ЦИКЛЫ. ОПЕРАТОРЫ ПЕРЕХОДА (окончание).

В языке C/C++ предусмотрены четыре оператора безусловного перехода: `return`, `goto`, `break`, `continue`.

Операторы `return` и `goto` можно применять в любом месте программы, в то время как операторы `break` и `continue` связаны с операторами циклов. Кроме того, оператор `break` можно применять внутри оператора `switch`.

Оператор return

Этот оператор используется для возврата управления из функции. Он относится к операторам безусловного перехода (jump operators), поскольку выполняет возврат в точку вызова функции. С ним может быть связано определенное значение, хотя это и не обязательно. Если оператор `return` связан с определенным значением, оно становится результатом функции.

```
return выражение;
```

Главная функция в программе, функция main(), должна обязательно содержать оператор `return` вида:

```
return 0;
```

«сообщающий» операционной системе о благополучном завершении вашей программе.

Оператор goto

`Goto` позволяет осуществить «прыжок» в другую точку программы. Этот безусловный переход игнорирует уровни вложенности. Поэтому этот оператор следует использовать с осторожностью и предпочтительно в пределах одного блока операторов, особенно при наличии локальных переменных.

Конечная точка (точка «прыжка») определяется *меткой*, которая затем используется в качестве аргумента для оператора goto. Метка является допустимым идентификатором языка C++, за которым следует двоеточие (:).

`Goto` считается низкоуровневым оператором, который НЕ рекомендуется использовать без особой надобности. Но, как пример, вот версия цикла обратного отсчета (от 10 до старта) с использованием `goto`:

```
// goto loop example
#include <iostream>
using namespace std;

int main ()
{
  int n=10;
  mylabel:
  cout << n << ", ";
  n--;
  if (n>0) goto mylabel;
  cout << "liftoff!\n";
}
```

Оператор break

Оператор `break` применяется в двух ситуациях:

– для прекращения выполнения раздела `case` внутри оператора `swich`;

– для немедленного выхода из цикла независимо от истинности или ложности его условия.

Если в цикле встречается оператор `break`, итерации прекращаются и выполнение программы возобновляется с оператора, следующего за оператором цикла. Рассмотрим пример:

```
#include <iostream>
int main() {
   for (int t=0; t<100; t++) {
      cout << t;
      if (t==10) break;
   }
   return 0;
```

```
}
```

Эта программа выводит на экран числа от 0 до 10, а затем цикл прекращается, поскольку выполняется оператор `break`. Условие `t<100` при этом игнорируется.

Оператор `break` часто применяется в циклах, выполнение которых следует немедленно прекратить при наступлении определенного события. В приведенном ниже примере нажатие клавиши `Enter` прекращает выполнение функции `main()`:

```
int main ()
{
   do {
       /* что-то делаем ... */
      if (std::cin.get()) // ожидаем ввод пользователя с клавиатуры
         break;           // если нажат Enter, то ...
   } while(!found);
    /* Точка выхода */      // ...выходим из цикла независимо
                            // от истинности условия !found
}
```

Если циклы вложены друг в друга, оператор `break` выполняет выход из внутреннего цикла во внешний. Например, приведенная ниже программа 100 раз выводит на экран числа от 1 до 10, причем каждый раз, когда счетчик достигает значения 10, оператор `break` передает управление внешнему циклу `for`:

```
for (t=0; t<100; ++t) {
   count = 1;
   for ( ; ; ) {  // бесконечный цикл
     cout << count;
     count++;
     if (count==10) break;
   }
}
```

Если оператор `break` содержится внутри оператора `switch` , который вложен в некий цикл, то выход будет осуществлен только из оператора `switch` , а управление останется во внешнем цикле.

Оператор continue

Оператор `continue` напоминает оператор `break`. Они различаются тем, что оператор `break` прекращает выполнение всего цикла, а оператор `continue` – лишь его текущей итерации, вызывая переход к следующей итерации и пропуская все оставшиеся операторы в теле цикла. В цикле `for` оператор `continue` вызывает проверку условия и приращение счетчика цикла. В циклах `while` и `do-while` оператор `continue` передает управление операторам, входящим в условие цикла.

Приведенная ниже программа подсчитывает количество пробелов в строке, введенной пользователем.

```
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

Результат:

```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

ВЫРАЖЕНИЯ В С++

Выражение – это последовательность операторов и их операндов, задающих вычисления. Выражения состоят из операторов, констант, функций и переменных. В языке С/С + + выражением считается любая допустимая комбинация этих элементов. Поскольку большинство выражений в языке С /С + + напоминают алгебраические, часто их таковыми и считают. Но есть и специфические особенности. Однако главное то, что в процессе вычислений формируется *результат* (например, результатом вычисления 2+2 является число 4),

Порядок выполнения операций: приоритеты операций и ассоциативность

1. С++ руководствуется правилами *приоритетов*, чтобы определить, какая операция должна быть выполнена первой. Например:

```
int flyingpigs = 3 + 4*5; // каким будет результат: 35 или 23?
```

Получается, что операнд 4 может участвовать и в сложении, и в умножении. Когда над одним и тем же операндом может быть выполнено несколько операций, С++ руководствуется правилами старшинства или приоритетов, чтобы определить, какая операция должна быть выполнена первой.

Арифметические операции выполняются в соответствии с алгебраическими правилами, согласно которым умножение, деление и нахождение остатка от деления выполняются раньше операций сложения и вычитания. Поэтому выражение `3 + 4*5` следует читать как `3+ (4 * 5)`, но не `(3 + 4) * 5`. Таким образом, результатом этого выражения будет 23, а не 35.

Для смены порядка вычислений необходимо сменить приоритет операции. Для этого используются круглые скобки. Однако в С++ заложены приоритеты и ассоциативность всех операций (см. https://en.cppreference.com/w/cpp/language/operator_precedence).

Контрольное задание 1:

- изучить таблицу приоритетов и ассоциативности всех операций – см. https://en.cppreference.com/w/cpp/language/operator_precedence;
- скопировать указную таблицу в конспект лекции с указанием описания перечисленных операций/оператор на русском языке;
- привести примеры использования известных вам операций/операторов. Для этого дополнить таблицу приоритетов новым столбцом «Пример».

Дополнительное описание основных операций см. на www.cplusplus.com/doc/tutorial/operators/

Обратите внимание, что в таблице приоритетов операции *, / и % занимают одну строку. Это означает, что они имеют одинаковый уровень приоритета. Операции сложения и вычитания имеют более низкий уровень приоритета (в таблице приоритеторв они расположены строкой ниже).

2. Однако знания только лишь приоритетов операций не всегда бывает достаточно.

Взгляните на следующий оператор:

```
float logs = 120 /4*5; // каким будет результат: 150 или 6?
```

И в этом случае над операндом 4 могут быть выполнены две операции. Однако операции /и * имеют одинаковый уровень приоритета, поэтому программа нуждается в уточнении, чтобы определить, нужно сначала 120 разделить на 4 либо 4 умножить на 5. Поскольку результатом первого варианта является 150, а второго – 6, выбор здесь очень важен. В том случае, когда две операции имеют одинаковый уровень приоритета, С++ анализирует их *ассоциативность*: слева направо или справа налево.

Ассоциативность слева направо означает, что если две операции, выполняемые над одним и тем же операндом, имеют одинаковый приоритет, то сначала выполняется операция слева от операнда. В случае ассоциативности справа налево первой будет выполнена операция справа от операнда.

Сведения об ассоциативности – в табл. выше. Здесь показано, что для операций умножения и деления характерна ассоциативность слева направо. Это означает, что над операндом 4 первой будет выполнена операция слева. То есть 120 делится на 4, в результате получается 30, а затем этот результат умножается на 5, что в итоге дает 150.

3. Следует отметить, что приоритет выполнения и правила ассоциативности действуют только тогда, когда две операции относятся к одному и тому же операнду.

Рассмотрим следующее выражение:

```
int dues = 20 * 5 + 24 * 6;
```

В соответствии с приоритетом выполнения операций, программа должна умножить 20 на 5, затем умножить 24 на 6, после чего выполнить сложение. Однако ни уровень приоритета выполнения, ни ассоциативность не помогут определить, какая из операций умножения должна быть выполнена в первую очередь. Можно было бы предположить, что в соответствии со свойством ассоциативности должна быть выполнена операция слева, однако в данном случае две операции умножения не относятся к одному и тому же операнду, поэтому эти правила здесь не могут быть применены. В действительности выбор порядка выполнения операций, который будет приемлемым для системы, оставлен за конкретной реализацией C++. В данном случае при любом порядке выполнения будет получен один и тот же результат, однако иногда результат выражения зависит от порядка выполнения операций.

Порядок вычислений функций

Ни в языке С, ни в языке С + + порядок вычисления подвыражений, входящих в выражения, не определен. Компилятор может сам перестраивать выражения, стремясь к созданию оптимального объектного кода. Это означает, что программист не должен полагаться на определенный порядок вычисления подвыражений. Скажем, при вычислении выражения `x = f1() + f2();` нет никаких гарантий, что функция `f1()` будет вызвана раньше функции `f2()`.

Компилятор может вычислять операнды и подвыражения в любом порядке и может выбирать другой порядок вычислений при повторном определении значения этого же выражений. Правил вычисления «слева-направо» или «справа-налево», аналогичных ассоциативности «слева-направо» или «справа-налево», в C++ нет. Выражение `a()+b()+c()` вычисляется как `(a()+b())+c()` согласно ассоциативности «слева-направо» оператора `+`. Однако вызов функции `c()` может быть выполнен до вызовов функции `a()` или `b()`.

Неопределённый результат выражений [1]

В C++ существуют и другие выражения, результат которых не опеределён:
https://en.cppreference.com/w/cpp/language/eval_order#Undefined_behavior_2

[1] Материал повышенного уровня сложности. Изучается по желанию.

ПРОСТЫЕ ТИПЫ ДАННЫХ

Значения переменных хранятся где-то в памяти в виде нулей и единиц. Программа нет необходимости знать точное расположение значений в памяти, т.к. для обращения к ним (значениям) используется имя переменной. То, что нужно знать программе обязательно, – это вид хранимой информации, т.к. от этого зависит её способ представления в памяти. В программе вид информации задаётся типом переменных.

Базовым типам данных являются:

- Символьный тип – для представления одиночных символов (символьных литералов).
- Целый тип – для хранения целых чисел знаковых (signed) и беззнаковых (unsigned).
- Вещественный тип – для представления дробей
- Логический тип

В стандарте языка C++ минимальный размер и диапазон изменения переменных, имеющих элементарный тип, не определен. Вместо этого в нем просто указано, что они должны соответствовать определенным условиям. Например, стандарт требует, чтобы переменная типа int имела "естественный размер, соответствующий архитектуре операционной системы". В любом случае, ее диапазон должен совпадать или превосходить диапазон изменения переменной данного типа, предусмотренный стандартом языка C. Каждый компилятор языка C++ задает размер и диапазон изменения переменных всех элементарных типов в заголовке <climits> .

Вид информации	Тип C++*	Примечания	Префикс/суффикс констант (Z)
Символы	**char**	Exactly one byte in size. At least 8 bits.	-
	char16_t	Not smaller than char. At least 16 bits.	u'N' или u"qwe"
	char32_t	Not smaller than char16_t. At least 32 bits.	U'N'
	wchar_t	Can represent the largest supported character set.	L'N'
Целые знаковые числа	**signed char**	Same size as char. At least 8 bits.	
	signed **short** *int*	Not smaller than char. At least 16 bits.	
	signed **int**	Not smaller than short. At least 16 bits.	
	signed **long** *int*	Not smaller than int. At least 32 bits.	5l или 5L
	signed **long long** *int*	Not smaller than long. At least 64 bits.	5ll или 5LL
Беззнаковые значения	**unsigned char**	(same size as their signed counterparts)	
	unsigned short *int*		
	unsigned *int*		5u или 5U
	unsigned long *int*		5ul, 5UL, 5Ul, 5uL
	unsigned long long *int*		5ull или 5ULL
Вещественные значения	**float**		5.0f или 5.0F
	double	Precision not less than float	-
	long double	Precision not less than double	5.0l или 5.0L
Логический тип	**bool**		

* Числовые типы могут именоваться без `signed` и `int` (части указанные курсивом необязательны). Например, вместо `signed short int` можно записать `signed short`, `short int`, или просто `short`.

Типы в каждой группе отличаются только размером, что влияет на диапазон представимых данных: первый самый наименьший, каждый последующий не менее предыдущего. Остальные характеристики одинаковы (табл. ниже).

Размер	Диапазон представимых данных	Примечания
8-bit	0..256 (или -128..127)	$= 2^8$
16-bit	0..65 536 (или -32768..32767)	$= 2^{16}$
32-bit	4 294 967 296	$= 2^{32}$ (~4 billion)
64-bit	18 446 744 073 709 551 616	$= 2^{64}$ (~18 billion billion)

Среди всех базовых типов только тип `char` имеет определённый размер в 1 байт (или самое большее минимальный размер). Это не означает, что типы безразмерны. Это показывает то, что каждый компилятора выбирает наиболее подходящий размер типа в зависимости от архитектуры машины, на которой будет выполняться программа. Такой подход в определении размеров типов делает язык C++ довольно гибким и адаптированным для оптимальной работы во всех видах платформ, как в настоящем, так и в будущем.

Размер вещественных типов влияет на точность представимых чисел.

Основными из всех перечисленных типов являются `char`, `int` и `double`. Именно их в основном и рекомендуется использовать. Другие типы предназначены для узкого класса задач.

Типы: символьные, численные и логический, – известны также под названием *арифметических* типов. Но есть ещё два базовых типа `void`, который определяет отсутствие типа, и тип `nullptr`, который является специальным типом указателя. Про них подробнее будет сказано в следующих лекциях.

Минимальный размер и диапазон изменения переменных числовых типов данных согласно стандарту языка C++ даны в описании лабораторной работы №4 «Простые типы данных» а табл. 3 и 4.

Символы

Хранение чисел в памяти компьютера не представляет сложности, тогда как хранение букв связано с рядом проблем. В языках программирования принят простой подход, при котором для букв используются числовые коды.

Таким образом, тип `char` является еще одним целочисленным типом. Для целевой компьютерной системы гарантируется, что `char` будет настолько большим, чтобы представлять весь диапазон базовых символов — все буквы, цифры, знаки препинания и т.п. На практике многие системы поддерживают менее 128 разновидностей символов, поэтому одиночный байт может представлять весь диапазон. Следовательно, несмотря на то, что тип `char` чаще всего служит для хранения символов, его можно применять как целочисленный тип, который обычно меньше, чем `short`.

Самым распространенным набором символов в США является ASCII. Каждый символ в этом наборе представлен числовым кодом (ASCII-кодом). Например, символу A соответствует код 65, символу M – код 77 и т.д.

Ни ASCII, ни EBCDIC (код в мэйнфреймах IBM) не в состоянии полностью представить все интернациональные символы, поэтому в C++ поддерживается расширенный символьный тип, способный хранить более широкий диапазон значений (таких как входящие в международный набор символов Unicode). Для этого используется тип `wchar_t`.

Пример.

```
char ch = 'M';     // присваивает ch код ASCII символа M
int i = ch;        // сохраняет этот же код в int
cout << "The ASCII code for " << ch<< " is " << i << endl;
cout << "Add one to the character code:" << endl;
ch = ch + 1;       // изменяет код символа в ch
i = ch;            // сохраняет код нового символа в i
cout << "The ASCII code for " << ch << " is " << i << endl;
// Использование функции-члена cout.put() для отображения символа
cout << "Displaying char ch using cout.put (ch) : ";
cout.put(ch);  cout.put('!');
```

Результат:

```
The ASCII code for M is 77
Add one to the character code:
The ASCII code for N is 78
Displaying char ch using cout.put (ch) : N!
```

Замечания. В программе обозначение `'M'` представляет числовой код символа M, поэтому инициализация переменной `ch` типа `char` значением `'M'` приводит к ее установке в 77. Затем такое же значение присваивается переменной `i` типа `int`, так что обе переменных имеют значение 77. Далее объект `cout` отображает значение `ch` как M, а значение `i` – как 77. Ранее уже утверждалось,

что объект cout выбирает способ отображения значения на основе его типа – еще один пример поведения интеллектуального объекта cout.

Поскольку переменная ch на самом деле является целочисленной, над ней можно выполнять целочисленные операции, такие как добавление 1. В результате значение ch изменится на 78. Это новое значение затем было присвоено переменной i. (Эквивалентный результат дало бы прибавление 1 к i.) И в данном случае объект cout отображает вариант char этого значения в виде буквы и вариант int в виде числа.

И, наконец, с помощью функции cout.put() в программе отображается значение переменной ch и символьная константа.

Факт представления символов в C++ в виде целых чисел является большим удобством, поскольку существенно облегчается манипулирование символьными значениями. Никаких функций для преобразования символов в коды ASCII и обратно использовать не придется.

Даже цифры, вводимые с клавиатуры, читаются как символы. Например:

```
char ch;
cin >> ch; // читает символ и сохраняет его ASCII-код в ch
```

Если вы наберете 5 и нажмете клавишу <Enter>, этот фрагмент прочитает символ 5 и сохранит код для символа 5 (53 в ASCII) в переменной ch. Например:

```
int n;
cin >> n; // читает символ и сохраняет перевод в соответствующее число
```

Тот же самый ввод приводит к тому, что программа прочитает символ 5 и запустит подпрограмму, преобразующую символ в соответствующее числовое значение 5, которое и сохранится в переменной n.

Для определения принадлежности символьной константы определённому типу с стандарте C++ указаны префиксы типов:

'c-char'	Без префикса	символ типа char
u'c-char'	C++11	символ типа char16_t
U'c-char'	C++11	символ типа char32_t
L'c-char'		символ типа wchar_t

Пример.

```
char ch='M';
char16_t c16=u'M';
char32_t c32=U'M';
wchar_t wc=L'M';
std::cout << ch  <<" sizeof char - "     << sizeof (ch)  <<'\n'
          << c16 << " sizeof char16_t - "<< sizeof (c16) <<'\n'
          << c32 << " sizeof char32_t - "<< sizeof (c32) <<'\n'
          << wc  << " sizeof wchar_t - " << sizeof (wc)  <<'\n'
```

Результат:

```
M sizeof char - 1
77 sizeof char16_t - 2
77 sizeof char32_t - 4
77 sizeof wchar_t  - 2
```

Контрольное задание 2: Выполнить задания по материалам ресурса https://en.cppreference.com/w/cpp/language/ascii:

- изучить приводимую таблицу ASCII-символов
- изучить приведённую программу, печатающую таблицу ASCII-символов;
- изменить приведённую программу так, чтобы таблица ASCII-символов печаталась в 8 колонок
- проверить работоспособность программы с помощью встроенного редактора кода.

Целочисленные типы (продолжение)

Суффиксы целочисленных литералов:

Вид информации	Тип C++	Суффикс констант, пример
Целые знаковые числа	*signed* **short** *int*	
	signed **int**	
	signed **long** *int*	**5l** или **5L**
	signed **long long** *int*	**5ll** или **5LL**
Беззнаковые значения	**unsigned short** *int*	
	unsigned *int*	**5u** или **5U**
	unsigned long *int*	**5ul, 5UL, 5Ul, 5uL**
	unsigned long long *int*	**5ull** или **5ULL**

Пример.

```
//префиксы, определяющие систему счисления
int d = 42;
int o = 052;
int x = 0x2a;
int X = 0X2A;
int b = 0b101010;
//суффиксы, задающие длину
std::cout << 123    << '\n'
          << 0123   << '\n'
          << 0x123  << '\n'
          << 0b10   << '\n'
          << 12345678901234567890ull << '\n'
          << 1234567890123456789u   << '\n'; // тип unsigned long long
//  std::cout << -9223372036854775808 << '\n';
// ошибка: значение 9223372036854775808 не может быть
// представлено типом signed long long, т.к. целые числа
// без суффиксов автоматически преобразуются в тип int
     std::cout << -9223372036854775808u << '\n'; // получили 9223372036854775808
     std::cout << -9223372036854775807 - 1 << '\n'; // тот же результат
```

Вещественные типы

Эти типы характеризуются количеством значащих цифр, которые они могут представлять, и минимальным допустимым диапазоном порядка. Значащими цифрами являются значащие разряды числа: 14.32 – четыре значащие цифры, 14000 – две значащие цифры.

Требования в C и C++ относительно количества значащих разрядов следующие:

- `float` должен иметь, как минимум, 32 бита,
- `double` – как минимум, 48 битов и естественно быть не меньше чем `float`,
- `long double` должен быть минимум таким же, как и тип `double`.

Все три типа могут иметь одинаковый размер. Однако обычно `float` занимает 32 бита, `double` — 64 бита, а `long double` — 80, 96 или 128 битов.

Кроме того, диапазон порядка для каждого из этих трех типов — как минимум, от -37 до +37.

Ограничения для конкретной системы можно узнать из файла `cfloat` или `float.h` (см. описание лабораторной работы №4)

Когда в программе записывается константа с плавающей точкой, то с каким именно типом она будет сохранена? По умолчанию константы с плавающей точкой, такие как 8.24 и 2.4E8, имеют тип `double`. Если константа должна иметь тип `float`, необходимо указать суффикс f или F. Для типа

long double используется суффикс 1 или L. (Поскольку начертание буквы 1 в нижнем регистре очень похоже на начертание цифры 1, рекомендуется применять L в верхнем регистре.) Ниже приведены примеры использования суффиксов:

```
1.234f // константа float
2.45E20F    // константа float
2.345324E28 // константа double
2.2L   // константа long double
```

Преимущества вещественных типов:

– представляют значения, расположенные между целыми числами;

– представляют более широкий диапазон значений за счёт масштабного коэффициента.

Пример 1.

```
float tub = 10.0 / 3.0;    // подходит для 6 разрядов
double mint = 10.0 / 3.0;  // подходит для 15 разрядов
const float million = 1.0e6;
cout << fixed             // задаём вывод в формате с фиксированной запятой
     << "tub = " << tub;
cout << ", a million tubs = " << million * tub;
cout << ",\nand ten million tubs = ";
cout << 10 * million * tub << endl;
cout << "mint = " << mint << " and a million mints = ";
cout << million * mint << endl;
```

Результат

```
tub = 3.333333, a million tubs = 3333333.250000,
and ten million tubs = 33333332.000000
mint = 3.333333 and a million mints = 3333333.333333
```

Пояснения к примеру:

Обычно объект cout отбрасывает завершающие нули. Например, он может отобразить 3333333.250000 как 3333333.25. Использование манипулятора fixed переопределяет это поведение, во всяком случае, в новых реализациях. Главное – тип float имеет меньшую точность, чем тип double. Переменные tub и mint инициализируются выражением 10.0/3.0, результат которого равен 3.33333333333333333... (3 в периоде). Поскольку cout выводит шесть цифр справа от десятичной точки, вы можете видеть, что значения tub и mint отображаются с довольно высокой точностью. Однако после того как каждое число умножается на миллион, вы увидите, что значение tub отличается от правильного результата после седьмой тройки, tub дает хороший результат до седьмой значащей цифры. (Эта система гарантирует 6 значащих цифр для float, по это самый худший сценарий.) Переменная типа double показывает 13 троек, поэтому она дает хороший результат до 13-й значащей цифры. Поскольку система гарантирует 15 значащих цифр, то такой и должна быть точность этого типа. Обратите также внимание, что умножение произведения million и tub на 10 дает не совсем точный результат; это еще раз указывает на ограничения по точности типа float.

Пример 2. Недостатки вещественных типов рассмотрим на примере

```
float a = 2.34E+22f;
float b = a + 1.0f;
cout << "a = " << a << endl;
cout << "b - a = " << b - a << endl;
```

Результат:

```
a = 2.34e+022
b - a = 0
```

Проблема в том, что 2.34E+22 представляет число с 23 цифрами слева от десятичной точки. За счет прибавления 1 происходит попытка добавления 1 к 23-й цифре этого числа. А тип float может представлять только первые 6 или 7 цифр числа, поэтому попытка изменить 23-ю цифру не оказывает никакого воздействия на значение.

Пример 3. Опять точность вычислений

```
cout << fixed;
cout << "Integer division: 9/5 = " << 9 / 5 << endl;
```

```
cout << "Floating-point division: 9.0/5.0 = ";
cout << 9.0 / 5.0 << endl;
cout << "Mixed division: 9.0/5 = " << 9.0 / 5 << endl;
cout << "double constants: 1e7/9.0 = ";
cout << 1.e7 / 9.0 << endl;
cout << "float constants: 1e7f/9.0f = ";
cout << 1.e7f / 9.0f << endl;
```

Результат:

```
Integer division: 9/5 = 1
Floating-point division: 9.0/5.0 = 1.800000
Mixed division: 9.0/5 = 1.800000
double constants: 1e7/9.0 = 1111111.111111
float constants: 1e7f/9.0f = 1111111.125000
```

Примечания к примеру:

Первая строка вывода показывает, что в результате деления целого числа 9 на целое число 5 было получено целое число 1. Дробная часть от деления 4/5 (или 0.8) отбрасывается. Следующие две строки показывают, что если хотя бы один из операндов имеет формат с плавающей точкой, искомый результат (1.8) также будет представлен в этом формате. В действительности, при комбинировании смешанных типов C++ преобразует их к одному типу. Относительная точность в двух последних строках вывода показывает, что результат имеет тип `double`, если оба операнда имеют тип `double`, и тип `float`, если оба операнда имеют тип `float`. Не забывайте, что константы в формате с плавающей точкой имеют тип `double` по умолчанию.

Логический тип

Тип `bool` назван честь английского математика Джорджа Буля (George Boole), разработавшего математическое представление законов логики.

Значения булевской переменной: `true` (истина) или `false` (ложь) – литералы.

Ранее в языке C++, как и в C, булевский тип отсутствовал. Вместо этого C++ интерпретировал ненулевые значения как `true`, а нулевые – как `false`.

Пример 1

```
bool is_ready = true;
// Литералы true и false могут быть преобразованы в тип int
int ans = true; // ans присваивается 1
int promise = false; // promise присваивается 0
// Неявное преобразование в значение bool.
bool start = -100; // start (любое ненулевое значение) присваивается true
bool stop =0; // stop (только нуль) присваивается false
```

Пример 2. Для управления форматом вывода логических значений можно использовать манипуляторы:

```
std::cout << std::boolalpha
          << true << '\n'          // выведет true
          << false << '\n'         //         false
          << std::noboolalpha
          << true << '\n'          // выведет 1
          << false << '\n';        //         0
```

Выполняем лабораторную работу №4 «Простые типы».