

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Урок № 17

Битовые операции,
работа с файлами,
поиск файлов

Содержание

1. Краткие сведения из курса двоичной арифметики...	3
Сложение	7
Вычитание	8
Шестнадцатеричная система исчисления.....	10
2. Битовые операции	16
3. Объединения.....	29
4. Битовые поля	35
Синтаксис объявления битового поля.....	36
5. Работа с файлами.....	43
Понятие файла.....	43
6. Поиск файлов.....	71

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе [Adobe Acrobat Reader](#).

1. Краткие сведения из курса двоичной арифметики

В повседневной жизни мы работаем с десятичной системой исчисления. То есть, любое число мы можем представить с помощью цифр **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**.

В зависимости от того, где в числе стоит цифра в числе, она может означать разное: если эта цифра последняя, то она расположена в разряде единиц, предпоследняя — разряд десятков, еще левее — разряд сотен и так далее.

Такая систем исчисления, в которой «смысл» цифры зависит от ее позиции в числе, называется позиционной системой исчисления.

По сути, любое число можно расписать в виде суммы цифр, каждая из которых умножена на десять в определенной степени (позиции числа). В случае единиц, эта степень — нулевая.

Рассмотрим вначале на примере двузначного числа.

$$42 = 4 \cdot 10^1 + 2 \cdot 10^0.$$

Так как нумерация позиций начинается с нуля (справа налево), то цифра **2** умножается на десять в нулевой степени (позиция цифры **2** в числе **42** равна нулю), а цифра **4** — на десять в первой степени (позиция цифры **4** — это единица).

Рассмотрим следующий пример с четырехзначным числом **1231**.

Начинаем анализ числа справа налево: позиция самой правой единицы — ноль, позиция тройки — один, позиция двойки — два, позиция самой левой в числе единицы — три. Итого получаем следующее:

$$1231 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 3 \cdot 10^0.$$

Число, на степень которого умножаются цифры называется базой (основой) системы счисления. Для десятичной системы базой, логично, является десятка.

Мы с Вами давно уже знаем о том, что компьютер может различить только нулевое и единичное состояние бита, и работает компьютер в системе исчисления с основанием 2 или в двоичной системе.

Примечание: *Бит* получил свое название от английского *Binary digit* (двоичная цифра).

Пора нам познакомиться с правилами выполнения действий с двоичными числами. Чем мы, собственно, и займемся в этом уроке.

Все числа (да и вообще информация) в памяти компьютера состоят только из двух цифр — 0 и 1. Их расположение, как и в случае десятичной системы счисления, указывает на разряд. Только теперь число можно разложить на сумму цифр, умноженные не на степени десятки, а степени двойки.

Таким образом, для представления числа двоичной системы в десятичной системе исчисления нужно каждую цифру числа, начиная с самой правой, умножить на базу системы исчисления (два) в степени, соответствующей её разряду. Затем складываем все получившиеся таким образом числа.

Давайте определим десятичный аналог двоичного числа 101:

Самое правое число = $1 \cdot 2^0$.

Следующее число = $0 \cdot 2^1$.

Третье справа число = $1 \cdot 2^2$.

$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 1 + 0 + 4 = 5.$$

Теперь определим десятичный аналог двоичного числа 101101:

Самое правое число = $1 \cdot 2^0$.

Следующее число = $0 \cdot 2^1$.

Третье справа число = $1 \cdot 2^2$.

Четвертое = $1 \cdot 2^3$.

...

$$101101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 = \\ 1 + 0 + 4 + 8 + 0 + 32 = 45.$$

Рассмотрим процесс обратного перевода, т.е. каким образом можно из десятичной системы перевести число в двоичную.

Одним из алгоритмов перевода десятичного числа в двоичное является деление нацело на два с последующим «сбором» двоичного числа из остатков. Переведем таким образом число 14 в двоичное представление.

$$14 / 2 = 7, \text{ остаток } 0$$

$$7 / 2 = 3, \text{ остаток } 1$$

$$3 / 2 = 1, \text{ остаток } 1$$

$$1 / 2 = 0, \text{ остаток } 1$$

Собирать остатки надо с конца, то есть с последнего деления. Получаем 1110.

Выполним то же самое для числа 77:

$$77 / 2 = 38, \text{ остаток } 1;$$

$$38 / 2 = 19, \text{ остаток } 0;$$

$$19 / 2 = 9, \text{ остаток } 1;$$

$$9 / 2 = 4, \text{ остаток } 1;$$

$$4 / 2 = 2, \text{ остаток } 0;$$

$$2 / 2 = 1, \text{ остаток } 0;$$

$$1 / 2 = 0, \text{ остаток } 1.$$

Собираем остатки вместе, начиная с конца: 1001101.

Проверим, выполнив обратный перевод:

$$1001101 =$$

$$1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 =$$

$$64 + 0 + 0 + 8 + 4 + 0 + 1 = 77.$$

Рассмотрим второй способ перевода.

У нас есть десятичное число 57. Чтобы перевести его в двоичную систему, нужно определить, какая максимальная степень двойки не превосходит это число.

$$2^6 = 64.$$

Это явно многовато. А вот $2^5 = 32$.

Мы определили старший разряд.

$$32 = 100000.$$

Теперь ищем следующий разряд.

$$57 - 32 = 25.$$

Теперь для 25 ищем степень двойки, которая не превосходит 25.

$$2^4 = 16.$$

Значит, следующий разряд у нас тоже равен 1.

$$32 + 16 = 48 = 110000.$$

$$57 - 48 = 9.$$

$2^3 = 8$, это меньше, чем 9. Значит следующий разряд тоже будет единичкой.

$$32 + 16 + 8 = 56 = 111000.$$

$57 - 56 = 1$, то есть осталась только одна степень 2^0 . Таким образом, $57 = 111001$.

Сложение

Компьютер выполняет арифметические действия только в двоичном формате. Поэтому, необходимо знать правила сложения в двоичной системе исчисления. Напомним их:

$$0 + 0 = 0;$$

$$1 + 0 = 1;$$

$$1 + 1 = 0 \text{ (1 переносится в старший разряд)}.$$

Давайте рассмотрим использование этих правил на конкретном примере.

Пример

Сложить числа 65 и 42, представленные в двоичной системе исчисления. В десятичной системе исчисления все осуществляется достаточно просто: $65+42=107$. Для сложения этих чисел в двоичной системе исчисления нужно сначала перевести их в эту систему.

Пусть для хранения числа нам доступны 8 бит. Для этого можно воспользоваться любым из описанных выше способов.

Итак, $65 = 01000001$.

Обратите внимание на то, что ведущий ноль в двоичном представлении числа добавлен для дополнения двоичного представления до восьми бит.

Аналогично: $42 = 00101010$.

Выполним сложение этих чисел:

$$\begin{array}{r}
 01000001 \\
 + \\
 00101010 \\
 \hline
 01101011
 \end{array}$$

Можете перепроверить и убедиться, что $01101011 = 107$:

$$\begin{aligned}
 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = \\
 64 + 32 + 8 + 2 + 1 = 107
 \end{aligned}$$

Вычитание

Только что, мы рассмотрели сложение чисел в компьютере. А как же осуществляется вычитание? Для выполнения операции вычитания последнее заменяется сложением, а в качестве второго слагаемого берется противоположное число.

Например, пусть надо выполнить вычитание: $65 - 42$.
Заменим его сложением: $65 + (-42)$.

Но как получить соответствующее отрицательное двоичное число, спросите вы?

Этот вопрос мы сейчас и рассмотрим.

Все представленные выше двоичные числа имеют положительные значения, что обозначается нулевым значением самого левого (старшего) разряда.

Отрицательные двоичные числа содержат единичный бит в старшем разряде.

Для получения отрицательного двоичного числа можно использовать следующий алгоритм:

1. Взять соответствующее положительное число и инвертировать его биты (1 заменить на 0 и наоборот).
2. К полученному числу прибавить 1.

Пример использования рассмотренного алгоритма.

Пример 1

Получить двоичное представление числа -65 .

Переводим вначале число 65 в двоичную систему. Из примера выше, $65 = 01000001$.

Инвертируем каждый разряд: 10111110.

Добавляем к младшему разряду единицу:

$$\begin{array}{r}
 10111110 \\
 + \\
 1 \\
 \hline
 10111111
 \end{array}$$

Убедимся в правильности представления. Сумма $+65$ и -65 должна составить нуль:

$$\begin{array}{r}
 01000001 \\
 + \\
 10111111 \\
 \hline
 (1) 00000000
 \end{array}$$

Все восемь бит имеют нулевое значение. Пока будем считать, что полученная единица, перенесенная влево, потеряна. Это правило позволяет выполнять вычитание чисел в двоичной системе исчисления: **Вычитание заменяется сложением и в качестве второго слагаемого берется отрицательное число.**

Пример 2

Вычесть из 65 число 42.

Двоичное представление для 42 — это 00101010, для -42 двоичное представление будет следующим: 11010110:

$$\begin{array}{r}
 65 = 01000001 \\
 + \\
 (-42) = 11010110 \\
 \hline
 23 (1) \quad 00010111
 \end{array}$$

Пример 3

Какое значение необходимо прибавить к двоичному числу 00000001, чтобы получить число 00000000?

В терминах десятичного исчисления ответом будет число -1. Для двоичного исчисления это число 11111111:

$$\begin{array}{r}
 00000001 \\
 + \\
 11111111 \\
 \hline
 00000000
 \end{array}$$

Шестнадцатеричная система исчисления

Шестнадцатеричная система счисления, как следует из названия, имеет в своём основании число 16. Почему так? Дело в том, что единица информации в информатике — это бит. Восемь бит образуют байт. Также информационной среде существует такое понятие, как машинное слово — это минимальная единица данных, представляющая собой шестнадцать бит, то есть два байта. Считается, что

машинное слово — это минимальная величина разрядности регистров процессора, при которой можно работать с компьютер.

Так вот, как мы знаем, компьютер работает на двоичном коде, т.е. использует двоичную систему исчисления. Однако, можно заметить, что в ней бывает довольно много разрядов, особенно при переводе больших чисел.

Шестнадцатеричная система счисления использует 16 цифр:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Шестнадцатеричное число является компактным и лёгким для чтения. Его легко преобразовать в двоичное и наоборот.

Каждый разряд шестнадцатеричного числа — это тетрада (четыре разряда). Каждую тетраду легко преобразовать в двоичное число и наоборот.

При переводе двоичного числа в шестнадцатеричное, первое разбивается на группы по четыре разряда, начиная с конца. В случае, если количество разрядов не кратно четырем, первая четверка дописывается нулями впереди. Каждой четверке соответствует одноразрядное число шестнадцатеричной системы счисления.

Таблица 1

Десятичное	Двоичное	Шестнадцатеричное
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4

Десятичное	Двоичное	Шестнадцатеричное
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Пример

$$10001100101 = 0100\ 1100\ 0101 = 4\ C\ 5 = 4C5.$$

В случае обратного перевода шестнадцатеричные цифры заменяются соответствующими четырехразрядными двоичными числами.

Перевод из шестнадцатеричной системы счисления в десятичную выполняется аналогично переводу из двоичной и восьмеричной. Только здесь в качестве основания степени выступает число 16, а цифры от A до F заменяются десятичными числами от 10 до 15.

$$4C5 = 4 * 16^2 + 12 * 16^1 + 5 * 16^0 = \\ 4 * 256 + 192 + 5 = 1221.$$

Максимальное двухразрядное число, которое можно получить с помощью шестнадцатеричной записи, — это число FF.

$$FF = 15 * 16^1 + 15 * 16^0 = 240 + 15 = 255.$$

В двоичном представлении FF будет выглядеть как восьмиразрядное число 11111111. Наименьшей рабочей ячейкой компьютерной памяти является байт, который состоит из 8-ми битов. Каждый бит может быть в двух состояниях — «включено» и «выключено». Одному из них сопоставляют ноль, другому — единицу.

Следовательно, в одном байте можно сохранить любое двоичное число в диапазоне от 00000000 до 11111111. В десятичном представлении это числа от 0 до 255. В шестнадцатеричном — от 0 до FF. С помощью шестнадцатеричной системы счисления удобно кратко, с помощью двух цифр-знаков, записывать значения байтов. Например, 0E или F5.

Несмотря на то, что 255 — это максимальное значение, которое можно сохранить в байте, состоящий из 8-ми битного байта 256, так как одно из них отводится под хранение нуля. Количество возможных состояний ячейки памяти вычисляется по формуле 2^n , где n — количество составляющих ее бит. В случае восьми бит получаем: $2^8 = 256$.

Рассмотрим несколько простых примеров шестнадцатеричной арифметики. Следует помнить, что после шестнадцатеричного числа F следует шестнадцатеричное 10, что равно десятичному числу 16:

$$6+4=A.$$

В десятичной системе $6+4=10$, число 10 в шестнадцатеричной системе — это A.

$$\begin{aligned} 5+8 &= D, \\ F+1 &= 10. \end{aligned}$$

Число F в десятичной системе это 15, $15+1=16$, 16 в шестнадцатеричной системе — это 10.

Однако при сложении больших чисел нам уже понадобится таблица правил сложения в шестнадцатеричной системе.

Таблица 2

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Используя таблицу:

$$F + F = 1E.$$

Число **F** в десятичной системе — это **15**, $15 + 15 = 30$, переводим **30** в 16-ричную систему: делим **30** на **16**, получаем

целую часть от деления — это 1, а остаток — 14 (это E в 16-ричной системе), итого результат — 1E.

$$FF+1=100.$$

Прибавляем к последней цифре F единицу, получаем (согласно таблице) 10 (в 16-ричной системе), 0 становится последней цифрой результата, а единица переносится в старший разряд. Поэтому на следующем шаге к первой цифре F прибавляем эту (перенесенную) единицу, опять получаем 10, 0 становится следующей цифрой, а 1 переносится дальше. Результат равен 100.

В этом случае удобнее будет представить особенности этой операции сложения в столбик:

$$\begin{array}{r} {}^1F^1F \\ + \\ \phantom{{}^1}1 \\ \hline 100. \end{array}$$

2. Битовые операции

Только что мы с Вами разобрались с общей теорией, теперь пообщаемся с битовой арифметикой с точки зрения языка C++.

В языке C++ существует ряд операторов, которые позволяют выполнять операции над отдельными разрядами (битами). Они носят название битовые (или побитовые) операции.

Зачем нужны битовые операторы?

Есть ряд ситуаций, когда максимально разумно использовать каждый доступный бит, т.е. необходима максимальная оптимизация используемой памяти. Например, программные модули научных задач, которые используют огромное количество данных, игры, где манипуляции с битами могут быть использованы для дополнительной скорости; встроенные устройства, использующие технологию Интернета вещей, или, например, кофемашины, где память по-прежнему ограничена.

В языке C++ есть 6 побитовых операторов:

Оператор Символ Пример Операция

Таблица 2

Название оператора	Представление в C++	Пример использования	Описание
Побитовый сдвиг влево	<<	$x \ll y$	Все биты в x смещаются влево на y бит
Побитовый сдвиг вправо	>>	$x \gg y$	Все биты в x смещаются вправо на y бит

Название оператора	Представление в C++	Пример использования	Описание
Побитовое НЕ	<code>~</code>	<code>~x</code>	Все биты в <code>x</code> меняются на противоположные
Побитовое И	<code>&</code>	<code>x & y</code>	Над каждым битом в <code>x</code> и соответствующим ему битом в <code>y</code> выполняется операция логического И
Побитовое ИЛИ	<code> </code>	<code>x y</code>	Над каждым битом в <code>x</code> и соответствующим ему битом в <code>y</code> выполняется операция логического ИЛИ
Побитовое исключающее ИЛИ (XOR)	<code>^</code>	<code>x ^ y</code>	Над каждым битом в <code>x</code> и соответствующим ему битом в <code>y</code> выполняется операция логического исключающего ИЛИ (XOR)

Примечание: в битовых операциях следует использовать только целочисленные типы данных *unsigned*, так как C++ не всегда гарантирует корректную работу побитовых операторов с целочисленными типами *signed*.

Рассмотрим более детально использование перечисленных операторов на примерах.

Побитовый сдвиг влево (<<)

Сдвигает разряды левого операнда влево на число позиций, указанное правым операндом. Освобождающиеся позиции заполняются нулями, а разряды, сдвигаемые за левый предел левого операнда, теряются.

Например, выражение $2 \ll 1$, обозначает «сдвинуть биты влево в литерале 2 на одно место».

Рассмотрим число 2, которое в двоичной системе равно 0010 (пусть нам доступно только 4 бита):

$$2 = 0010$$

$$2 \ll 1 = 0100 = 4$$

$$2 \ll 2 = 1000 = 8.$$

Если мы после последнего примера выполним еще один сдвиг, то один бит перемещается за пределы самого литерала! Биты, сдвинутые за пределы двоичного числа, теряются навсегда.

То есть если для нашего примера (когда доступно только 4 бита) необходимо сдвинуть биты влево в литерале 2 на три позиции, то: $2 \ll 3 = 0000 = 0$ (бит со значением 1 был сдвинут за предел четвертого из четырех доступных битов и был потерян)

Побитовый сдвиг влево вправо (>>)

Сдвигает биты аналогичным образом вправо.

Например:

$$12 = 1100$$

$$12 \gg 1 = 0110 = 6$$

$$12 \gg 2 = 0011 = 3$$

$$12 \gg 3 = 0001 = 1.$$

В последнем примере мы снова переместили бит за пределы литерала. Он также потерялся навсегда.

Хотя в примерах, приведенных выше, мы смещаем биты только в литералах, мы также можем смещать биты и в переменных:

```
int main()
{
    unsigned int x = 4;
    x = x << 1; // x должен стать равным 8
    cout << x;
    return 0;
}
```

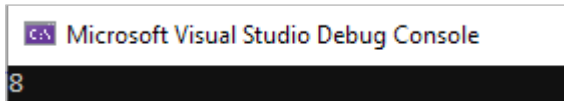


Рисунок 1

Инверсия или побитовый оператор НЕ (~)

Просто меняет каждый бит на противоположный, например, с 0 на 1 или с 1 на 0.

Примечание: результаты побитового НЕ зависят от размера типа данных.

Предположим, что размер типа данных составляет 4 бита:

$$4 = 0100$$

$$\sim 4 = 1011 \text{ (двоичное)} = 11 \text{ (десятичное)}.$$

Предположим, что размер типа данных составляет 8 бит:

$$4 = 0000\ 0100$$

$$\sim 4 = 1111\ 1011 \text{ (двоичное)} = 251 \text{ (десятичное)}.$$

Побитовые операторы И, ИЛИ

Побитовые операторы И (&) и ИЛИ (|) работают аналогично логическим операторам И и ИЛИ.

Однако, побитовые операторы применяются к каждому биту отдельно! Например, рассмотрим выражение: $5 \mid 6$.

В двоичной системе это: $0101 \mid 0110$.

Лучше операнды размещать один под одним, чтобы было более понятно, как именно выполняется побитовая операция, т.е. следующим образом:

```
0 1 0 1 // 5
0 1 1 0 // 6
```

А затем применять операцию к каждому столбцу с битами по отдельности.

Как вы помните, логическое ИЛИ возвращает **true** (1), если один из двух или оба операнды истинны (1). Аналогичным образом работает и побитовое ИЛИ.

Выражение $5 \mid 6$ обрабатывается следующим образом:

```
0 1 0 1 // 5
0 1 1 0 // 6
-----
0 1 1 1 // 7
```

Результат: 0111 (двоичное) = 7 (десятичное).

Также можно обрабатывать и комплексные выражения ИЛИ, например, $1 \mid 4 \mid 6$. Если хоть один бит в столбце равен **1**, то результат целого столбца — **1**.

Например:

```
0 0 0 1 // 1
0 1 0 0 // 4
0 1 1 0 // 6
-----
0 1 1 1 // 7
```

Результатом $1 \mid 4 \mid 6$ является десятичное 7 .

Побитовое И работает аналогично логическому И — возвращается **true**, только если оба бита в столбце равны 1.

Рассмотрим выражение: $5 \& 6$:

```

0 1 0 1 // 5
0 1 1 0 // 6
-----
0 1 0 0 // 4

```

Также можно решать и комплексные выражения И, например, $1 \& 3 \& 7$. Только при условии, что все биты в столбце равны 1, результатом столбца будет 1.

```

0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
-----
0 0 0 1 // 1

```

Побитовое исключающее ИЛИ (^)
(сокр. «XOR» от англ. «eXclusive OR»).

При обработке двух операндов, исключающее ИЛИ возвращает **true** (1), только если один и только один из операндов является истинным (1). Если таких нет или все операнды равны 1, то результатом будет **false** (0).

Рассмотрим выражение: $6 \wedge 3$:

```

0 1 1 0 // 6
0 0 1 1 // 3
-----
0 1 0 1 // 5

```

Также можно решать и комплексные выражения XOR, например, $1 \wedge 3 \wedge 7$. Если единиц в столбце чётное

количество, то результатом будет 0, если же нечётное количество, то результат — 1. Например:

```

0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
-----
0 1 0 1 // 5

```

Напишем программу, реализующую рассмотренные выше примеры.

```

#include <iostream>
using namespace std;

int main()
{
    int x, y, z, k;

    x = 5;
    y = 6;

    k = x | y ;    // Операция    5 | 6
    cout << k << " ";

    x = 1;
    y = 4;
    z = 6;

    k = x | y | z;    // Операция    1 | 4 | 6
    cout << k << " ";

    x = 5;
    y = 6;

    k = x & y;    // Операция    5 & 6
    cout << k << " ";
}

```

```

x = 1;
y = 3;
z = 7;

k = x & y & z;    // Операция    1 & 3 & 7
cout << k << " ";

x = 6;
y = 3;

k = x ^ y;        // Операция    6 ^ 3
cout << k << " ";

return 0;
}

```

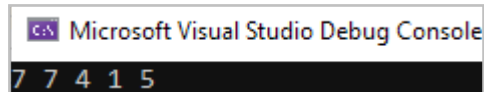


Рисунок 2

Рассмотрим еще один пример на применение различных битовых операций

```

#include <iostream>
using namespace std;

int main()
{
    int y = 02, x = 03, z = 01, k;

    k = x | y & z;    // Операция    1
    cout << k << " ";

    k = x | y & ~z;    // Операция    2
    cout << k << " ";
}

```

```

k = x ^ y & ~z;    // Операция      3
cout << k << " ";

k = x & y && z; // Операция      4
cout << k << " ";

x = 1;
y = -1;

k = !x | x;    // Операции      5
cout << k << " ";

k = -x | x;    // Операции      6
cout << k << " ";

k = x ^ x;    // Операции      7
cout << k << " ";

x <<= 3;    // Операции      8
cout << x << " ";

y <<= 3;    // Операции      9
cout << y << " ";

y >>= 3;    // Операции 10
cout << y << " ";

return 0;
}

```

Результат работы программы:

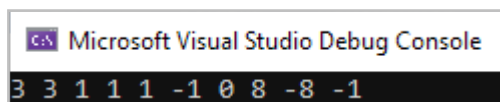


Рисунок 3

Примечание: здесь мы с Вами знакомимся с еще одной системой исчисления. Целые константы, начинающиеся с цифры 0, являются восьмеричными числами. Восьмеричное представление целых чисел особенно удобно, когда приходится работать с поразрядными операциями, так как восьмеричные числа легко переводятся в двоичные. В этой задаче числа 01, 02, 03 соответствуют числам 1, 2 и 3, так что появление восьмеричных чисел служит намеком, что программа рассматривает значения x , y и z как последовательности двоичных цифр.

Комментарии к коду

Операция 1

```
x = 03; y = 02; z = 01;
k = x|y&z.
```

Вследствие приоритетов операций: $k = (x|(y&z));$.
Самое внутреннее выражение вычисляется первым.

```
k = (x|(02&01));
02 & 01 = 00000010 & 00000001 =
00000000 = 0 k = (x|00);
03 | 00 = 00000011 | 00000000 =
00000011 = 03
```

03

Операция 3

```
x = 03; y = 02; z = 01;
k = x^y&~z;
k = (03^02);
1
```

Операция 4

```

x = 03; y = 02; z = 01;
k = x&y&&z;

k = ((x&y)&&z);
k = ((03&02)&&z);
k = (02&&z);
k = (true&&z);
k = (&&01);
k = (true&&true)
true или 1.

```

Операция 5

```

x = 01;
k = !x|x;
k = ((!x)|x);
k = ((!true)|x);
k = (false|01);
k = (0|01);
1.

```

Операция 6

```

x = 01;
k = -x|x;

k = ((-x)|x);
k = (-01|01);
-01 | 01 = 11111111 | 00000001 =
11111111 = -1 -1.

```

Операция 7

```

x = 01;
k = x^x;

```

$k = (01 \wedge 01);$
0.

Операция 8

$x = 01;$
 $x \ll= 3;$
 $x = 01 \ll 3;$
 $01 \ll 3 = 000000001 \ll 3 =$
 $00001000 = 8 \quad x = 8;$
8.

Операция 9.

$y = -01;$
 $y \ll= 3;$
 $y = -01 \ll 3$
 $-01 \ll 3 = 11111111 \ll 3 =$
 $11111000 = -8$
 $y = -8;$
-8

Операция 10.

$y = -08;$
 $y \gg= 3;$
 $y = -08 \gg 3;$
-1

Примечание: В некоторых случаях вместо *-1* может получиться другой результат (*8191*). Появление этого значения объясняется тем, что на некоторых компьютерах при операции сдвига знак числа может не сохраниться. Не все трансляторы языка

С++ гарантируют, что операция сдвига арифметически корректна, поэтому в любом случае более ясным способом деления на 8 было бы явное деление $y=y/8$.

3. Объединения

Основное назначение технологии объединений **union** — это экономия памяти, т.е. использование объединения переменных позволяет создавать оптимальную по использованию памяти программу.

Чтобы понять в чем смысл объединения нужно вспомнить как хранятся переменные.

Разные переменные разного типа или одинаковой группы типов (вроде **int**, **long** и **short**) несмотря на работу с одним и тем же типом данных (имею ввиду целое) занимают в памяти разное количество байт.

long будет занимать большее количество байт в памяти, чем **int** или **short** (не во всех компиляторах), но при этом в память для переменной этого типа вполне можно записать значения **int** или **short** при необходимости.

Получится, что не все зарезервированные байты **long-a** будут востребованы. Если поместить, к примеру число **325** в **long**, будут заняты два байта (зависит от разрядности процессора), а остальные байты заполнятся нулями.

Именно в этом и появляется смысл **union**, ибо эта инструкция говорит компилятору: «Зарезервируй мне место для типа данных с максимальным запросом объема памяти, а мы уже будем сами разбираться, как и какие значения в них положить».

Объединение — это такой формат данных, который подобен структуре и способен хранить в пределах одной зарезервированной области памяти различные типы данных.

Но в каждый определенный момент времени в объединении хранится только один из этих типов данных и возможно использовать лишь значение этого элемента (компонента).

Синтаксически определяют объединение так (очень похоже на определение структуры):

```
union
{
    <имя типа1> <компонента1>;
    <имя типа2> <компонента2>;
    .....
    <имя типаN> <компонентаN>;
};
```

Пример

```
union
{
    short int component1;
    int component2;
    long int component3;
} myUnion; // объект объединения
```

Доступ к элементам объединения осуществляется так же, как и к элементам структур: Имя объекта объединения **myUnion**, точка **.** и имя элемента **name1**.

К данным, которые хранят элементы структуры (например **short**, **int**, **long**) мы можем обращаться в любой момент (хоть к одному, хоть и ко всем сразу). А в объединении могут храниться данные либо **short**, либо **int**, либо **long**.

Например:

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

int main()
{
    union
    {
        short int component1;
        int component2;
        long int component3;
    } myUnion; // объект объединения

    myUnion.component1= 22;
    cout << myUnion.component1<< endl;

    myUnion.Component3= 22222222;
    cout << myUnion.Component3<< endl;

    cout << myUnion.component1<< endl; // снова
                                         // обращаемся к component1

    return 0;
}
```

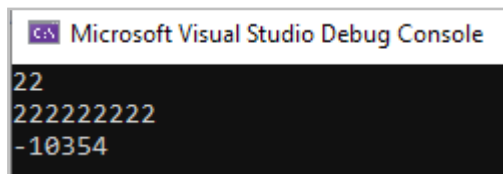


Рисунок 4

Как видите после того, как мы записали значение в элемент `component3` типа `long int`, получить значение элемента `component1` уже нельзя. Все потому, что в их общую

память уже записано значение `long int`, а переменная типа `short int` неспособна работать с данными такого объема. Схематически это можно представить так:



Рисунок 5

Поэтому, чтобы опять работать с данными типа `short int` необходимо снова присвоить элементу `component1` новое значение. Вот и получается — память одна и та же, а переменные в ней размещаются разные. К какой обращаемся — такая и запрашивает из этой памяти значение.

Элементы объединения располагаются в памяти начиная с одного места (как бы накладываются друг на друга), в отличие от элементов структуры (они располагаются в памяти последовательно один за другим).

Применение объединений вызвано необходимостью экономии памяти, когда нужно хранить и использовать данные разных типов, но обращаться к ним можно не одновременно.

Если бы мы описали просто набор переменных, не объединяя их в `union`, то для их размещения потребовалось бы $2 + 4 + 4 \text{ байта} = 10 \text{ байт}$. Вот и экономия. А так объединение занимает 4 байта.

Рассмотрим еще один пример работы с объединениями.

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

union Employee {
    int id;
    double salary;
};

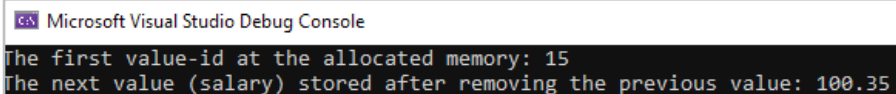
int main()
{
    Employee myEmployee;

    myEmployee.id = 15;
    cout << "The first value (id) at ";
    cout << "the allocated memory: ";
    cout << myEmployee.id << "\n";

    myEmployee.salary = 100.35;
    cout << "The next value (salary) stored ";
    cout << "after removing the previous value: ";
    cout << myEmployee.salary << "\n";

    return 0;
}
```

Результаты работы программы:



```
Microsoft Visual Studio Debug Console
The first value-id at the allocated memory: 15
The next value (salary) stored after removing the previous value: 100.35
```

Рисунок 6

Разберем наш код. Мы создали объединение `Employee` (и далее переменную `myEmployee` такого типа), состоящее из двух элементов: первый (`id`) элемент целочисленного типа, второй (`salary`) — вещественного.

Вначале в первый элемент мы занесли значение `15` и вывели значение этого компонента объединения в консоль. Далее мы занесли значение `100.35` во второй компонент объединения (значение первого компонента уже недоступно после этого действия) и также вывели его в консоль.

4. Битовые поля

В прошлом уроке мы с Вами рассматривали понятие структуры.

Предположим, что мы создали следующую структуру для хранения **TRUE/FALSE** значений некоторого набора переменных (например, некоторых флагов — признаков, знает ли пользователь определенный язык программирования):

```
struct statusField
{
    unsigned int JS;
    unsigned int PHP;
    unsigned int Python;
} state;
```

В этом случае размер занимаемой памяти будет:

$$3 * \text{sizeof}(\text{unsigned int}) = 3 * 4 = 12 \text{ байтов.}$$

Даже, если вместо типа **unsigned int** мы будем использовать для флагов (знает или не знает пользователь JS, PHP, Python) тип **bool** или тип **char**, то размер занимаемой памяти в этом случае составит 3 байта.

Но ведь в каждом элементе данной структуры мы собираемся хранить только 0 или 1, т.е. фактически более трех бит нам никогда не понадобится.

Таким образом мы приходим к необходимости задавать для каждого поля структуры такое количество бит, которое нам нужно для хранения нашей информации.

В языке C++ есть возможность устанавливать для полей (элементов) структур такое количество памяти в битах, как нужно для определенной задачи.

Такое поле структуры, которое определяет последовательность бит (количество памяти в битах) называется битовым полем. Чаще всего битовые поля используются в задачах, связанных с булевой логикой (например, для кодирования цветов и шифрования данных).

Хотя правила языка не имеют ограничений на характер этих полей, кроме требования, чтобы они помещались в объеме машинного слова, в типичных применениях поля битов служат для хранения целых данных (чаще типа `unsigned`).

Синтаксис объявления битового поля

```
struct <имя> {  
    <тип> <имя>: <размер>;  
    ...  
}
```

Описание поля битов состоит из описания типа поля, его имени и указанного после двоеточия размера поля в битах.

Таким образом, для того, чтобы наша первоначальная структура занимала именно 3 бита, необходимо использовать в ней битовые поля:

```
struct statusField  
{  
    unsigned int JS: 1;  
    unsigned int PHP: 1;
```

```
    unsigned int Python: 1;
} state;
```

Рассмотрим еще один пример, когда нам нужно использовать один байт:

```
struct fieldbite
{
    unsigned short field1: 2;
    unsigned short field2: 2;
    unsigned short field3: 4;
} field;
```

Здесь мы создали структуру, в которой поля будут занимать указанное количество бит. $2 + 2 + 4$ дает 8 бит (т.е. мы выравнивали размер нашей структуры до одного байта).

В отличие от объединений (**union**) размер битовых полей изменяется, в зависимости от того, сколько бит было указано программистом при создании данного поля. Если мы установили, например, размер структуры в 7 бит (создали два поля по 3 бита, и одно — 1 бит), то фактически будет выделен один байт (8 бит) под эти три переменные.

Если мы закажем для нашей структуры с битовыми полями 11 бит, то фактически будет выделено два байта (16 бит). При этом во втором байте будут использованы только 5 бит, а остальные, скорее всего, будут простаивать. Поэтому в момент создания битовых полей следует учитывать такое автоматическое «выравнивание» до байта. Нужно стараться распределять в нем переменные таким образом, чтобы каждый бит был востребован. Для

выравнивания занимаемой памяти можно использовать неименованные битовые поля.

Если имя поля опущено, то создается скрытое поле. Если размер поля битов представлен числом 0, то следующее поле битов начнется с границы машинного слова.

Приведем еще один короткий пример, в котором битовые поля отводятся под дату.

```
struct Date {
    unsigned short nWeekDay : 3; // 0..7 (3 bits)
    unsigned short nMonthDay : 6; // 0..31 (6 bits)
    unsigned short nMonth : 5; // 0..12 (5 bits)
    unsigned short nYear : 8; // 0..100 (8 bits)
};
```

На рисунке 7 представлена структура памяти для объекта типа `Date`.

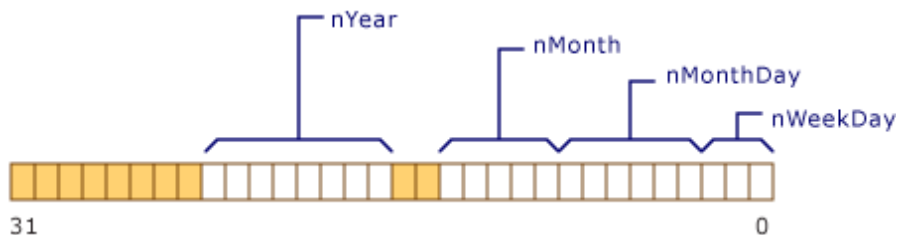


Рисунок 7

Так как поле `nYear` имеет длину 8 бит, то это приведет к переполнению границы слова объявленного типа `unsigned short`. Поэтому данное битовое поле начинается в начале нового `unsigned short`.

Рассмотрим еще один пример использования битовых полей для хранения компонентов даты и времени.

```

#include <iostream>
using namespace std;

int main()
{
    struct DateTime {
        unsigned short DayNum: 5;
        unsigned short MonthNum: 4;
        unsigned short YearNum: 7;
        unsigned short HourNum: 5;
        unsigned short MinuteNum: 6;
        unsigned short SecondNum: 6;
    };

    DateTime d;
    int i

    cout << "Input the day number (from 1 to 31):"
         << '\t';
    cin >> i;
    d.DayNum = i;
    cout << "Input the month number (from 1 to 12):"
         << '\t';
    cin >> i;
    d.MonthNum = i;
    cout << "Input Year (from 00 to 99) : " << '\t';
    cin >> i;
    d.YearNum = i;

    cout << endl << "Input Hours (from 0 to 24):"
         << '\t';
    cin >> i;
    d.HourNum = i;
    cout << "Input Minutes (from 0 tp60):" << '\t';
    cin >> i;
    d.MinuteNum = i;
    cout << "Input Seconds (0-60):" << '\t';

```

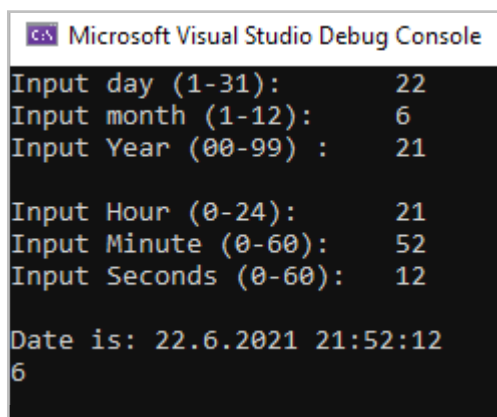
```

cin >> i;
d.SecondNum = i;

cout << endl << "Date is: " << d.DayNum << "."
    << d.MonthNum << ".20" << d.YearNum << " ";
cout << d.HourNum << ":" << d.MinuteNum << ":"
    << d.SecondNum << endl;
cout << sizeof(d) << endl;

return 0;
}

```



```

Microsoft Visual Studio Debug Console
Input day (1-31):      22
Input month (1-12):    6
Input Year (00-99) :   21

Input Hour (0-24):     21
Input Minute (0-60):   52
Input Seconds (0-60):  12

Date is: 22.6.2021 21:52:12
6

```

Рисунок 8

В рамках одной структуры можно сочетать обычные поля и битовые. Однако в этом случае для более оптимального использования памяти (чтобы под всю структуру выделялось меньше памяти для ее хранения) рекомендуется битовые поля располагать подряд и не чередовать их с обычными полями структуры.

Рассмотрим пример объявления битовых полей в структуре, в которой есть и обычные поля.


```

#include <iostream>
using namespace std;

int main()
{
    struct myInfo {
        unsigned char part1:2;
        unsigned char part2:3;
        unsigned char part3:1;
        unsigned char part4:1;
        unsigned char mySymbol;

    };

    myInfo user1;

    cout << "Size of my data is " << sizeof(user1);

    return 0;
}

```

Рассмотрим расположение полей нашей структуры в памяти.

	part4		part3	part2			part1	
byte 0	7	6	5	4	3	2	1	0
	mySymbol							
byte 1	7	6	5	4	3	2	1	0

Рисунок 9

Первое битовое поле (**part1**) занимает самые младшие два бита младшего байта, второе битовое поле (**part2**) — следующие три бита этого же байта. Следующий (пятый) бит этого байта — битовое поле размером в 1 бит **part3**,

следующий (шестой) бит — такое же поле `part4`. 7-й бит младшего байта не участвует в данных вообще, так как следующее поле структуры (`mySymbol`) не битовое.

Поэтому следующее поле `mySymbol` будет занимать следующий (старший) байт.

Таким образом, произойдет небольшое выравнивание, так как суммарное количество битов в битовых полях нашей структуры — семь, а не восемь. То есть если количество битов в непрерывно следующих битовых полях в структуре не кратно восьми, то следующее за ним не битовое обычное поле будет выравниваться по биту следующего адреса.

Как видно из результатов работы программы, для хранения нашей структуры действительно было выделено 2 байта.

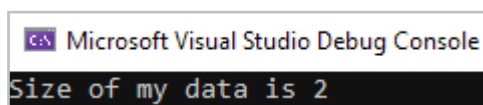


Рисунок 10

5. Работа с файлами

Понятие файла

Файл — это именованная область (совокупность данных), расположенная во внешней памяти (на внешнем устройстве), и обладающая следующими характеристиками:

- имя файла (на определенном диске), которое позволяет программам идентифицировать файл и, при необходимости, работать одновременно с несколькими файлами;
- длина файла может быть ограничена только объемом диска.

Имя файла состоит из его названия и расширения (например, *myData.txt* и *myData.dat* — совершенно разные файлы).

Очень распространенной операцией является получение необходимых данных из файла или сохранение (запись) результатов, полученных в ходе работы программы, в файл.

Полное имя файла представляет собой полный путь к каталогу с файлом, включающий также имя данного файла.

При работе с файлами наши данные хранятся не в оперативной памяти (например, как при использовании массивов), а вне ее, что обеспечивает долговременное хранение данных за пределами программы. В случае отключения питания повреждения данных внутри файла не произойдет, если в этот момент программа не выполняла над ними операции.

Длина файла не является фиксированной, т.е. может уменьшаться и увеличиваться.

В C++ не предусмотрены операторы для работы с файлами. Все действия, необходимые для взаимодействия с файлами, осуществляются с помощью функций, которые есть в стандартной библиотеке (так же, как, например, команда `cout` находится в библиотеке `iostream`, которая должна быть обязательно подключена перед функцией `int main()`). Они позволяют программе взаимодействовать с разными устройствами (например, с диском, принтером и т.д.). Так как данные устройства полностью отличаются друг от друга, то файловая система преобразует их в общее абстрактное логическое устройство — *поток*.

Поток — это последовательность байтов, не зависящая от конкретного устройства, с которым осуществляется взаимодействие (файл на некотором диске, клавиатура, принтер, оперативная память). Для того, чтобы увеличить скорость передачи данных во время работы с потоком, часто используется специальная область оперативной памяти — буфер. В буфере накапливаются байты данных, передача которых производится после его заполнения.

Текстовый поток — представляет собой последовательность символов, которая может состоять из одной или нескольких строк текста. При передаче символа из текстового потока на экран некоторые из символов не отображаются (например, символ новой строки).

Двоичный (бинарный) поток — это последовательность байтов произвольных данных, которые всегда однозначно соответствуют тем байтам информации, которая хранится на внешнем устройстве.

Соответственно, потоки бывают текстовые или бинарные. Однако, независимо от того, к какой категории относится поток, данные в них фактически хранятся в двоичном формате. Просто в текстовых файлах для разделения на строки внутри символьных последовательностей применяется символ «новая строка».

В данном уроке мы рассмотрим классический способ взаимодействия с файлами в C/C++, который основан на использовании библиотеки `stdio.h` (`cstdio`). В этом подходе для доступа к данным используется структура `FILE`. Альтернативным механизмом является работа с потоками и библиотеками `<fstream>`, `<ifstream>`, `<ofstream>`.

Вначале файл нужно описать (как и любой другой объект программы). После чего нам нужно открыть его, создав этим действием связь нашей программы с некоторым внешним носителем информации. После этого можно приступить к обработке файла, например, записывать в него данные, созданные или модифицированные в программе; читать данные из файла; выполнять операции поиска данных и т.д. После завершения работы с файлом мы должны его закрыть.

Программа получает доступ к нужному нам файлу с помощью указателя (файловой переменной) на структуру типа `FILE`, которая определена в файле `stdio.h`. Эта структура содержит необходимые поля для работы с файлами. Например: текущий указатель буфера, текущий счетчик байтов, базовый адрес буфера ввода-вывода, номер файла.

Вначале необходимо создать указатель на структуру `FILE`, который далее инициализируется в результате

открытия файла. Вся последующая работа с файлом будет проводится только через данный указатель.

Общий синтаксис объявления указателя:

```
FILE <имя_указателя>
```

Также возможны ситуации, когда в программе необходимо работать с несколькими файлами. В этом случае объявление данного указателя может выглядеть так:

```
FILE <имя_указателя1>, <имя_указателя2>, ...,  
    <имя_указателяN>
```

Например:

```
FILE *myfile;
```

Обработка файла проводится только после его открытия, после чего описанный в программе указатель файла связывается с конкретным файлом.

Процедура открытия файла реализуется с помощью функции `fopen()`:

```
<имя_указателя_файла> = fopen(<имя_файла>,  
                                <режим_открытия>);
```

- `<имя_указателя_файла>` — указатель на структуру типа `FILE` (объявленный выше в программе);
- `<имя_файла>` — имя файла (строка символов в кавычках), в которое может входить и полный путь к файлу;
- `<режим_открытия>` — строка символов (в кавычках), определяющая, как можно будет далее использовать

файл (). Может принимать значения, представленные в таблице.

В любой операционной системе всегда присутствуют и не удаляются из оперативной памяти следующие потоки: стандартный ввод и вывод, и стандартная ошибка.

Поток создают каналы для передачи данных между, например, файлами (или консолью) и программами.

Например, с помощью стандартного потока ввода мы ранее вводили в программу данные с клавиатуры, а стандартный поток вывода использовали для вывода информации на экран.

Таблица 3. Режимы открытия файлов

Ре- жим	Описание	Обработка данных начи- нается с...
r	Открытие текстового файла только для чтения. Если такого файла не существует, то будет выведена ошибка	Начала файла
w	Открытие текстового файла только для записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено и файл будет перезаписан.	Начала файла
a	Открытие текстового файла для добавления. Если такой файл не существует, то он будет создан. Иначе данные из него будут удалены	Конца файла
r+	Открытие текстового файла для чтения и записи без возможности изменения размера файла. Если такого файла не существует, то будет выведена ошибка.	Начала файла
w+	Открытие текстового файла для чтения и записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено и файл будет перезаписан.	Начала файла

Ре- жим	Описание	Обработка данных начи- нается с...
a+	Открытие текстового файла для чтения и записи. Если такой файл не существует, то он будет создан. Иначе данные из него будут удалены	Конца файла
rb	Открытие двоичного файла для чтения. Если такого файла не существует, то будет выведена ошибка.	Начала файла
wb	Открытие двоичного файла для записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено и файл будет перезаписан.	Начала файла
b	Открытие двоичного файла для добавления. Если такой файл не существует, то он будет создан. Иначе данные из него будут удалены	Конца файла
r+b или b+	Открытие двоичного файла для чтения и записи без возможности изменения размера файла. Если такого файла не существует, то будет выведена ошибка.	Начала файла
w+b или wb+	Открытие двоичного файла для чтения и записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено и файл будет перезаписан.	Начала файла
a+b или ab+	Открытие двоичного файла для чтения и записи. Если такой файл не существует, то он будет создан. Иначе его содержимое будет удалено	Конца файла

В таблице представлены возможные режимы открытия потока в текстовом или бинарном режиме (**b**). Если символ «**b**» не указан в строке, задающий режим, то поток будет открыт в текстовом режиме (по умолчанию).

Допустим, исходные данные для программы расположены в файле с именем *mydata.txt*. Открыть этот файл для чтения можно следующим образом:


```
FILE* myFile;  
myFile = fopen("mydata.txt", "r");
```

В результате выполнения функции `fopen()` переменная `myFile` будет содержать указатель на структуру, отражающую информацию о файле *mydata.txt*. Далее все необходимые действия по работе с этим файлом (*mydata.txt*) будут проводиться только с помощью этого указателя (переменной `myFile`).

Если в результате вызова функции `fopen()` файл по каким-либо причинам не был открыт, то в программу вернется значение `NULL`.

Причины могут следующие: файл, связанный с потоком не найден (при чтении из файла); диск заполнен или защищен (при записи) и т.д.

Результат работы функции `fopen()` — это указатель на структуру типа `FILE`. Данная структура содержит следующие поля:

- дескриптор файла;
- указатель на буфер ввода-вывода и текущее место в нем (откуда надо выдать или куда записать очередной символ);
- счетчик байтов (оставшихся в буфере (при чтении) или свободных (при записи));
- режимы открытия файла и его текущее состояние.

Обращаться из нашей программы к отдельным полям указанной структуры нам нет необходимости, мы просто будем использовать этот указатель как параметр в функциях по работе с данным файлом.

Проверка ситуации, а был ли успешно открыт файл, показана в следующем примере (файл будет открыт в режиме записи):

```
#define _CRT_SECURE_NO_WARNINGS

#include <iostream>
using namespace std;

int main()
{
    FILE* out;
    if ((out = fopen("D:\\examples\\outfile.txt", "w")) ==
        NULL)
        cout << "The file could not be created!";
    else
        cout << "OK!";

    return 0;
}
```

Примечание: если в функции *fopen()* в качестве аргумента используется «w» для уже существующего файла, то его предыдущая версия уничтожается (т.е. предыдущая информация будет потеряна).

Рассмотрим строку, задающую имя файла (первый параметр функции *fopen()*). Если в этой строке содержится только имя файла, то файл создается в текущем каталоге. В этой строке может быть указан полный путь (как в нашем примере), и тогда файл будет создан в указанном каталоге на указанном диске.

Функции, результатом работы которых является указатель, в том числе, *fopen*, считаются небезопасными в

некоторых компиляторах, например, Visual Studio 2019. Если их использование создает ошибку при компиляции, то можно воспользоваться одним из следующих способов для ее предотвращения:

1. Заменить вызов предыдущих версий функций на вызов их безопасных версий, например, заменить `fopen` на `fopen_s`. Но при таком подходе могут измениться параметры в вызове функций, например, придётся использовать:

```
FILE* out;
fopen_s(&out, "data.txt", "w");
```

ВМЕСТО

```
FILE* out = fopen("data.txt", "w");
```

2. В начало файла (до всех `#include`) включить директиву

```
#define __CRT_SECURE_NO_WARNINGS
```

В зависимости от типа (структуры) файла мы выбираем способ работы с ним (чтения и записи). В случае, когда наш файл текстовый, т.е. состоит из символов, разделённых стандартными разделителями (пробел, табуляция, перевод строки), мы можем использовать средства форматированного чтения (`fscanf`) и записи (`fprintf`).

Первым параметром этих функций является файловый указатель, остальные параметры и особенности работы функций — как у стандартных `scanf` и `printf`.

В качестве примера форматированной записи в файл сохраним массив из пяти целых чисел в файле таким

образом, чтобы каждый элемент массива располагался в новой строке:

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

int main()
{
    const int n = 5;
    int arr[n];
    FILE* out;
    const char* path = "D:\\examples\\outfile.txt";

    for (int i = 0; i < n; i++)
    {
        arr[i] = i + 5;
    }

    if ((fopen_s(&out, path, "w")) != NULL)
        cout << "The file could not be created";
    else
    {
        for (int i = 0; i < n; i++)
        {
            fprintf(out, "%1d ", arr[i]);
            fprintf(out, "\n");
        }
        fclose(out);
    }

    return 0;
}
```

Открытые на диске файлы после окончания работы с ними рекомендуется закрыть явно.

```
int fclose(файловый указатель);
```

Для контроля успешного закрытия файла желательно также анализировать возвращаемое значение функции `fclose()`. Функция возвращает значение 0, если файл закрыт успешно, и 1(EOF) в противном случае:

Например,

```
if (fclose(out) == EOF)
    cout << "The file is not closed";
else
    cout << "The file is closed";
```

Если в пользовательской программе не была выполнена операция закрытия файла, то операционная система (ОС) закрывает его сама по завершению программы. Однако, если в программе при этом использовался буферизованный вывод, то некоторая часть данных (особенно при их значительном объеме) может не успеть записаться в файл (останется в системном буфере).

Системный буфер используется ОС для оптимизации операций чтений и записи. Допустим, нам необходимо прочитать или записать в файл 1000000 чисел, это не означает, что система будет 1000000 раз записывать или читать (по одному числу), т.е. ОС фактически не будет 1000000 раз обращаться к диску. Наши числа будут перенесены в буфер с запасом для возможного последующего чтения, или они будут накапливаться в буфере для записи. Запись на диск начнет выполняться только в одной из следующих ситуаций: буфер заполнен; файл (поток) закрыт; программа завершена без закрытия файла.

Теперь рассмотрим пример чтения из файла с помощью функции форматированного ввода `fscanf`.

Работа функции `fscanf()` похожа на функцию `scanf()`, но в отличие от нее форматированный ввод происходит не из консоли, а из файла.

Пример функции `fscanf()`:

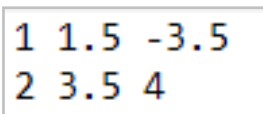
```
fscanf(myfile, "%s%d", str, &a);
```

Первый параметр функции (`myfile`) — файловый указатель, второй ("`%s%d`") — строка формата, остальные (`str, &a`) — адреса областей памяти (переменных) для записи данных.

В нашем примере с помощью функции `fscanf()` мы считываем из некоторого файла (указатель на который после его открытия хранится в переменной `myfile`) два значения: первое как строку (об этом говорит первая часть строки формата "`%s...`"), второе — как целое число (за это отвечает вторая часть строки формата "`...%d`"). Первое значение (строка) будет помещено в переменную `str`, а второе (целое число) в переменную `a`.

Если внутри файла есть символы перехода на новую строку, то они учитываются как разделители данных.

Пусть файл *infile.txt* имеет следующий вид:



```
1 1.5 -3.5
2 3.5 4
```

Рисунок 11

Предположим, что нам необходимо выполнить из него чтение данных.

Выполнять чтение из файла будем, пока не достигнем его конца.

Для контроля достижения конца файла предусмотрена функция `feof` со следующим синтаксисом:

```
int feof(FILE * fileName);
```

Результат работы данной функции целое число. Если оно не равно нулю, то достигнут конец файла.

Так как нам необходимо считать заранее неизвестно количество данных, то проверку окончания файла организуем следующим образом:

```
while (!feof(in)) {  
  
}
```

Очередная операция чтения данных изменяет внутренний файловый указатель и перемещает его на следующее значение, которое будет прочитано.

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    float a;  
    FILE* in;  
    const char* path = "D:\\examples\\infile.txt";  
  
    if (fopen_s(&in, path, "r") != NULL)  
        cout << "The file cannot be opened";  
    else  
    {  
        while (!feof(in)) {  
            fscanf_s(in, "%f", &a);  
            cout << a << " ";  
        }  
    }  
}
```

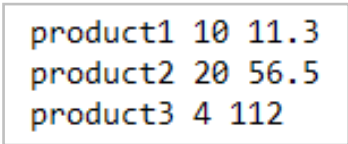
```

    }
}
return 0;
}

```

Рассмотрим еще один пример считывания данных из файла в структуру.

Допустим, у нас есть файл, содержащий такое описание объектов:



```

product1 10 11.3
product2 20 56.5
product3 4 112

```

Рисунок 12

```

#include <iostream>
using namespace std;

struct Item {
    char title[20];
    unsigned int qty;
    float price;
};

int main()
{
    int i = 0;
    FILE* in;
    Item myshop[10];
    const char* path = "D:\\examples\\data.txt";

    if (fopen_s(&in, path, "r") != NULL)
        cout << "The file cannot be opened";
}

```



```

else
{
    while (!feof(in)) {
        fscanf_s(in, "%s", myshop[i].title,
            sizeof(myshop[i].title));
        fscanf_s(in, "%u", &myshop[i].qty,
            sizeof(myshop[i].qty));
        fscanf_s(in, "%f", &myshop[i].price,
            sizeof(myshop[i].price));
        cout << myshop[i].title << " "
            << myshop[i].qty << " "
            << myshop[i].price << "\n";
        i++;
    }
}

return 0;
}

```

В данном примере используется структура и массив структур. Каждая строка из файла должна быть занесена в отдельный элемент массива, который представляет собой структуру с одним строковым и двумя числовыми полями. За один шаг цикл считывает одну строку из файла (заносят в элемент массива). Процесс продолжается, пока не достигнут конец файла.

Так как мы используем буферизованный вывод, то перед выполнением очередной операции чтения рекомендуется очищать буфер (от возможных оставшихся с предыдущих операций данных). Для этого используется метод `fflush()`.

Кроме рассмотренных функций форматированного ввода и вывода для чтения и записи в файл, также

существуют функции посимвольного и построчного чтения или записи.

Рассмотрим вначале функции посимвольного чтения и посимвольной записи данных — `fgetc` и `fputc`.

```
int fgetc(FILE *fp);
```

`fp` — файловый указатель.

Результат работы — очередной символ в формате `int` из потока `fp`. Если символ не может быть прочитан, то возвращается значение `EOF`.

```
int fputc(int c, FILE*fp);
```

`fp` — файловый указатель;

`c` — переменная типа `int`, в которой содержится записываемый в файл символ.

Результат работы — записанный в поток `fp` символ в формате `int`. Если символ не может быть записан, то возвращается значение `EOF`.

Пример

Необходимо прочитать текстовый файл и определить длину каждой строки (в символах). Помним, что в текстовых потоках строки завершаются символом «перевод строки».

```
Here is a wish for your birthday.  
May you receive whatever you ask for, may you find whatever you seek.  
Happy birthday!
```

Рисунок 13

```

#include <iostream>
using namespace std;

int main()
{
    const char* path = "D:\\examples\\text.txt";

    int c;
    int len = 0, cnt = 0;
    FILE* in;

    if (fopen_s(&in, path, "r") != NULL)
        cout << "The file cannot be opened";
    else
    {
        while (!feof(in)) {
            c = fgetc(in);
            if (c == '\\n') {
                cnt++;
                cout << "String " << cnt
                    << " length=" << len
                    << "\\n";
                len = 0;
            }
            else
            {
                len++;
            }
        }
        if (len)
        {
            cnt++;
            cout << "String " << cnt << " length="
                << len << "\\n";
        }
    }
    return 0;
}

```

Результаты работы программы:

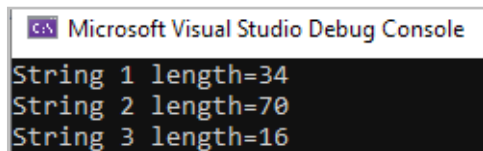


Рисунок 14

Последняя проверка, после цикла, необходима из-за того, что в текстовых потоках последняя строка не заканчивается символом перевода строки и, соответственно, программа «не увидит» ее иначе.

Так как функция `fgetc()` возвращается считанный из файла символ в виде целого значения, то для вывода символов, а не их кодов в консоль, нужно выполнить преобразование из типа `int` в тип `char`.

Изменим предыдущий пример так, чтобы посимвольно вывести текст из файла.

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

int main()
{
    const char* path = "D:\\examples\\text.txt";

    int c;
    int len = 0, cnt = 0;
    FILE* in;

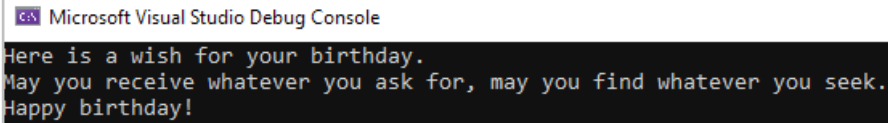
    if (fopen_s(&in, path, "r") != NULL)
        cout << "The file cannot be opened";
    else
    {
```

```

while (!feof(in)) {
    c = fgetc(in);
    if (c == '\n') {
        cout << "\n";
    }
    else
    {
        cout << (char)c;
    }
}
return 0;
}

```

Результат работы программы:



Microsoft Visual Studio Debug Console

```

Here is a wish for your birthday.
May you receive whatever you ask for, may you find whatever you seek.
Happy birthday!

```

Рисунок 15

Для построчного чтения (записи) с указанным максимальным размером в байтах предусмотрены функции `fgets` и `fputs`.

```
char *fgets(char *s, int n, FILE *f);
```

где

`char *s` — адрес, по которому размещаются считанные байты;

`int n` — количество считанных байтов;

`FILE *f` — указатель на файл, из которого производится считывание.

После передачи $n-1$ байтов или при получении символа перевода строки (`'\n'`) прием байтов заканчивается. При этом символ перевода строки также попадет в принимающую строку, которая заканчивается `'\0'`. При успешном завершении считывания функция возвращает указатель на прочитанную строку, при неуспешном — `0`.

```
int fputs(char *s, FILE *f);
```

где

`char *s` — адрес, из которого берутся записываемые в файл байты;

`FILE *f` — указатель на файл, в который производится запись.

В отличие от особенностей работы функции `fgets` при работе `fputs` символ конца строки в файл не записывается.

Пример

Выполним построчное чтение из файла (с предыдущего примера) с известной максимальной длиной строки.

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

int main()
{
    const char* path = "D:\\examples\\text.txt";

    int c;
    int len = 0;
    char buf[128];
    FILE* in;

    if (fopen_s(&in, path, "r") != NULL)
```

```

        cout << "The file cannot be opened";
    else
    {
        while (!feof(in)) {
            fgets(buf, 127, in);
            len = strlen(buf);
            if (buf[len - 1] == '\n')
            {
                buf[len - 1] = '\0';
            }
            puts(buf);
        }
    }
    return 0;
}

```

Microsoft Visual Studio Debug Console

```

Here is a wish for your birthday.
May you receive whatever you ask for, may you find whatever you seek.
Happy birthday!

```

Рисунок 16

Как мы уже знаем, файлы бывают текстовые и бинарные. Однако рассмотренные выше функции предусмотрены только для работы с текстовыми файлами.

Для бинарного файла не предусмотрено использование строк данных и операции чтения и записи подразумевают работу с набором байт указанного размера.

При открытии бинарного файла вторым параметром функции `fopen()` будет строка «`rb`» или «`wb`» (и их альтернативы).

Для чтения и записи двоичных данных основными функциями являются `fread` и `fwrite` соответственно.

В базовой реализации каждой из них есть следующие четыре параметра:

- указатель на место хранения данных;
- размер считываемого или записываемого элемента в байтах;
- максимальное количество элементов, которые требуется прочитать (записать);
- указатель на структуру `FILE`.

Пример

Создадим массив целых чисел и запишем его в бинарный файл.

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

int main()
{
    const char* path = "D:\\examples\\b_data.dat";

    const int n = 5;
    int arr[n];
    FILE* out;

    if (fopen_s(&out, path, "wb") != NULL)
        cout << "The file cannot be opened";
    else
    {
        for (int i = 0; i < 10; i++)
        {
            fwrite(&arr[i], sizeof(int), 1, out);
        }
    }
    return 0;
}
```


Примечание: *учитывая, что данные массива хранятся в памяти последовательно, то цикл, организованный выше для поэлементной записи в массив, может быть заменен всего лишь одним оператором:*

```
fwrite(&arr[0], sizeof(int), n, out);
```

Использование функции `fread()` для чтения бинарных данных аналогично.

```
unsigned char myChar;
//...предположим, что файл уже открыт для чтения
// в режиме "rb"
fread(&myChar, 1, 1, myOut); //читаем по 1 байту

unsigned char myBuf[512];
//...
fread(&myBuf, 1, 512, myOut); // читаем по 1 сектору —
                               // по 512 байт
```

Так как бинарный файл — это последовательная структура данных, то в результате открытия файла будет доступен первый байт хранящейся в нем информации.

Предположим, что нам необходимо считать десятое число, а затем первое. С помощью последовательного доступа можно это реализовать следующим способом:

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

int main()
{
    const char* path = "D:\\examples\\b_data.dat";
```

```

const int n = 10;
int arr[n], a;
FILE* f;

if (fopen_s(&f, path, "wb") != NULL)
    cout << "The file cannot be opened";
else
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = i + 1;
        fwrite(&arr[i], sizeof(int), 1, f);
    }
    fclose(f);
}

if (fopen_s(&f, path, "rb") != NULL)
    cout << "The file cannot be opened";
else
{
    for (int i = 0; i < n; i++)
    {
        fread(&a, sizeof(int), 1, f);
    }
    cout << a << "\n";
    fclose(f);
}

if (fopen_s(&f, path, "rb") != NULL)
    cout << "The file cannot be opened";
else
{
    fread(&a, sizeof(int), 1, f);
    cout << a;
    fclose(f);
}
return 0;
}

```

Чтение чисел из файла подобным образом (в цикле) с последующим повторным открытием файла не самый удобный и компактный способ решения.

Мы можем открыть наш файл в режиме «**r+b**», получив права на чтение, и запись. Такой режим обеспечивает нам произвольный доступ. После этого нам понадобятся функции позиционирования файлового указателя.

Вначале необходимо выполнить чтение текущей позиции указателя в файле, например, с помощью функции **fgetpos()**.

Затем нужно выполнить переход к нужной позиции в файле, например, с помощью функции **fseek()**.

Рассмотрим общий синтаксис функции **fseek**:

```
int fseek(FILE *fileName, long int offset, int origin);
```

fileName — файловый указатель;

offset — смещения (количество байтов, на которые будет смещен указатель файла, начиная с точки начала отсчета);

origin — позиция начала отсчета, которое может быть одним из следующих значений (определены в **stdio.h**):

SEEK_SET — с начала файла;

SEEK_CUR — с текущей позиции;

SEEK_END — с конца файла.

Функция вернет нулевое значение при успешном выполнении операции смещения указателя или ненулевое — в результате возникновения сбоя.

Для организации такого прямого доступа нужно знать месторасположение (номер байта) значения в файле.

Пример

В двоичном файле *b_data.dat* поменять местами наибольшее и наименьшее из вещественных чисел.

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

int main()
{
    const char* path = "D:\\examples\\b_data.dat";

    int n = 10, imax, imin, i;
    int* a, max, min;
    FILE* f;

    if (fopen_s(&f, path, "wb") != NULL)
        cout << "The file cannot be opened";
    else
    {
        for (i = 0; i < n; i++)
        {
            fwrite(&i, sizeof(int), 1, f);
        }
        fclose(f);
    }

    if (fopen_s(&f, path, "rb") != NULL)
        cout << "The file cannot be opened";
    else
    {
        a = new int[n];
        fread(a, sizeof(int), n, f);

        for (imax = imin = 0, max = min = a[0], i = 1;
            i < n; i++)
        {
            if (a[i] > max)
```

```

        {
            max = a[i];
            imax = i;
        }

        if (a[i] < min)
        {
            min = a[i];
            imin = i;
        }
    }

    fseek(f, sizeof(int) + imax * sizeof(double),
          SEEK_SET);

    fwrite(&min, sizeof(double), 1, f);

    fseek(f, sizeof(int) + imin * sizeof(double),
          SEEK_SET);

    fwrite(&max, sizeof(double), 1, f);

    fclose(f);
    delete[] a;
}
return 0;
}

```

Вначале мы открыли бинарный файл в режиме записи и записали в него целые числа от 0 до 9 включительно. Затем открыли бинарный файл для чтения и выполнили чтение чисел из файла в динамический массив `a` (предварительно выделив память для хранения вещественных чисел, которые будут содержаться в массиве). Далее мы выполнили поиск в массиве `a` максимального (`max`) и

минимального (**min**) элемента, определили их индексы (**imax, imin**).

Последним этапом является передвижение указателя файла к позиции максимального значения (**imax**) и запись в эту позицию значения минимального элемента (**min**). Этот же шаг был выполнен и для максимального элемента и позиции **imin**.

6. Поиск файлов

Поиск файлов можно выполнять, используя две функции из библиотеки `io.h`: `_findfirst` и `_findnext`.

Поиск файлов выполняется по двум параметрам (критериям поиска): где искать файл (например, конкретный каталог на диске) и какой именно (каким требованиям, шаблону, должны удовлетворять найденные в результате поиска файлы).

Так как результаты поиска могут содержать не один, а несколько файлов, которые удовлетворяют критерию поиска, то поиск выполняется в два этапа, каждый из которых реализуется с помощью вызова соответствующей функции. Вначале используется функция `findfirst()`, которая находит только первое имя файла, соответствующее шаблону поиска. Далее применяется функция `findnext()`, которая продолжает поиск, начатый функцией `findfirst()`, с теми же параметрами.

Рассмотрим общий синтаксис функции `findfirst()`:

```
long _findfirst(char * path, _finddata_t * fileinfo)
```

где

`path` — символьная строка, которой представляет собой комбинацию пути (где искать файлы) и шаблона-маски (какие файлы искать);

`fileinfo` — указатель на объект структуры `finddata_t`, в который будет записана информация о найденном файле в случае успешного поиска.

Результат работы функции `findfirst()` — это уникальный дескриптор поиска, идентифицирующий файл или группу файлов, которые соответствуют критериям поиска.

Если поиск завершится неудачей, то функция `findfirst()` вернет `-1`.

Задать целевые атрибуты файла (например, найти файлы только для чтения) для ограничения операции поиска файлов невозможно. Эти атрибуты будут возвращены в поле `attrib` структуры `_finddata_t`.

Рассмотрим состав (поля) структуры `_finddata_t`:

- `unsigned attrib` — данные об атрибутах файла, примеры возможных значений (определены в `io.h`):
 - ▷ `_A_NORMAL` — обычный файл, чтение и запись возможны без ограничений.
 - ▷ `_A_RDONLY` — файл только для чтения (невозможно открыть для записи или создать файл с этим именем).
 - ▷ `_A_ARCH` — архивный файл.
- `time_t time_create` — время (и дата) создания файла (равно `-1` для FAT систем) в формате UTC.
- `time_t time_access` — время (и дата) последнего открытия файла (равно `-1` для FAT систем) в формате UTC.
- `time_t time_write` — время (и дата) последнего редактирования (последней записи) файла.
- `_fsize_t size` — размер файла (его длина в байтах).
- `char name[260]` — имя файла.

Давайте рассмотрим пример использования функции `findfirst()` — пусть необходимо найти первый файл с расширением `cpp` в текущем каталоге и вывести его имя:


```

#include <iostream>
#include <io.h>

using namespace std;

int main()
{
    struct _finddata_t myfileinfo;

    long done = _findfirst("*.cpp", &myfileinfo);

    cout << myfileinfo.name;

    return 0;
}

```

Если поиск будет успешным (т.е. в текущем каталоге существует хотя бы один файл с расширением `cpp`), то информация об этом первом найденном файле будет записана в объект структуры `_finddata_t myfileinfo`.

Для задания маски поиска можно использовать любые комбинации всех подстановочных знаков, которые поддерживаются текущей операционной системой (например, `*` и `?`).

После работы функции `findfirst()` в оперативной памяти будет сформирован некоторая группа объектов (с уникальным номером, возвращаемым функцией `findfirst`) структуры `_finddata_t` с собственным внутренним указателем, который в начале установлен на объект, ассоциируемый с первым найденным файлом.

Для продолжения поиска (в случае, когда несколько файлов удовлетворяют критериям поиска) нам нужно

после вызова функции `findfirst()` вызвать функцию `findnext()` и передать ей в качестве параметра результаты работы функции `findfirst()`, а именно переменную, которая содержит указатель на группу объектов структуры `_finddata_t` (результат работы функции `findfirst()`).

Рассмотрим общий синтаксис функции `findnext()`.

```
int findnext(long done, _finddata_t * fileinfo)
```

Один вызов данной функции реализует переход на следующий найденный файл в группе, полученной в результате работы функции `findfirst()`. Соответственно, данную функцию обычно вызывают в цикле (пока результат работы функции не равен `-1`, ситуация достижения конца списка файлов из группы).

Параметры функции:

- `done` — уникальный номер группы файлов в памяти (результат работы функции `findfirst()` на первом этапе процедуры поиска).
- `fileinfo` — указатель на объект структуры `_finddata_t`, в который будет записана информация о найденном файле в случае успешного поиска.

В случае успешного поиска (в группе файлов, полученной в результате работы функции `findfirst()` еще есть файлы) функция возвращает значение `0`, иначе `-1`.

Рассмотрим несколько примеров на использование функций `findfirst()` и `findnext()`.

На рисунке 17 представлено содержимое каталога `examples`.

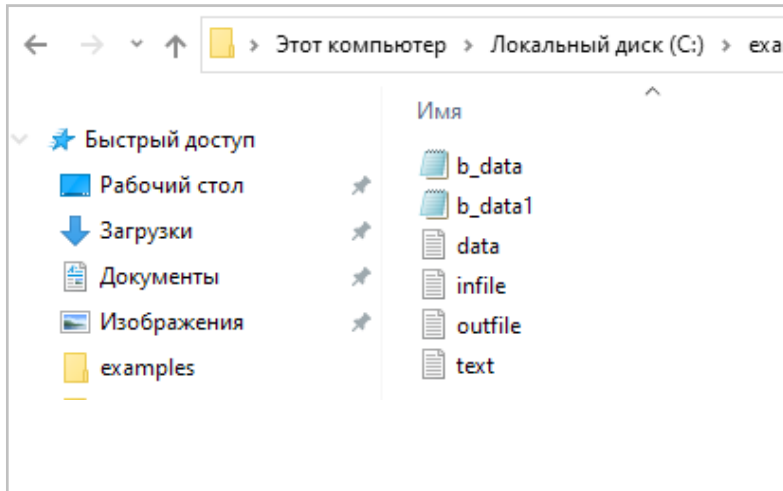


Рисунок 17

Давайте предположим, что нам необходимо найти первый файл на диске **D** в каталоге **examples** с расширением **.dat**

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

using namespace std;

int main()
{
    struct _finddata_t fileinfo;
    char str[200] = "D:\\examples\\*.dat";

    long done = _findfirst(str, &fileinfo);
    cout << fileinfo.name;

    return 0;
}
```

Результат работы программы:

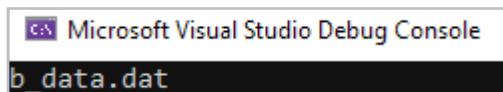


Рисунок 18

Так как нам нужен только первый файл, то достаточно вызова функции `findfirst()`. Имя файла содержится в поле `name` структуры `finddata_t` (которая была заполнена в результате работы функции `findfirst()`).

Теперь добавим в наш пример проверку успешности результата поиска, иначе будем выводить сообщение об ошибке.

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

using namespace std;

int main()
{
    struct _finddata_t fileinfo;
    char str[200] = "D:\\examples\\*.dat";
    long done = _findfirst(str, &fileinfo);

    if (done == 1)
    {
        cout << "The search result is unsuccessful";
    }
    else
    {
        cout << fileinfo.name;
    }
    return 0;
}
```

В следующем примере мы выведем все файлы с расширением *.dat* из данного каталога (используем последовательно вызовы функций `findfirst` и `findnext` — в цикле).

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

using namespace std;

int main()
{
    struct _finddata_t fileinfo;
    char str[200] = "D:\\examples\\*.dat";

    long done = _findfirst(str, &fileinfo);

    while (done != -1) {
        cout << fileinfo.name << "\n";
        done = _findnext(done, &fileinfo);
    }

    return 0;
}
```

Результат работы программы:

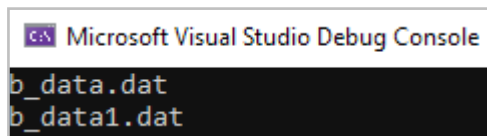


Рисунок 19

В следующем примере мы предложим пользователю ввести путь к месту поиска файла (где искать) и шаблон (маску) для поиска.

Однако, функция `_findnext` имеет только один параметр для задания пути и маски, поэтому нам необходимо предварительно, до ее вызова, выполнить объединение строки, которая задает путь, и строки, содержащую маску. Объединение строк можно реализовать, например, используя функцию `strcat`.

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

using namespace std;
int main()
{
    struct _finddata_t fileinfo;
    char path[100];
    char mask[20];

    // Запросим путь
    cout << "Enter full path (for example, D:\\):\\n";
    cin >> path;

    // Запросим маску файлов
    cout << "Enter mask (for example, *.* or *.txt):";
    cin >> mask;

    // Соединим две строки: путь и маску
    strcat_s(path, mask);

    long done = _findfirst(path, &fileinfo);

    while (done != -1) {
        cout << fileinfo.name << "\\n";
        done = _findnext(done, &fileinfo);
    }
    return 0;
}
```

Результат работы программы:

```
Enter full path (for example, D:\):  
C:\examples\  
Enter mask (for example, *.* or *.txt):*.dat  
b_data.dat  
b_data1.dat
```

Рисунок 20