Язык программирования Си

Бикулов Д.А., Иваницкая Н.В., Иванов А.П.

Семинар 3. Операторы в выражениях, приоритет операторов, преобразование типа. Директивы препроцессора и их использование. Массивы. Оператор sizeof(). Оператор typedef. Математические функции стандартной библиотеки Си.

1 Операторы в выражениях, приоритет операторов, преобразование типа

Как и многие другие языки программирования, язык Си содержит развитый набор операторов, которые можно использовать для вычисления результатов арифметических и логических выражений. Круглые скобки управляют порядком выполнения операций.

Примеры выражений.

```
int x = 1, y = 2, z, t;

z = t = 0;

z = (x + y) / 2;
t = (x + y) % 2;

bool q = (x < y) && (z <= y);

z = x += 8; y *= x++; z++;
```

1.1 Приоритет операторов

Если в одном выражении встречается несколько операторов, то они выполняются в порядке убывания приоритета. Чем меньше номер группы операторов в таблице ниже, тем выше приоритет этой операции. В рамках одной группы операторы выполняются справа налево.

1	**x++** **x--**	постфиксный инкремент постфиксный декремент
2	**++x** **--x** **!x** **&x** ***p**	префиксный инкремент префиксный декремент логическое отрицание получение адреса переменной разыменование указателя
3	**x * y** **x / y** **x % y**	умножение деление остаток от деления

4	`x + y` `x - y`	сложение вычитание

5	`x < y`
	`x <= y`
	`x > y`
	`x >= y`
		сравнение «меньше»
		сравнение «меньше или равно»
		сравнение «больше»
		сравнение «больше или равно»

6	`x == y` `x != y`	сравнение «равно» сравнение «не равно»
7	`x && y`	логическое «и»
8	`x || y`	логическое «или»
9	`q ? x : y`	тернарный условный оператор

10	`x = y`
	`x *= y`
	`x /= y`
	`x %= y`
	`x += y`
	`x -= y`
		присваивание
		умножение с присваиванием
		деление с присваиванием
		остаток от деления с присваиванием
		сложение с присваиванием
		вычитание с присваиванием

11	`x , y`	операция «запятая» (перечисление)

1.2 Операция преобразования типа

Си является статически типизированным языком. Это означает, что каждая переменная в программе имеет определённый тип. Случаются ситуации, в которых может потребоваться изменение типа переменной. Для этого служит приведение типов.

Приведение типов бывает двух видов: явное и неявное. Явным называется такое преобразование типа, которое программист явно записывает в коде. Рассмотрим пример с делением двух целых чисел:

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

int main()
{
    int a = 4;
    int b = 3;
    cout << a/b << endl;
    return 0;
}
```

Если скомпилировать и выполнить данную программу, то в качестве результата в консоль будет выведено число 1. Это происходит потому, что обе переменных в операции деления имеют тип **int**, поэтому и результат имеет тип int, целое число. Для того, чтобы в результате деления получить ожидаемое значение 1.33333..., необходимо привести

переменные к типу **double** или **float**. Для явного приведения типа переменных к типу **double** необходимо указать желаемый тип в скобках перед переменной:

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

int main()
{
    int a = 4;
    int b = 3;
    cout << (double)a/(double)b << endl;
    cout << double(a+b) / 2 << endl; // другая форма записи операции
    return 0;
}
```

В примере было произведено явное приведение типов переменных к типу **double**.

Вторым видом преобразования типов переменных является неявное приведение типов. В случае, если в одном выражении встречаются переменные разных типов, компилятор пытается привести их к «старшему» типу в выражении. «Младший» тип — **char**, он приводится к **int**, **int** приводится к вещественным типам и так далее. Если бы в примере выше переменная a сразу имела тип **double**, то никаких приведений типов не потребовалось бы.

Благодаря неявному преобразованию типов в примере выше можно явно приводить тип к **double** только у одной переменной (причём любой). Результат будет верным:

```
cout << (double)a/b << endl;
```

2 Директивы препроцессора и их использование

Препроцессор выполняет преобразование программного кода до этапа компиляции. Например, в его список задач входит удаление комментариев из кода, макроподстановки, условная компиляция и включение файлов. Управление работой препроцессора осуществляется с помощью директив препроцессора: специальных команд, начинающихся с решётки. Рассмотрим основные директивы.

Под макроподстановками (макросами) понимается замена одной части строки в коде на другую без каких-либо дополнительных проверок. Например, чтобы использовать число π в коде, создадим макроподстановку **MY_PI**, и везде вместо числа **3.1415** будем писать **MY_PI**, а препроцессор перед компиляцией заменит все **MY_PI** на число **3.1415**. **MY_PI** выбрано произвольно, можно использовать другое имя. Для того, чтобы заставить препроцессор заменить **MY_PI** на **3.1415** в коде программы в начале файла следует написать:

```
#define MY_PI 3.1415
```

Общий синтаксис этого оператора выглядит так:

```
#define что_заменить на_что_заменить
```

Здесь «**что_заменить**» должно подчиняться обычным правилам именования переменных – комбинация латинских букв, цифр и символа подчеркивания, без пробелов, начинающаяся с буквы.

Пример:

```
#include <iostream>
#define MY_PI 3.1415
using namespace std;

int main()
{
    cout << MY_PI << endl;
    return 0;
}
```

Макроподстановка может иметь один и более параметров. Для этого параметры перечисляются в скобках через запятую после объявления ее имени. В примере ниже объявлен макрос **MACROS_WITH_PARAMETERS** с параметрами **par1** и **par2**.

Пример:

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

#define MACROS_WITH_PARAMETERS(par1, par2) cout << par1/par2 << endl;

int main()
{
    MACROS_WITH_PARAMETERS(2.0,3.0)
    return 0;
}
```

Включение файлов производится с помощью директивы **#include**. Так,

```
#include <math.h>
```

вставляет все содержимое файла **math.h** в то место, где была эта строчка.

Условная компиляция позволяет позволяет «включать» и «выключать» фрагменты кода в зависимости от условий до начала компиляции. Для этого служат директивы **#if**, **#ifdef**, **#elif**, **#else**, **#endif**. Например, в последний пример можно добавить проверку наличия определенного макроса **MY_PI**. Если такого макроса не определено, то придется вывести сообщение в консоль.

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

#define MY_PI 3.1415

int main(void) {
#ifdef MY_PI
    cout << MY_PI << endl;
#endif

#ifndef MY_PI
    cout << "no MY_PI" << endl;
#endif
    return 0;
}
```

Если убрать строку **#define MY_PI 3.1415**, в консоль вместо **3.1415** будет выведено «**no MY_PI**».

3 Массивы

3.1 Одномерные массивы

Массив — это последовательный блок памяти, в котором может храниться одно и более значений одного и того же типа. Например, для хранения пяти целых чисел можно использовать пять переменных с различными именами:

```
int a = 6, b = 2, c = 3, d = 5, e = 0, f = 10;
```

а можно использовать один массив:

```
int arr[6];
arr[0] = 6;
arr[1] = 2;
arr[2] = 3;
arr[3] = 5;
arr[4] = 0;
arr[5] = 10;
```

Другой способ заполнения массива из примера выше:

```
int arr[] = {6, 2, 3, 5, 0, 10};
```

В случае, когда длина массива достаточно мала и заранее известна на момент компиляции программы, можно использовать использовать массивы со статическим выделением памяти (как в примере выше, под массив **`arr`** выделен блок памяти под 6 значений типа целое число).

В массиве значения хранятся последовательно, одно за другим. Обращение к элементам массива осуществляется посредством индекса в квадратных скобках:

```
cout << arr[2] << endl;
```

3.2 Многомерные массивы

Многомерные массивы — это массивы массивов более низкой размерности. Например, двумерный массив — это массив одномерных массивов. В случае статического выделения памяти пример работы с двумерным массивом:

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

int main()
{
    double a[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };

    a[2][3] = 3.1415;

    cout << a[2][3] << endl;

    return 0;
}
```

В примере выше создаётся массив массивов **`a`**, каждый элемент **`a[...]`** является массивом. Поэтому запись **`a[2][3]`** можно читать как «третий элемент (счёт начинается с нуля) второго массива **`a`**».

Аналогичным образом можно создавать массивы и большей размерности.

4 Операторы sizeof() и typedef

4.1 Оператор sizeof()

Оператор **`sizeof()`** позволяет определить размер любого типа данных в байтах. Он может быть применён к любому типу данных, будь то структура или класс. Этот оператор важен для определения точного размера переменных в данной операционной системе. Выведем размеры основных типов:

```
    cout << sizeof(char) << endl;
    cout << sizeof(int) << endl;
    cout << sizeof(float) << endl;
    cout << sizeof(double) << endl;
```

Результат:

```
1
4
4
8
```

4.2 Ключевое слово typedef

Ключевое слово **`typedef`** позволяет определить собственное имя-синоним для типа данных. Его имеет смысл использовать, если имя типа оказывается длинным или сложным.

Например, можно написать:

```
typedef long int mytype;
```

и далее объявлять переменные вида:

```
mytype a;
mytype b;
```

В этом примере тип **`mytype`** является синонимом для типа **`long int`**.

5 Полезные функции стандартной библиотеки языка Си

Для использования математических функций стандартной библиотеки Си в начале файла с исходным кодом необходимо подключить заголовочный файл:

```
#include <math.h>
```

Список основных математических функций:

- **`cos(x)`** косинус;
- **`sin(x)`** синус;
- **`tan(x)`** тангенс;
- **`asin(x)`** арксинус;
- **`atan(x)`** арктангенс;
- **`exp(x)`** экспонента;
- **`log(x)`** натуральный логарифм;
- **`log10(x)`** десятичный логарифм;
- **`pow(x,y)`** возведение в степень (x^y);
- **`sqrt(x)`** квадратный корень;

- **ceil(x)** округление вверх;
- **floor(x)** округление вниз;
- **round(x)** округление к ближайшему;
- **fabs(x)** модуль вещественного числа.

Пример.

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

int main()
{
    cout << "cos(3.14)=" << cos(3.14) << endl;
    cout << "sin(3.14)=" << sin(3.14) << endl;
    cout << "tan(3.14)=" << tan(3.14) << endl;
    cout << "asin(1)=" << asin(1) << endl;
    cout << "atan(1)=" << atan(1) << endl;
    cout << "exp(2)=" << exp(2) << endl;
    cout << "log(2.71)=" << log(2.71) << endl;
    cout << "log10(100)=" << log10(100) << endl;
    cout << "pow(3,2)=" << pow(3,2) << endl;
    cout << "sqrt(9)=" << sqrt(9) << endl;
    cout << "ceil(1.3)=" << ceil(1.3) << endl;
    cout << "floor(1.3)=" << floor(1.3) << endl;
    cout << "round(1.3)=" << round(1.3) << endl;
    cout << "fabs(-3)=" << fabs(-3) << endl;
    return 0;
}
```

Результат работы программы:

```
cos(3.14)=-0.999999
sin(3.14)=0.00159265
tan(3.14)=-0.00159265
asin(1)=1.5708
atan(1)=0.785398
exp(2)=7.38906
log(2.71)=0.996949
log10(100)=2
pow(3,2)=9
sqrt(9)=3
ceil(1.3)=2
floor(1.3)=1
round(1.3)=1
fabs(-3)=3
```

При подключении другого стандартного заголовочного файла:

```
#include <stdlib.h>
```

можно получить доступ к нескольким другим полезным функциям стандартной библиотеки языка Си:

- **atoi(s)** преобразование строки в целое число (ноль по умолчанию);
- **atof(s)** преобразование строки в вещественное число (ноль по умолчанию);
- **max(a,b)** **возвращает наибольшее из** двух чисел;
- **min(a,b)** **возвращает наименьшее из** двух чисел;

- **abs(a)** возвращает абсолютное значение (модуль) целого числа;
- **rand()** возвращает очередное псевдослучайное целое число в диапазоне от нуля до предопределенной константы **RAND_MAX**;
- **srand(a)** инициализирует генератор псевдослучайных чисел заданным числом, чтобы сгенерированные последовательности отличались одна от другой;
- **system**("pause") таким образом можно вызвать любую системную команду, в данном случае – ожидание нажатия на любую клавишу;
- **exit(0)** завершение программы (параметр – код возврата).

Наконец, вот такой способ поможет использовать русский язык при выводе в текстовом режиме (увы, вводить русские слова таким способом нельзя):

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

void main()
{
	setlocale(LC_ALL,"rus");
	cout << "Так можно напечатать приглашение по-русски" << endl << flush;
}
```

6 Методы сортировки и поиска

Сортировка — преобразование массива неупорядоченных элементов в массив упорядоченных по какому-либо критерию элементов. В простейшем случае элементами могут выступать числа, а сортировка производится для расположения чисел по возрастанию.

6.1 «Пузырьковая» сортировка

Одна из простейших для понимания сортировок. Получила свое название из-за того, что на каждой итерации наибольший элемент оказывается на своей позиции в массиве, «всплывая», как пузырек.

1. Последовательно для пары 0 и 1 элемента, для 1 и 2 элемента и т. д. до N-2 и N-1 элементов массива производится сравнение. Если в паре элементы расположены не по возрастанию, то производится обмен значений элементов.
2. Повторяется первый шаг, каждый раз уменьшается количество рассматриваемых элементов массива на 1 (т. к. крайний правый элемент уже оказался на своей позиции).
3. Алгоритм завершается, если при проходе не было произведено ни одного обмена значений элементов.

6.2 Сортировка выбором

На каждой итерации выбирается наименьший элемент массива.

1. Производится поиск наименьшего элемента массива. Как только этот элемент найден, производится обмен этого значения с первым неотсортированным элементом.
2. Повторяется первый шаг для оставшихся элементов из конца массива.
3. Алгоритм завершается, как только остаётся только один элемент.

6.3 Сортировка вставкой

Массив делится на две части: уже отсортированную (вначале в ней только один элемент) и еще не отсортированную.

На каждой итерации очередной элемент неотсортированной части массива вставляется на нужную позицию в отсортированной части.

1. Взять первый элемент из неотсортированной части массива и найти место для его вставки в отсортированной части (последовательным перебором или методом дихотомии).
2. Начиная с найденной позиции весь хвост отсортированной части массива сдвинуть на одну позицию вправо и сохранить выбранный элемент в найденную позицию.
3. Повторить первый шаг для оставшихся элементов входного массива.
4. Алгоритм завершается, когда в неотсортированной части не останется ни одного элемента.

6.4 Алгоритм поиска строки в тексте

Алгоритм проверяет на совпадение с искомой строкой фрагменты текста со всеми возможными смещениями.

1. Сравнить первый символ искомой строки с символом в тексте с текущим смещением относительно начала текста.
2. Если символы совпадают, то проверить следующие по порядку символы. Если совпадения нет – увеличить смещение строки в тексте на 1 и перейти на шаг 1.
3. Если все символы искомой строки совпадают с соответствующими символами в тексте, то поиск завершен.

6.5 Алгоритм Рабина-Карпа для поиска строки в тексте

Использует контрольную сумму при поиске строки в тексте.

1. Вычислить контрольную сумму всех символов искомой строки.
2. Вычислить контрольную сумму символов подстроки текста начиная с заданного смещения и с длиной, равной длине искомой строки.
3. Если вычисленные суммы совпадают, то перейти к посимвольному сравнению строки и участка текста. Если суммы не совпали или посимвольное сравнение неуспешно – перейти к следующему шагу, иначе – поиск завершен.
4. Из контрольной суммы подстроки текста вычесть значение первого символа, увеличить смещение подстроки в строке на 1, для этого к контрольной сумме подстроки текста прибавить последний символ подстроки после ее смещения.
5. Перейти к шагу 3.

Типовое задание на сортировку: написать и протестировать программу сортировки массива. Массив объявляется переменной длины, его длина объявляется через `#define`. Массив заполняется случайными числами при помощи функции `rand()`, после чего печатается на экран. Затем массив сортируется с подсчетом количества операций сравнения и перестановок элементов (отдельными переменными-счетчиками) и печатается на экран.

Типовое задание на поиск элемента: у пользователя запрашивается искомая подстрока, затем запрошенное ищется с подсчетом количества операций сравнения элементов и печатается позиция, в которой подстрока нашлась, а также количество операций сравнения элементов, которые были выполнены.

1. Вариант

Реализовать приближение линейной функции **методом наименьших квадратов**.
Значения хранятся в массивах **x** и **y**, размер массивов задаётся с помощью **#define**. Пользователь вводит значения из консоли. Вывести параметры прямой, являющейся наилучшим приближением к заданным точкам на плоскости.

2. Вариант

Сортировка массива действительных чисел **методом вставки** в порядке возрастания.
Массив заполняется случайными значениями с помощью функции **rand()**. Размер массива задаётся с помощью **#define**. Необходимо вывести количество перестановок и сравнений, а также входной и выходной массивы.

3. Вариант

Сортировка массива действительных чисел **методом вставки** в порядке убывания.
Массив заполняется случайными значениями с помощью функции **rand()**. Размер массива задаётся с помощью **#define**. Необходимо вывести количество перестановок и сравнений, а также входной и выходной массивы.

4. Вариант

Сортировка массива действительных чисел **методом выбора** в порядке возрастания.
Массив заполняется случайными значениями с помощью функции **rand()**. Размер массива задаётся с помощью **#define**. Необходимо вывести количество перестановок и сравнений, а также входной и выходной массивы.

5. Вариант

Вычислить **сумму положительных и сумму отрицательных элементов** массива.
Массив заполняется случайными значениями с помощью функции **rand()**. Размер массива задаётся с помощью **#define**.

6. Вариант

Сортировка массива действительных чисел **методом выбора** в порядке убывания.
Массив заполняется случайными значениями с помощью функции **rand()**. Размер массива задаётся с помощью **#define**. Необходимо вывести количество перестановок и сравнений, а также входной и выходной массивы.

7. Вариант

Сортировка массива действительных чисел **методом пузырька** в порядке возрастания.
Массив заполняется случайными значениями с помощью функции **rand()**. Размер массива задаётся с помощью **#define**. Необходимо вывести количество перестановок и сравнений, а также входной и выходной массивы.

8. Вариант

Сортировка массива действительных чисел **методом пузырька** в порядке убывания.
Массив заполняется случайными значениями с помощью функции **rand()**. Размер массива задаётся с помощью **#define**. Необходимо вывести количество перестановок и сравнений, а также входной и выходной массивы.

9. Вариант

Выполнить **транспонирование матрицы**.
Размеры матрицы задаются с помощью **#define**. Матрица заполняется случайными значениями с помощью функции **rand()**.

10. Вариант

Поиск **первого вхождения подмассива** в массиве.
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задается с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождения подмассива, сами массив и подмассив.

11. Вариант

Поиск **последнего вхождения подмассива** в массиве.
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задается с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождения подмассива, сами массив и подмассив.

12. Вариант

Поиск **всех вхождений подмассива** в массиве.
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задается с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождений подмассива, сами массив и подмассив.

13. Вариант

Поиск **первого вхождения подмассива** в массиве с одной ошибкой (то есть один элемент найденного вхождения имеет право отличаться от искомого массива).
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задается с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождения подмассива, сам массив и искомый подмассив.

14. Вариант

Поиск **второго вхождения подмассива** в массиве с одной ошибкой (то есть один элемент найденного вхождения имеет право отличаться от искомого массива).
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задаётся с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождения подмассива, сам массив и искомый подмассив.

15. Вариант

Поиск **последнего вхождения подмассива** в массиве с одной ошибкой (то есть один элемент найденного вхождения имеет право отличаться от искомого массива).
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задается с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождения подмассива, сам массив и искомый подмассив.

16. Вариант

Поиск **всех вхождений подмассива** в массиве с одной ошибкой.
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задаётся с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождения подмассива, сам массив и искомый подмассив.

17. Вариант

Поиск **первого вхождения подмассива** в массиве с заданным пользователем числом ошибок (то есть указанное количество элементов найденного вхождения имеет право отличаться от искомого массива).
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задаётся с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождения подмассива, сам массив и искомый подмассив.

18. Вариант

Поиск **последнего вхождения подмассива** в массиве с заданным пользователем числом ошибок (то есть указанное количество элементов найденного вхождения имеет право отличаться от искомого массива).
Массив заполняется случайными значениями с помощью функции **`rand()`**. Размер массива и подмассива задается с помощью **`#define`**. Необходимо вывести количество сравнений, начало вхождения подмассива, сами массив и подмассив.

19. Вариант

Вывести **строки матрицы** в порядке возрастания суммы их элементов.
Для этого завести отдельный массив сумм элементов по строкам, искать каждый раз минимальный элемент из оставшихся и выводить соответствующую строку матрицы. Матрица заполняется случайными значениями с помощью функции **`rand()`**. Размеры матрицы задаются с помощью **`#define`**.

20. Вариант

Вывести **строки матрицы** в порядке убывания суммы их элементов.
Для этого завести отдельный массив сумм элементов по строкам, искать каждый раз максимальный элемент из оставшихся и выводить соответствующую строку матрицы. Матрица заполняется случайными значениями с помощью функции **`rand()`**. Размеры матрицы задаются с помощью **`#define`**.

21. Вариант

Подсчитать **среднее значение и дисперсию** в каждом столбце матрицы.
Матрица заполняется случайными значениями с помощью функции **`rand()`**. Размеры матрицы задаются с помощью **`#define`**.

22. Вариант

Реализовать **сложение и вычитание** двух чисел в столбик. Числа представлены массивами десятичных цифр и запрашиваются у пользователя.
Выводить слагаемые и результат.

23. Вариант

Реализовать **сложение и вычитание** двух шестнадцатеричных чисел в столбик. Числа представлены массивами шестнадцатеричных цифр и запрашиваются у пользователя.
Выводить слагаемые и результат.

24. Вариант

Построить и вывести **вертикальную гистограмму** (количество элементов каждого значения) значений целочисленного массива. Каждая строка гистограммы содержит столько символов «**`*`**», сколько раз встретилось уникальное значение входного массива, соответствующее данной строке гистограммы. Строки гистограммы должны выводиться от меньшего значения исходного массива к большему.
Размеры исходного массива и количество разных значений в гистограмме задать через **`#define`**.

25. Вариант

Построить **график функции**, фигурировавшей в уравнении, которое решалось после предыдущего семинара в окрестности найденного корня уравнения. График должен изображаться в виде матрицы символов, в которой значения функции помечаются символом «**`*`**», а также помечаются оси координат, прочие символы должны быть пробелами. Необходимо сделать преобразование масштаба, чтобы окрестность корня изображалась максимально информативно.
Размерность матрицы задается с помощью **`#define`** (не более 80 столбцов и 25 строк).