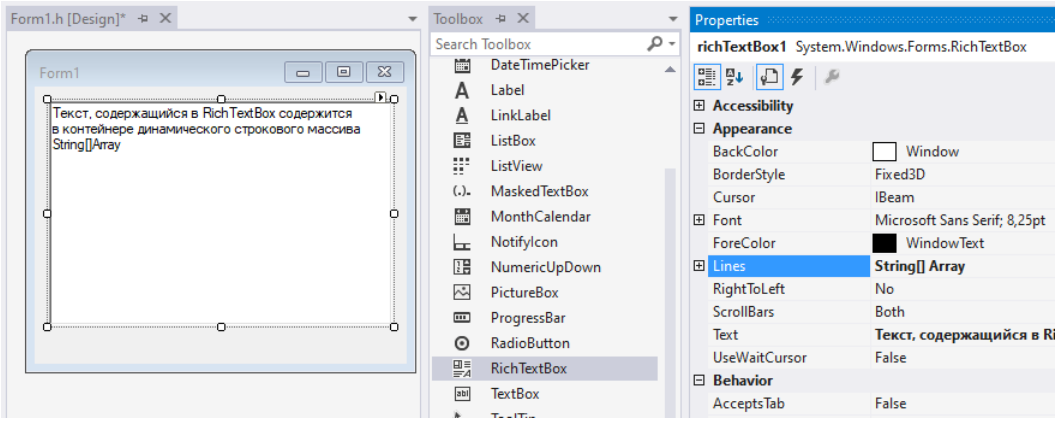

Рис. 146. Элемент RichTextBox и его свойства

Это практически готовый редактор форматированного текста, позволяющий работать с многострочным массивом `Lines` (*Рис. 146*). Предоставляет элемент управления полем *форматированного* текста `Windows`. Находится в группе элементов `Common Controls`.

Таблица 21 Основные свойства элемента RichTextBox

`Text`	Получает или задает текст, сопоставленный с этим элементом управления.
`Lines`	Массив строк текстового поля.
`MaxLength`	Получает или задает максимальное разрешенное число знаков в текстовом окне (по умолчанию `2 147 483 647` символов).
`Multiline`	Определяет поддерживает ли элемент многострочный текст.
`Enabled`	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
`Name`	Возвращает или задает имя элемента управления. Стандарты те же что и для переменных C++.

Visible	Получает или задает значение, указывающее, отображается ли элемент управления.
ReadOnly	Возвращает или задает значение, определяющее возможность изменения содержимого элемента управления RichTextBox.
Rows	Возвращает или задает число строк, отображаемых в многострочном текстовом окне.
Columns	Возвращает или задает ширину отображаемого текстового окна в знаках.
ScrollBars	Получает или задает тип полос прокрутки, отображающихся в элементе управления RichTextBox.

Таблица 22 Основные события элемента RichTextBox

Click	Происходит при щелчке элемента управления.
DoubleClick	Происходит при двойном щелчке мышью элемента управления RichTextBox.
KeyDown	Происходит при нажатии клавиши на клавиатуре, когда фокус находится на элементе RichTextBox.
KeyPress	Происходит при вжатом положении клавиши на клавиатуре, когда фокус находится на элементе RichTextBox.
KeyUp	Происходит при отпускании клавиши на клавиатуре, когда фокус находится на элементе RichTextBox.
TextChanged	Происходит при изменении содержимого текстового окна RichTextBox.

Элемент RadioButton

Представляет собой переключатель между множественными вариантами, который пользователь может устанавливать (выбирать), но не снимать (выбор отменяется автоматически при выборе другого варианта) – делается один выбор из нескольких.

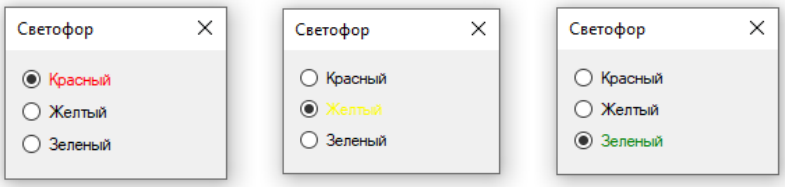

Рис. 147. Элементы RadioButton

В приведенном на примере (см. *Рис. 147*) на форме размещены три объекта RadioButton, для каждого создан обработчик событий, изменяющий цвет соответствующего текста. Реализацию обработки события для одного из объектов RadioButton иллюстрирует фрагмент программы, приведенный ниже (*Листинг 161*).

Таблица 23 Основные свойства элемента RadioButton

Text	Получает или задает текст, сопоставленный с этим элементом управления.
TextAlign	Получает или задает выравнивание текста в элементе управления RadioButton.
Image	Возвращает или задаёт изображение, отображаемое на элементе управления RadioButton.

Visible	Получает или задает значение, указывающее, отображается ли элемент управления.
Enabled	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
Checked	Возвращает или задаёт значение, указывающее, находится ли RadioButton во включённом состоянии.

Таблица 24 Основные события элемента RadioButton

Click	Происходит при щелчке элемента управления.
DoubleClick	Происходит при двойном щелчке мышью элемента управления RadioButton.
CheckedChanged	Возникает при переходе RadioButton во включённое состояние.

Листинг 161

```
private: System::Void radioButton3_CheckedChanged(System::Object^ sender,
System::EventArgs^ e)
{
	this->radioButton1->ForeColor = System::Drawing::SystemColors::ControlText;
	this->radioButton2->ForeColor = System::Drawing::SystemColors::ControlText;
	this->radioButton3->ForeColor = System::Drawing::Color::Green;

}
```

Обратите внимание на указатель this, он содержит адрес рабочей формы form1. В проекте может содержаться несколько форм, и в каждой могут находиться одинаковые визуальные объекты, поэтому нужен указатель this – указатель на каждый конкретный объект данного класса (*указатель на себя*).

Элемент CheckBox

Элемент CheckBox это такой элемент управления, который визуализирует выбор нескольких вариантов среди многих (*Рис. 148*). Каждый знак выбора (*флажок*) пользователь может устанавливать и снимать. Находится в группе элементов Common Controls.

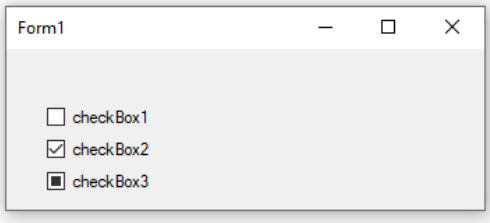

Рис. 148. Элемент CheckBox

Таблица 25 Основные свойства элемента CheckBox

Text	Получает или задает текст, сопоставленный с этим элементом управления.
TextAlign	Получает или задает выравнивание текста в элементе управления CheckBox.
Image	Возвращает или задаёт изображение, отображаемое на элементе управления CheckBox.
Visible	Получает или задает значение, указывающее, отображается ли элемент управления.

Enabled	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
Name	Возвращает или задает имя элемента управления. Стандарты те же что и для переменных C++.
Checked	Возвращает или задаёт значение, указывающее, находится ли CheckBox во включённом состоянии.
CheckState	Возвращает или задаёт состояние CheckBox. Помимо отмеченного и пустого может принимать еще и всегда отмеченное состояние.

Таблица 26 Основные события элемента CheckBox

Click	Происходит при щелчке элемента управления.
DoubleClick	Происходит при двойном щелчке мышью элемента управления CheckBox.
CheckedChanged	Возникает при переходе CheckBox во включённое состояние.

Элемент numericUpDown

Представляет регулятор Windows (также известный как элемент управления "вверх-вниз"), отображающий числовые значения, позволяющий, кроме того, вводить числа и с клавиатуры (*Рис. 149*). Находится в группе элементов Common Controls.

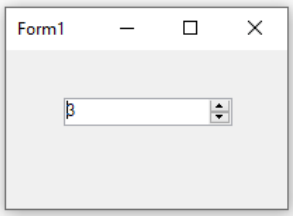

Рис. 149. Элемент numericUpDown

Таблица 27 Основные свойства элемента numericUpDown

Value	Возвращает или задает значение, присвоенное счетчику.
Visible	Получает или задает значение, указывающее, отображается ли элемент управления.
Enabled	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
Name	Возвращает или задает имя элемента управления. Стандарты те же что и для переменных C++.
Minimum	Возвращает или задает минимальное допустимое значение для счетчика.
Maximum	Возвращает или задает максимальное значение для счетчика.
Increment	Возвращает или задает значение для увеличения или уменьшения счетчика.
ReadOnly	Возвращает или задает значение, определяющее возможность изменения значений счетчика, путем ввода с клавиатуры.

Таблица 28 Основные события элемента numericUpDown

Click	Происходит при щелчке элемента управления.
DoubleClick	Происходит при двойном щелчке мышью элемента управления numericUpDown.
ValueChanged	Возникает при изменении значений счетчика.

Элемент DataGridView

Элемент, который отображает данные в табличном виде (см. *Рис. 150*).

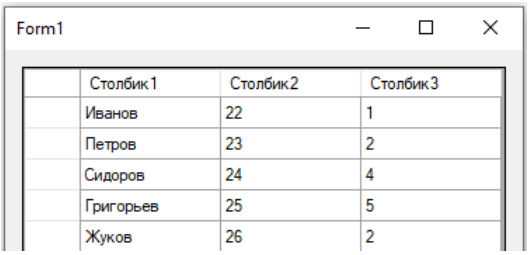

Рис. 150. Элемент DataGridView

Таблица 29 Основные свойства элемента DataGridView

`Visible`	Получает или задает значение, указывающее, отображается ли элемент управления.
`Enabled`	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
`Name`	Возвращает или задает имя элемента управления. Стандарты те же что и для переменных C++.
`Columns`	Коллекция столбцов `DataGridView`. Может быть создана вручную, при помощи специального окна свойств или создаваться программно. Каждая колонка может быть: текстовым полем, выпадающим списком, `CheckBox`-ом, кнопкой, картинкой или ссылкой.
`ReadOnly`	Возвращает или задает значение, определяющее возможность изменения содержимого элемента управления `DataGridView`.
`ScrollBars`	Получает или задает тип полос прокрутки, отображающихся в элементе управления `DataGridView`.
`ColumnHeadersVisible`	Возвращает или задает значение, указывающее, отображается ли строка заголовка столбца.
`RowHeadersVisible`	Возвращает или задает значение, указывающее, отображается ли столбец, содержащий заголовки строк.

Каждая колонка имеет ряд собственных свойств, общих для всего столбца данных, эти свойства могут быть настроены в специально вызванном редакторе свойств (см. *Рис. 151*).

Таблица 30 Основные свойства колонок (Columns) элемента DataGridView

`Visible`	Получает или задает значение, указывающее, отображается ли элемент управления.
`HeaderText`	Возвращает или задает значение текста заголовка колонки.
`Resizable`	Возвращает или задает значение, определяющее возможность пользовательского изменения ширины колонки.
`Frozen`	Возвращает или задает значение, определяющее возможность движения колонки при прокрутке элемента `DataGridView`.
`Width`	Возвращает или задает значение ширины колонки.
`ReadOnly`	Возвращает или задает значение, определяющее возможность изменения содержимого колонки элемента управления `DataGridView`.

Таблица 31 Основные события элемента DataGridView

`Click`	Происходит при щелчке элемента управления.
`CellEndEdit`	Возникает после завершения редактирования ячейки.

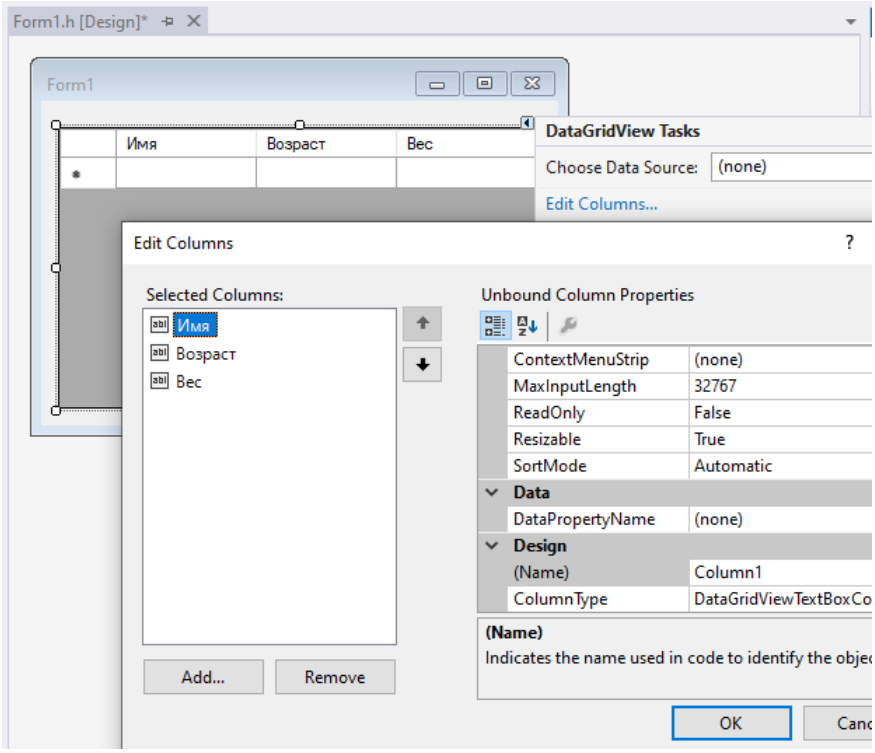

Рис. 151. Настройка свойств коллекции столбцов (Columns) элемента DataGridView

Элемент Chart

Элемент `Chart` используется для построения разного вида диаграмм и графиков. Находится в группе элементов `Data` (*Рис. 152*).

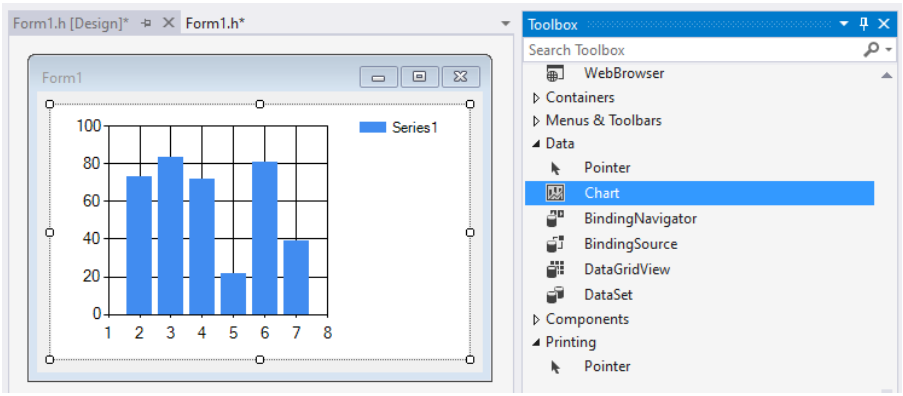

Рис. 152. Элемент Chart

Таблица 32 Основные свойства элемента Chart

`Visible`	Получает или задает значение, указывающее, отображается ли `Chart`
`Enabled`	Возвращает или задает значение, показывающее, сможет ли `Chart` отвечать на действия пользователя.
`Palette`	Получает или задает палитру для элемента управления `Chart`.
`Series`	Получает объект `SeriesCollection`, который содержит объекты `Series`. `SeriesCollection` – разного рода графики и диаграммы присутствующие на одном элементе `Chart`.

Titles	Объект TitleCollection, используемый для хранения всех объектов Title, используемых элементом управления Chart. Содержит коллекцию заголовков диаграмм.

Значительная часть параметров элемента управления Chart содержит оформительские свойства, а содержательная часть графиков размещается в хранилищах (Collection) – массивах данных, важнейшие из которых: Annotations (*аннотации, названия данных*), ChartAreas (*диапазоны, границы изменения данных*), Legends (*подписи данных*), Titles (*заголовки*) и самый важный – Series (*данные*) – см. *Рис. 153*.

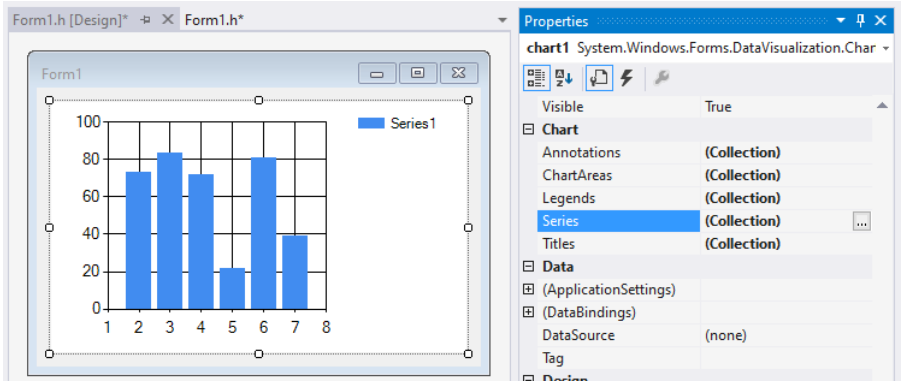

Рис. 153. Свойства элемента Chart, хранилище (Collection) данных (Series)

Каждое хранилище (Collection) данных (Series) представляет собой полноценный класс, содержащий свои соответствующие свойства (Properties) и события (Events). На *Рис. 154* показано, как можно вызвать отдельное окно настроек каждой последовательности данных (графика, диаграммы и пр.). Здесь самыми важными свойствами являются ChartType (*тип графика*) и Points (*последовательность точек*).

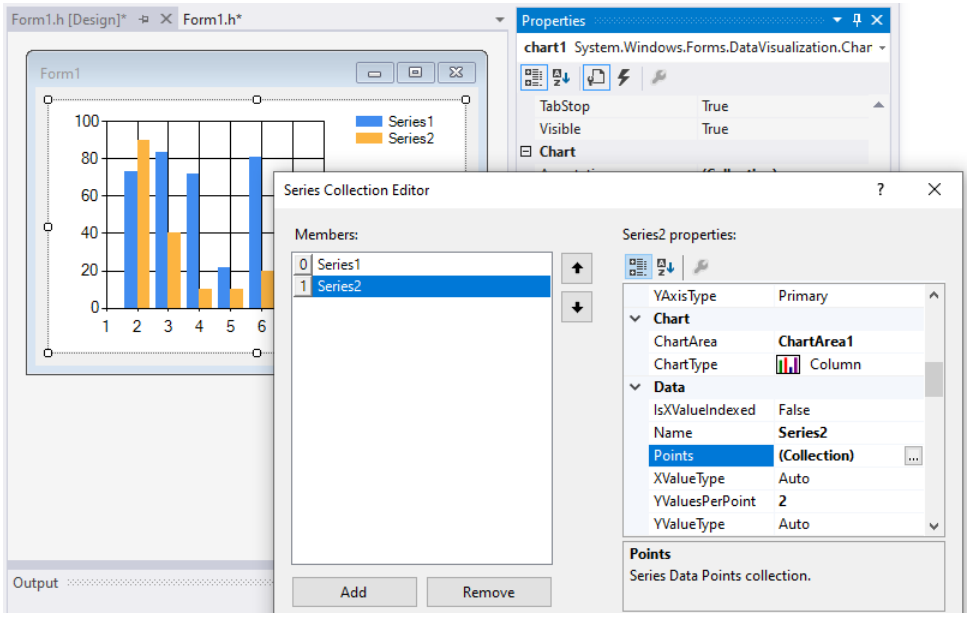

Рис. 154. Редактирование свойств последовательности данных Series

Содержательная часть каждой последовательности данных *SeriesCollection* расположена в свойстве Points – последовательность (Collection) точек данных (Point), для каждой из которых также можно вызвать редактор свойств, как показано на *Рис. 155*.

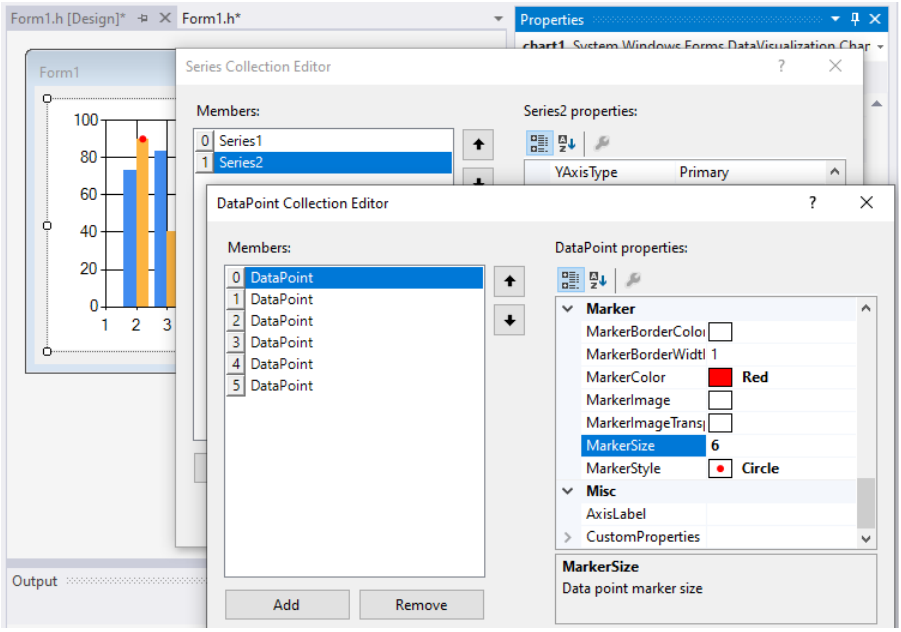

Рис. 155. Редактирование свойств последовательности точек Points

В этом редакторе можно настроить свойства каждой отдельной точки графика (`Point`), так как она тоже является классом визуальных компонент – можно настроить размер, форму, цвет, видимость и пр. каждой точки.

Как и для любых визуальных компонентов, свойства объекта `Chart` можно изменять в редакторе свойств (`Properties`), а можно непосредственно в теле методов в описании соответствующего класса в файле `Form1.h`.

Элемент GroupBox

Довольно часто требуется объединить визуальные компоненты в группу, для этого существует элемент управления `GroupBox`. Этот объект отображает рамку вокруг группы элементов управления и, необязательно, заголовок над ней, объединяя их. Находится в группе элементов `Containers` (*Рис. 156*).

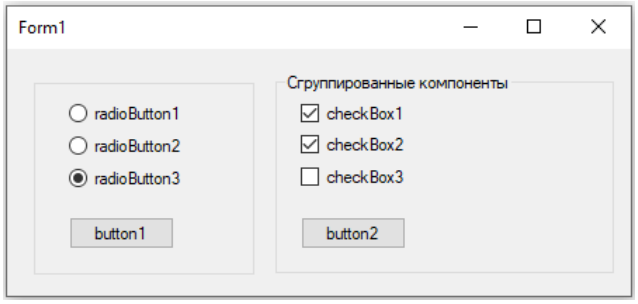

Рис. 156. Элемент GroupBox

Таблица 33 Основные свойства элемента GroupBox

`Text`	Получает или задает текст, сопоставленный с этим элементом управления.
`Visible`	Получает или задает значение, указывающее, отображается ли элемент управления.
`Enabled`	Возвращает или задает значение, показывающее, сможет ли элемент управления отвечать на действия пользователя.
`Name`	Возвращает или задает имя элемента управления. Стандарты те же что и для переменных C++.

Основная цель объекта класса GroupBox в объединении компонентов: после того, как компоненты помещены в панель GroupBox, некоторые их свойства объединяются. Например, чтобы изменить видимость (Visible) или доступность (Enable) всей группы визуальных компонентов нужно изменить эти свойства только у объекта GroupBox. Пример этого можно видеть на обработчиках событий Click (*Листинг 162*).

Листинг 162

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
       this->groupBox2->Visible = !(this->groupBox2->Visible);
}
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e)
{
       this->groupBox1->Visible = !(this->groupBox1->Visible);
}
```

MessageBox

Отображает появляющееся на экране окно сообщения MessageBox, в котором могут содержаться текст, кнопки и символы, которые информируют пользователя и дают ему указания (*Листинг 163*).

Основной метод этого класса:

```
Show(String, String) // отображает окно сообщения с указанными текстом и заголовком
```

Пример использования приводится ниже:

Листинг 163

```
if (!Double::TryParse(Convert::ToString(GridArray->CurrentCell->Value), item))
{
    MessageBox::Show("Введите действительное число","Ошибка");
}
```

Пример: Построение графика функции

Задача: создать визуальное приложение, при помощи которого можно строить график заданной наперед функции одной переменной *y=x*sin(x)*.

Создаем проект Windows Forms. Помещаем на форму элемент Chart, который будет служить окном для отображения графика. Изменим тип графика из гистограммы, на сглаженный график функции (в Series1 значение свойства ChartType из Column изменить на Spline), добавив в метод инициализации компонентов строчку:

```
series1->ChartType =
System::Windows::Forms::DataVisualization::Charting::SeriesChartType::Spline;
```

Здесь же свойству Name графику Series1 установим значение y(x)=x*sin(x), тем самым зададим легенду нашего графика, добавив строку:

```
series1->Name = L"x*Sin(x)";
```

Добавим два обработчика событий – события Load основной формы form1 и события Click графика chart1. После этих действий к тексту метода InitializeComponent() добавятся строки, приведенные ниже *Листинг 164*.

Листинг 164

```
void InitializeComponent(void)
{
       //. . . добавляем к настройкам, сделанным автоматически
```

```
        //
        // Form1: название формы
        this->Text = L"Построение графика";
        // Form1: обработчик события Load
        this->Load += gcnew
              System::EventHandler(this, &Form1::Form1_Load);
        // chart1: настройка линейного графика сплайнами
        series1->ChartType = System::Windows::Forms::DataVisualization::
              Charting::SeriesChartType::Spline;
        // chart1 series1: название серии
        series1->Name = L"x*Sin(x)";
        // chart1: обработчик события Click
        this->chart1->Click += gcnew
              System::EventHandler(this, &Form1::chart1_Click);
}
```

Событие `Load` нашей главной формы содержит следующий программный код (*Листинг 165*), смысл которого в настройках масштабирующих параметров графика по осям `X` и `Y` соответственно. Масштабирование будет происходить, если пользователь мышкой выделит область на графике, которую необходимо увеличить. Визуальный компонент `chart` – предок нашего графика `chart1` имеет методы, позволяющие производить масштабирование.

Листинг 165

```
private: System::Void Form1_Load(System::Object^ sender, System::EventArgs^ e)
{
        //масштабирование по оси X
        chart1->ChartAreas[0]->AxisX->ScaleView->Zoom(0, 100);
        // задаем область масштабирования
        chart1->ChartAreas[0]->CursorX->IsUserEnabled = true;
        //даем пользователю право масштабировать
        chart1->ChartAreas[0]->CursorX->IsUserSelectionEnabled = true;
        //даем пользователю право выбирать интервал масштабирования на графике
        chart1->ChartAreas[0]->AxisX->ScaleView->Zoomable = true;
        //включаем собственно режим масштабирования по оси X
        chart1->ChartAreas[0]->AxisX->ScrollBar->IsPositionedInside = true;
        //добавляем полосы прокрутки
        //аналогично для оси Y
        chart1->ChartAreas[0]->AxisY->ScaleView->Zoom(-100, 100);
        chart1->ChartAreas[0]->CursorY->IsUserEnabled = true;
        chart1->ChartAreas[0]->CursorY->IsUserSelectionEnabled = true;
        chart1->ChartAreas[0]->AxisY->ScaleView->Zoomable = true;
        chart1->ChartAreas[0]->AxisY->ScrollBar->IsPositionedInside = true;
}
```

Для построения координат точек `Points` непосредственно графика построим цикл от `0` до `100` по координате `X`, вычисляющий координаты `Y` по формуле $y=x*sin(x)$ и добавляющий точки в серию `Series[0]` при помощи метода `AddXY()`. Этот код (*Листинг 166*) поместим в обработчик события `Click` нашего элемента `chart1`:

Листинг 166

```
private: System::Void chart1_Click(System::Object^ sender, System::EventArgs^ e)
{
        for (int x = 0; x < 100; x++)
        {   //строим по точкам график, согласно формуле
              chart1->Series[0]->Points->AddXY(x, x * Math::Sin(x));
        }
}
```

Заметим, что метод `AddXY()` не является методом построения графика, а лишь добавляет точку с указанными координатами в массив точек, данный метод применим ко всем типам диаграмм. График по заполненному массиву точек строится автоматически. Результаты работы показаны на *Рис. 157*, после масштабирования на *Рис. 158*.

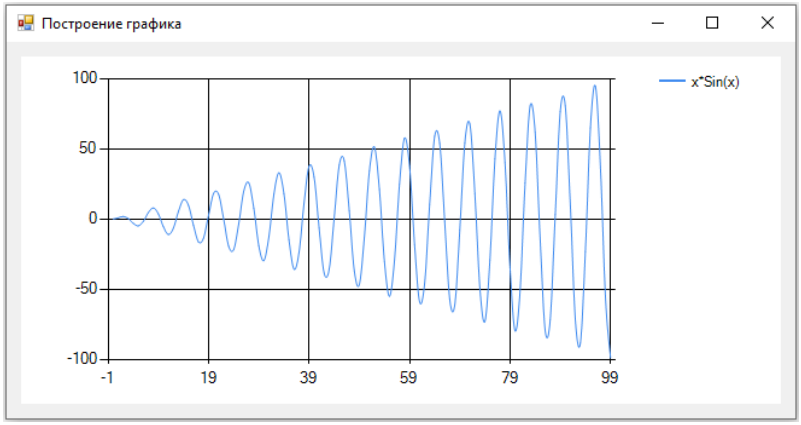

Рис. 157. Результат построения графика заданной функции

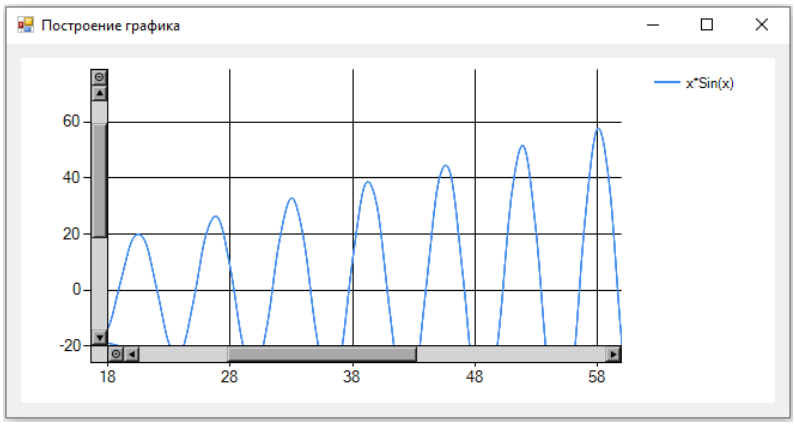

Рис. 158. Результат масштабирования графика

Мы убедились, что GUI (*Graphical User Interface*), реализуемый в среде *Visual C++* организован в виде иерархии наследуемых классов. GUI довольно «дружественный» и интуитивно понятен. Более глубокого понимания процессов, реализуемых в GUI, можно достичь, изучив технологию объектно-ориентированного программирования (ООП).

Заключение

В этой части учебного пособия мы познакомились с основами программирования на C++, в частности изучили процесс создания программного кода в среде Visual Studio, освоили структуру C++ программы, изучили стандартные типы данных языка C++. Освоено явное и неявное преобразование типов данных, изучены способы форматного и потокового ввода-вывода. Освоено умение работать с константами и перечислимым типом данных enum.

Алгоритмические конструкции языка C++ изучались во 2 главе данного пособия. Исследованы операторы выбора, цикла, операторы прерывания и безусловного перехода. Для построения логических выражений изучены переменные логического типа. Рассмотрен пример организации диалога с пользователем.

Третья глава посвящена указателям и ссылочным переменным. Рассмотрены типизированные и нетипизированные указатели, изучены способы статического и динамического распределения памяти. Рассмотрен пример с генерацией случайных чисел. Освоены принципы работы с константными указателями и ссылками.

Подпрограммы рассматривались в 4 главе. Рассмотрены три основных способа передачи параметров в тело функции. Изучена логика перегрузки и маскирования функций. Детально разобран процесс отладки программ в среде Visual Studio. Рассмотрены механизмы трассировки программного кода и контроля значений переменных в окне watch.

Пятая и шестая главы рассматривают массивы языка C++ и указатели на них. Рассматриваются статические и динамические одномерные и двумерные массивы и способы работы с ними. Особое внимание уделяется двумерным динамическим массивам, хранящимся в памяти в виде массива указателей или в виде одномерного массива.

Седьмая глава изучает строки в C++ в из стандартном исходном представлении – как массив символов. Изучаются статические и динамические строки, производится знакомство со строковыми операциями и с библиотекой <string.h>. Отдельно рассматриваются функции преобразования типов.

Работа с файлами и файловые операции изучаются в восьмой главе. Работа с файлами производится посредством альтернативных библиотек <stdio> и <fstream>.

Девятая глава посвящена структурам языка C++. Здесь изучается механизм конструирования собственных типов данных и работы с ними. Рассматриваются указатели на структуру и структуры, включающие в свой состав динамические массивы. Рассматривается динамический массив структур.

В десятой главе разбираются специальные структурные типы данных – битовые поля и объединения. На базе этих знаний в 11 главе изучаются побитовые методы работы с данными, логические и сдвиговые операции с разрядами.

Введение в классы, рассматриваемое в 12 главе, дает понятие классам, объектно-ориентируемому программированию, технике ограничения доступа к данным (Set и Get методы). Рассматриваются функции конструктора и деструктора, демонстрируется перегрузка операторов и использование дружественных функций. Студенты знакомятся с методами отделения интерфейса от реализации класса. Здесь же студенты знакомятся с понятием наследования.

Применение ООП к реализации графического интерфейса пользователя (GUI) демонстрируется в последней – 13 главе. Подробно рассматривается механизм создания проекта Windows Forms в Visual Studio на C++ и на его основе создается шаблон такого проекта. В части работы с визуальными объектами разбираются основные элементы Windows Forms и их свойства.

В завершении приводится список рекомендуемой литературы.

Список рекомендуемой литературы

1. Страуструп Бьярне. 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. Страуструп Бьярне. Язык программирования C++. Краткий курс, 2-е изд.: Пер. с англ. - СПб.: ООО "Диалектика", 2019. - 320 с.: ил. - Парал. тит. англ. ISBN 978-5-907144-12-5 (рус.)
3. Богуславский А.А., Соколов С.М. Основы программирования на языке Си++: Для студентов физико-математических факультетов педагогических институтов. – Коломна: КГПИ, 2007.
4. Вайсфельд Мэтт Объектно-ориентированный подход. 5-е межд. изд. — СПб.: Питер, 2020. — 256 с.: ил. — (Серия «Библиотека программиста»). ISBN 978-5-4461-1431-3
5. Васильев А. Н. Объектно-ориентированное программирование на C++ — СПб.: Наука и Техника, 2016. — 544 с., ил. ISBN 978-5-94387-984-5
6. Герберт Шилдт C/C++ Справочник программиста. — М.: «Вильямс», 2003. — С. 432. — ISBN 5-8459-0459-5
7. Готтшлинг, Питер. C++ для инженерных и научных расчетов.: Пер. с англ. — СПб.: ООО “Диалектика”. 2020. — 512 с.: ил. — Парал. тит. англ. ISBN 978-5-907203-30-3 (рус.)
8. Демидович Е.М. Основы алгоритмизации и программирования. Язык СИ : Учебное пособие. – СПб.: БХВ-Петербург, 2006.
9. Доусон М. Изучаем C++ через программирование игр. - СПб.: Питер, 2016. - 352 с.: ил. ISBN 978-5-496-01629-2
10. Дэвис С. C++ для «чайников». – К. : Диалектика, 2005.
11. Златопольский Д. М. Программирование: типовые задачи, алгоритмы, методы / Д.М. Златопольский. 4е изд., электрон. - М.: Лаборатория знаний, 2020. - 226 с. ISBN 978-5-00101-789-9
12. Кёниг Эндрю, Му Барбара Эффективное программирование на C++. Серия книг "C++ In-Depth"++. — М.: «Вильямс», 2002. — С. 384. ил. — ISBN 5-8459-0350-5
13. Кнут Дональд 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
14. Конова Е. А., Поллак Г. А. Алгоритмы и программы. Язык C++: Учебное пособие. — 2е изд., стер. — СПб.: Издательство «Лань», 2017.— 384 с.: ил. — (Учебники для вузов. Специальная литература). ISBN 978-5-8114-2020-9
15. Лоспинозо Джош C++ для профи. — СПб.: Питер, 2021. — 816 с.: ил. — (Серия «Для профессионалов»). ISBN 978-5-4461-1730-7
16. Лафоре Р. Объектно-ориентированное программирование в C++. - 4-е издание. – С.Петербург – Изд. Питер. – 2004. – 924 с.
17. Мейерс Скотт. Эффективный и современный C++: 42 рекомендации по испольованию C++ 11 и C++14.: Пер. с англ. - М. : ООО "ИЛ. Вильяме", 2016. - 304 с.: ил. - тит. англ. ISBN 978-5-8459-2000-3 (рус.)

18. Марапулец Ю.В. Язык C++. Основы программирования. Издание второе, исправленное и дополненное / Ю. В. Марапулец. — Петропавловск-Камчатский: КамГУ им. Витуса Беринга, 2019. — 158 с. ISBN 978-5-7968-0675-3
19. Павлов Л. А. Структуры и алгоритмы обработки данных : учебник / Л. А. Павлов, Н. В. Первова. — 2-е изд., испр. и доп. — СПб: Лань, 2020. — 256 с. : ил. — (Учебники для вузов. Специальная литература). ISBN 978-5-8114-4881-4
20. Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб: Питер, 2007.
21. Павловская Т. А., Щупак Ю. А. C++. Объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2006. — 265 с: ил. ISBN 5-94723-842-X
22. Павловская Т. А., Щупак Ю. А. C/C++. Структурное и объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2011. — 352 с.: ил. — (Серия «Учебное пособие»). ISBN 978-5-459-00613-1
23. Пай П., Абрахам П. Реактивное программирование на C++ / пер. с анг. В. Ю. Винника. – М.: ДМК Пресс, 2019. – 324 с.: ил. ISBN 978-5-97060-778-7
24. Прата Стивен. Язык программирования C++. Лекции и упражнения, 6-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2012. - 1248 с. : ил. - Парал. тит. англ. ISBN 978-5-8459-1 778-2 (рус.)
25. Пош М. Программирование встроенных систем на C++17 / пер. с анг. А. В. Снастина. – М.: ДМК Пресс, 2020. – 394 с.: ил. ISBN 978-5-97060-785-5
26. 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
27. Эккель Б., Эллисон Ч. Философия C++. Практическое программирование. — СПб.: Питер, 2004. — 608 с.: ил. ISBN 5-469-00043-5