Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА
Институт Информационных Технологий
Кафедра Промышленной Информатики

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

Тема лекции «Конструкция ветвления. Циклы. Массивы»

Лектор Каширская Елизавета Натановна (к.т.н., доцент, ФГБОУ ВО "МИРЭА - Российский технологический университет") e-mail: liza.kashirskaya@gmail.com

Лекция № 3

КОНСТРУКЦИЯ ВЕТВЛЕНИЯ. УСЛОВНЫЙ ОПЕРАТОР IF

Когда программа C++ должна принять решение о том, какое из альтернативных действий следует выполнить, такой выбор обычно реализуется оператором **if**. Этот оператор имеет две формы: просто **if** и **if else**.

Давайте сначала исследуем простой **if**. Он создан по образцу обычного английского языка, как в выражении "If you have a Captain Cookie card, you get a free cookie" (игра слов на основе созвучности фамилии Кук и слова "cookie" (печенье) — *прим. перев.*). Оператор **if** разрешает программе выполнять оператор или блок операторов при условии истинности проверочного условия и пропускает этот оператор или блок, если проверочное условие оценивается как ложное. Таким образом, оператор **if** позволяет компилятору принимать решение относительно того, нужно ли выполнять некоторую часть кода.

Синтаксис оператора if :

if (проверочное-условие)

оператор;

Оператор IF

Условная конструкция в C++ всегда записывается в круглых скобках после оператора **if.**

Внутри фигурных скобок указывается тело условия. Если условие выполнится, то начнется выполнение всех команд, которые находятся между фигурными скобками.

Пример конструкции ветвления

```
#include<iostream>
using namespace std;
int main() {
setlocale(0, "");
double num;
cout<<"Введите произвольное число: ";
cin>>num;
if (num <10) { // Если введенное число меньше 10.
cout<<"Это число меньше 10."<< endl;
}
else{ // иначе
cout<<"Это число больше либо равно 10."<< endl;
}
return 0;
}
```

```
if (num <10) {  // Если введенное число меньше 10.
cout<<"Это число меньше 10."<< endl;
}
else{  // иначе
cout<<"Это число больше либо равно 10."<< endl;
}
```

Здесь говорится: «Если переменная num меньше 10 — вывести соответствующее сообщение. Иначе вывести другое сообщение».

Усовершенствуем программу так, чтобы она выводила сообщение, о том, что переменная num равна десяти:

```
if (num <10) {  // Если введенное число меньше 10.
cout<<"Это число меньше 10."<< endl;
}
elseif (num == 10) {
 cout<<"Это число равно 10."<< endl;
}
else{  // иначе
cout<<"Это число больше 10."<< endl;
}
//Здесь мы проверяем три условия:
//Первое – когда введенное число меньше 10-ти
//Второе – когда число равно 10-ти
//Третье – когда число больше десяти
```

Заметьте, что во втором условии, при проверке равенства, мы используем оператор равенства = = (двойное «равно»), а не оператор присваивания (=), потому что мы не изменяем значение переменной при проверке, а сравниваем ее текущее значение с числом 10.

Если поставить оператор присваивания в условие, то при проверке условия значение переменной изменится, после чего это условие выполнится.

Каждому оператору **if** соответствует только один оператор **else**. Совокупность этих операторов —**if else** означает, что если не выполнилось предыдущее условие, то проверить данное. Если ни одно из условий не верно, то выполняется тело оператора **else.**

Оператор IF

Если после оператора **if, else или их связки else if** должна выполняться только одна команда, то фигурные скобки можно не ставить. Предыдущую программу можно записать следующим образом:

```
#include<iostream>
Using namespace std;
Int main() {
setlocale(0, "");
doublenum;
cout<<"Введите произвольное число: ";
cin>> num;
if (num <10) // Если введенное число меньше 10.
cout<<"Это число меньше 10."<< endl;
elseif (num == 10)
cout<<"Это число равно 10."<< endl;
else// иначе
cout<<"Это число больше 10."<< endl;
return0;
}
```

Оператор IF

Такой метод записи выглядит более компактно. Если при выполнении условия нам требуется выполнить более одной команды, то фигурные скобки необходимы. *Например*:

```
#include<iostream>
Usingnamespacestd;
Intmain() {
setlocale(0, "");
double num;
intk;
cout<<"Введите произвольное число: ";
cin>> num;
if (num <10) {  // Если введенное число меньше 10.
cout<<"Это число меньше 10."<< endl;
k = 1;
}
elseif (num == 10) { cout<<"Это число равно 10."<< endl;
k = 2;
}
else{  // иначе
cout<<"Это число больше 10."<< endl;
k = 3;
}
cout<<"k = "<< k << endl;
return0;
}
```

Оператор IF

Данная программа проверяет значение переменной **num**. Если она меньше 10, то присваивает переменной k значение единицы. Если переменная **num** равна десяти, то присваивает переменной k значение двойки. В противном случае — значение тройки. После выполнения ветвления, значение переменной k выводится на экран.

Оператор Switch (переключатель)

Предположим, что вы создаете экранное меню, которое предлагает пользователю на выбор один из нескольких возможных вариантов, например, «дешевый», «умеренный», «дорогой», «экстравагантный» и «непомерный». Вы можете расширить последовательность **if else if else** для обработки этих пяти альтернатив, но оператор C++ **switch** упрощает обработку выбора из большого списка. Ниже представлена общая форма оператора **switch**:

```
switch (целочисленное-выражение)
{
case метка 1 : оператор(ы)
case метка 2 : оператор(ы)
…
default : оператор (ы)
}
```

В данном случае слово **case** означает не «*ящик*», как наивно полагают некоторые, а «*в случае*».

Оператор Switch

Оператор **switch** действует подобно маршрутизатору, который сообщает компилятору, какую строку кода выполнять следующей. По достижении оператора **switch** программа переходит к строке, которая помечена значением, соответствующим текущему значению *целочисленное-выражение*. Например, если целочисленное-выражение имеет значение 4, то программа переходит к строке с меткой **case 4:**

Как следует из названия, выражение *целочисленное-выражение* должно быть целочисленным. Также каждая метка должна быть целым константным выражением. Чаще всего метки бывают константами типа char или int, такими как 1 или 'q', либо же перечислителями. Если *целочисленное-выражение* не соответствует ни одной метке, программа переходит к метке default. Метка default не обязательна. Если она опущена, а соответствия не найдено, программа переходит к оператору, следующему за switch (рис. 4.1.).

Оператор Switch

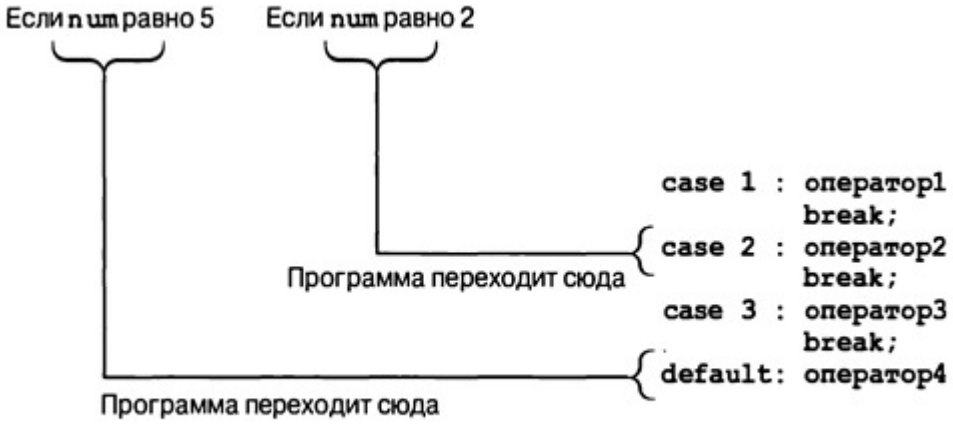


Рисунок 4.1. Структура оператора switch

Оператор switch в C++ отличается от аналогичных операторов в других языках, например, Pascal, в одном очень важном отношении. Каждая метка case в C++ **работает только как метка строки, а не граница между выборами.**

То есть после того, как программа перейдет на определенную строку в **switch**, она последовательно выполнит все операторы, следующие за этой строкой внутри **switch**, если только вы явно не направите ее в другое место. Выполнение не останавливается автоматически на следующем **case**. Чтобы прекратить выполнение в конце определенной группы операторов, вы должны использовать оператор **break**. Это передаст управление за пределы блока **switch**.

ЦИКЛЫ В С++. Цикл for

Обстоятельства часто требуют от программ выполнения повторяющихся задач, таких как сложение элементов массивов один за другим или 20-кратная распечатка похвалы за продуктивность. Цикл **for** облегчает выполнение задач подобного рода.

Если мы знаем точное количество действий (итераций) цикла, то можем использовать цикл for. Синтаксис его выглядит примерно так:

for (действие до начала цикла; условие продолжения цикла; действия в конце каждой итерации цикла)

{инструкция цикла 1; инструкция цикла 2; … инструкция цикла N; }

Итерацией цикла называется один проход этого цикла.

Обычно оператор выглядит так:

for (счетчик = значение; счетчик < значение; шаг цикла) {тело цикла }

Счетчик цикла — это переменная, в которой хранится количество проходов данного цикла. Описание синтаксиса следующее.

1. Сначала присваивается первоначальное значение счетчику, после чего ставится точка с запятой.

2. Затем задается конечное значение счетчика цикла. После того, как значение счетчика достигнет указанного предела, цикл завершится. Снова ставим точку с запятой.

3. Задаем шаг цикла. Шаг цикла — это значение, на которое будет увеличиваться или уменьшаться счетчик цикла при каждом проходе.

Цикл for

Пример кода

Напишем программу, которая будет считать сумму всех чисел от 1 до 1000.

```
#include <iostream>
Using name space std;
Int main() {
int i; // счетчикцикла
int sum = 0; // суммачиселот 1 до 1000.
set locale(0, "");
for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и задаем шаг цикла - 1.
{
sum = sum + i;
}
cout<<"Сумма чисел от 1 до 1000 = "<<sum<<endl;
return 0;
}
```

Если мы скомпилируем этот код и запустим программу, то она покажет нам ответ: 500500. Это и есть сумма всех целых чисел от 1 до 1000. Если считать это вручную, понадобится очень много времени и сил. Цикл выполнил всю рутинную работу за нас.

Заметьте, что конечное значение счетчика я задала нестрогим неравенством (<= — меньше либо равно), поскольку, если бы я поставила знак меньше, то цикл произвел бы 999 итераций, т.е. на одну меньше, чем требуется. Это довольно важный момент, т.к. здесь новички часто допускают ошибки, особенно при работе с массивами. Значение шага цикла я задала равное единице. i++ — это тоже самое, что и i = i + 1.

В теле цикла, при каждом проходе программа увеличивает значение переменной sum на i. Еще один очень важный момент — в начале программы я присвоила переменной sum значение нуля. Если бы я этого не сделала, программа вылетела вы в **сегфолт**. При объявлении переменной без ее инициализации эта переменная будет хранить «мусор».

Естественно, к мусору мы ничего прибавить не можем. Некоторые компиляторы инициализирует переменную нулем при ее объявлении.

Сегфолт

Ошибка сегментации (англ. *Segmentation fault*, сокр. *segfault*, жарг. *сегфолт*) — ошибка программного обеспечения, возникающая при попытке обращения к недоступным для записи участкам памяти, либо при попытке изменить память запрещённым способом.
Условия, при которых происходят нарушения сегментации, и способы их проявления зависят от операционной системы.
Ещё один способ вызвать ошибку сегментации заключается в том, чтобы вызвать функцию main() рекурсивно, что приведёт к переполнению стека:

```
int main()
{
   main();
 }
```

Ошибка сегментации возникнет, если при использовании массивов случайно указать в качестве размера массива неинициализированную переменную:

```
int main()
{
const int nmax = 10;
int i, n, a[n];
}
```

Операции инкремента и декремента

Язык C++ снабжен несколькими операциями, которые часто используются в циклах; давайте потратим немного времени на их изучение. Вы уже видели одну из них - операцию инкремента (++), которая получила отражение в самом названии C++. Есть также операция декремента (--). Эти операции выполняют два чрезвычайно часто встречающихся действия в циклах: увеличивают и уменьшают на единицу значение счетчика цикла. Однако к тому, что вы уже знаете о них, есть что добавить. Каждая из них имеет два варианта. Префиксная версия операции указывается перед операндом, как в ++x. Постфиксная версия следует после операнда, как в x++. Эти две версии имеют один и тот же эффект для операнда, но отличаются в контексте применения. Все похоже на получение оплаты за стрижку газона авансом или после завершения работы: оба метода имеют один и тот же конечный результат для вашего бумажника, но отличаются тем, в какой момент деньги в него добавляются. В листинге ниже демонстрируется разница на примере операции инкремента.

Операции инкремента и декремента

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

int main()
{

int a = 20;
int b = 20;
cout << "a = " << a << ": b = " << b << "\n";
cout << "a++ = " << a++ << " : ++b = " << ++b << "\n";
cout << "a = " << a << ": b = " << b << "\n";
return 0;
}
```

Результат выполнения этой программы показан ниже:

```
a = 20: b = 20
a++ = 20 : ++b = 21
a = 21: b = 21
```

Операции инкремента и декремента

Грубо говоря, нотация ***a***++ означает "использовать текущее значение *a* при вычислении выражения, затем увеличить *a* на единицу". Аналогично нотация ++***a*** означает "сначала увеличить значение *a* на единицу, затем использовать новое значение при вычислении выражения". Например, мы имеем следующие отношения:

```
int x = 5;
int y = ++x; // изменить x, затем присвоить его y
                    // y равно 6, x равно 6
int z = 5;
int y = z++; // присвоить y, затем изменить z
                    //y равно 5, z равно 6
```

Операции инкремента и декремента

Операции инкремента и декремента представляют собой простой удобный способ решения часто возникающей задачи увеличения или уменьшения значений на единицу. Операции инкремента и декремента — симпатичные и компактные, но не стоит поддаваться соблазну и применять их к одному и тому же значению более одного раза в одном и том же операторе. Проблема в том, что при этом правила "использовать и изменить" и "изменить и использовать" становятся неоднозначными. То есть, следующий оператор в различных системах может дать совершенно разные результаты:

```
x = 2 * x++ * (3 - ++x) ; //не поступайте так
```

В C++ поведение операторов подобного рода не определено.

Цикл While

Когда мы не знаем, сколько итераций должен произвести цикл, нам нужны циклы while или do...while. Синтаксис цикла while в C++ выглядит следующим образом.

while (Условие) {Тело цикла;}

Данный цикл будет выполняться, пока условие, указанное в круглых скобках, является истиной. Решим задачу о сумме рядя с помощью цикла while. Хотя здесь мы точно знаем, сколько итераций должен выполнить цикл, очень часто бывают ситуации, когда это значение неизвестно.

Ниже приведен исходный код программы, считающей сумму всех целых чисел от 1 до 1000.

```
#include <iostream>
Using name space std;
int main() {
set locale(0, "");
int i = 0; // инициализируем счетчи кцикла.
int sum = 0; // инициализируем счетчик суммы.
while (i <1000)
    {
i++;
sum += i;
}
cout<<"Сумма чисел от 1 до 1000 = "<<sum<<endl;
return 0;
}
```

После компиляции программа выдаст результат, аналогичный результату работы предыдущей программы. Но поясним несколько важных моментов. Я задала строгое неравенство в условии цикла и инициализировала счетчик **i** нулем, так как в цикле while происходит на одну итерацию больше, поэтому он будет выполняться до тех пор, пока значение счетчика не перестает удовлетворять условию, но данная итерация все равно выполнится. Если бы мы поставили нестрогое неравенство, то цикл бы закончился, когда переменная i стала бы равна 1001 и выполнилось бы на одну итерацию больше.

Теперь давайте рассмотрим по порядку исходный код нашей программы. Сначала мы инициализируем счетчик цикла и переменную, хранящую сумму чисел.

В данном случае мы обязательно должны присвоить счетчику цикла какое-либо значение. В предыдущей программе мы это значение присваивали внутри цикла for, здесь же, если мы не инициализируем счетчик цикла, то в него попадет «мусор» и компилятор, в лучшем случае, выдаст нам ошибку, а в худшем, если программа соберется — дефолт практически неизбежен. *Дефолт буквально – невыполнение договора.*

Затем мы описываем условие цикла — «пока переменная i меньше 1000 — выполняй цикл». При каждой итерации цикла значение переменной-счетчика i увеличивается на единицу внутри цикла.

Когда выполнится 1000 итераций цикла, счетчик станет равным 999 и следующая итерация уже не выполнится, поскольку 1000 не меньше 1000. Выражение sum += i является укороченной записью sum = sum + i.

После окончания выполнения цикла выводим сообщение с ответом.

Теперь мы с поговорим о массивах. Вы уже знаете, что переменная — это ячейка в памяти компьютера, где может храниться одно единственное значение.

Массив — это область памяти, где могут последовательно храниться несколько значений.

Цикл do while

Цикл **do while** очень похож на цикл **while**. Единственное их различие в том, что при выполнении цикла do while один проход цикла будет выполнен независимо от условия. Приведу решение задачи на поиск суммы чисел от 1 до 1000, с применением цикла do while. Принципиального отличия нет, но если присвоить переменной i значение, большее, чем 1000, то цикл все равно выполнит хотя бы один проход.

```
#include <iostream>
using namespace std;
Int main() {
setlocale(0, "");
int i = 0; // инициализируем счетчик цикла.
int sum = 0; // инициализируем счетчик суммы.
do {// выполняемцикл.
i++;
sum += i;
}
while (i <1000); // пока выполняется условие.
cout<<"Сумма чисел от 1 до 1000 = "<< sum << endl;
return 0;
}
```

Элементы массива

Каждая из переменных в массиве называется **элементом**. Элементы не имеют своих собственных уникальных имен. Вместо этого для доступа к ним используется имя массива вместе с **оператором индекса** [] и параметром, который называется **индексом**, и который сообщает компилятору, какой элемент мы хотим выбрать. Этот процесс называется **индексированием массива**.

Важно: в отличие от повседневной жизни, отсчет в программировании в C++ всегда начинается с 0, а не с 1. Для массива длиной N элементы массива будут пронумерованы от 0 до N-1. Это называется диапазоном массива.

Рассмотрим случай, когда нужно записать результаты тестов 30 студентов в классе. Без использования массива нам придется выделить почти 30 одинаковых переменных!

```
// Выделяем 30 целочисленных переменных (каждая с разным именем)
int testResultStudent1;
int testResultStudent2;
int testResultStudent3;
// ...
int testResultStudent30;
```

С использованием массива всё гораздо проще. Следующая строчка эквивалентна коду выше:

```
int testResult[30]; // выделяем 30 целочисленных переменных, используя фиксированный массив
```

Объявление переменной массива

В объявлении переменной массива мы используем квадратные скобки [], чтобы сообщить компилятору, что это переменная массива (а не обычная переменная), а в скобках — количество выделяемых элементов (это называется **длиной** или **размером массива**).

В примере, приведенном выше, мы объявили **фиксированный массив** с именем testResult и длиной 30. Фиксированный массив (или «**массив фиксированной длины**») представляет собой массив, размер которого известен во время компиляции. При создании testResult, компилятор выделит 30 целочисленных переменных.

В вышеприведенном примере первым элементом в нашем массиве является testResult[0], второй — testResult[1], десятый — testResult[9], последний — testResult[29]. Хорошо, что уже не нужно отслеживать и помнить кучу разных (хоть и похожих) имен переменных — для доступа к разным элементам нужно изменить только индекс.

Пример программы с использованием массива

Пример программы с использованием массива. Здесь мы можем наблюдать как определение, так и индексирование массива.

```
#include <iostream>

int main()
{
   int array[5]; // массив из пяти чисел
   array[0] = 3; // индекс первого элемента - 0 (нулевой элемент)
   array[1] = 2;
   array[2] = 4;
   array[3] = 8;
   array[4] = 12; // индекс последнего элемента - 4

   std::cout << "The lowest number is " << array[0] << "\n";
   std::cout << "The sum of the first 5 numbers is " << array[0] + array[1] + array[2] + array[3] + array[4] << "\n";

   return 0;
}
```

Результат выполнения программы:

The lowest number is 3

The sum of the first 5 numbers is 29

Массив может быть любого типа данных.

Например, объявляем массив типа double:

```
#include <iostream>

int main()
{
   double array[3]; // выделяем 3 переменные типа double
   array[0] = 3.5;
   array[1] = 2.4;
   array[2] = 3.4;

   std::cout << "The average is " << (array[0] + array[1] + array[2]) / 3 << "\n";

   return 0;
}
```

Результат выполнения программы:

```
The average is 3.1
```

Возьмем группу студентов из десяти человек. У каждого из них есть фамилия. Создавать отдельную переменную для каждого студента — не рационально. Создадим массив, в котором будут храниться фамилии всех студентов.

В языке C++ индексы массивов всегда должны быть интервального типа данных (т.е. типа char, short, int, long, bool и т.д.). Эти индексы могут быть либо константными значениями, либо неконстантными значениями.

***Пример* инициализации массива**

```
string students[10] = {
   "Иванов", "Петров", "Сидоров",
   "Ахмедов", "Ерошкин", "Выхин",
   "Андеев", "Вин Дизель", "Картошкин", "Чубайс"
};
```

Массив создается почти так же, как и обычная переменная. Для хранения десяти фамилий нам нужен массив, состоящий из 10 элементов. Количество элементов массива задается при его объявлении и заключается в квадратные скобки.

Чтобы описать элементы массива сразу при его создании, можно использовать фигурные скобки. В фигурных скобках значения элементов массива перечисляются через запятую. В конце закрывающей фигурной скобки ставится точка с запятой.

Попробуем вывести наш массив на экран с помощью **cout**.

```
#include <iostream>
#include <string>

int main()
{
    std::string students[10] =
        {“Иванов”, ”Петров”, ”Сидоров”, “Ахмедов”,
         ”Ерошкин”, “Выхин”, ”Андреев”,
         ”Вин Дизель”, ”Картошкин”, ”Чубайс”
        }
    std::cout<<students<<std::endl; // Пытаемся вывести весь
                массив непосредственно
    return 0;
}
```

Скомпилируйте этот код и посмотрите, на результат работы программы. Готово? А теперь запустите программу еще раз и сравните с предыдущим результатом. В моей операционной системе вывод был следующим:

- первый вывод: 0x7ffff8b85820
- второй вывод: 0x7fff7a335f90
- третий вывод: 0x7ffff847eb40

Мы видим, что выводится адрес этого массива в оперативной памяти, а никакие не «Иванов» и «Петров».

Дело в том, что при создании переменной компилятор выделяет ей определенное место в памяти. Если мы объявляем переменную типа int, то на машинном уровне она описывается двумя параметрами — ее адресом и размером хранимых данных.

Скомпилируйте этот код и посмотрите, на результат работы Массивы в памяти хранятся таким же образом. Массив типа int из 10 элементов описывается с помощью адреса его первого элемента и количества байт, которое может вместить этот массив. Если для хранения одного целого числа выделяется 4 байта, то для массива из десяти целых чисел будет выделено 40 байт.

Так почему же, при повторном запуске программы, адреса различаются? Это сделано для защиты от атак переполнения буфера. Такая технология называется рандомизацией адресного пространства и реализована в большинстве популярных операционных систем.

Попробуем вывести первый элемент массива – фамилию студента Иванова.

```
#include <iostream>
#include <string>

int main()
{
   std::string students[10] =
       {“Иванов”, ”Петров”, ”Сидоров”, “Ахмедов”,
         ”Ерошкин”, “Выхин”, ”Андреев”,
         ”Вин Дизель”, ”Картошкин”, ”Чубайс”
       }
   std::cout<<students[0]<<std::endl;

 return 0;
}
```

Смотрим, компилируем, запускаем. Убедились, что вывелся именно «Иванов». Заметьте, что нумерация элементов массива в C++ начинается с нуля. Следовательно, фамилия первого студента находится в students[0], а фамилия последнего — в students[9].

Во многих языках программирования нумерация элементов массива также начинается с нуля.

Попробуем вывести список всех студентов. Но сначала подумаем, а что если бы вместо группы из десяти студентов, была бы кафедра их ста, факультет из тысячи, или даже весь университет? Ну не будем же мы писать десятки тысяч строк с cout?

Конечно же нет! Мы будем использовать циклы.

Вывод элементов массива через цикл

```
#include <iostream>
#include <string>

int main()
{
    std::string students[10] =
        {“Иванов”, ”Петров”, ”Сидоров”, “Ахмедов”,
          ”Ерошкин”, “Выхин”, ”Андреев”,
          ”Вин Дизель”, ”Картошкин”, ”Чубайс”
        }
        for (int i=0;i<10;i++)
            {
                std::cout<<students[0]<<std::endl;
            }

    return 0;
}
```

Вывод элементов массива через цикл

Если бы нам пришлось выводить массив из нескольких тысяч фамилий, то мы бы просто увеличили конечное значение счетчика цикла — строку for (...; i < 10; ...) заменили на for (...; i < 10000; ...).

Заметьте, что счетчик нашего цикла начинается с нуля, а заканчивается девяткой. Если вместо оператора строгого неравенства — i < 10 использовать оператор «меньше, либо равно» — i <= 10, то на последней итерации программа обратится к несуществующему элементу массива — students[10]. Это может привести к ошибкам сегментации и аварийному завершению программы. Будьте внимательны — подобные ошибки бывает сложно отловить.

Массив, как и любую переменную можно не заполнять значениями при объявлении.

Объявление массива без инициализации

```
string students[10];
// или
string teachers[5];
```

Элементы такого массива обычно содержат в себе «мусор» из выделенной, но еще не инициализированной, памяти. Некоторые компиляторы заполняют все элементы массива нулями при его создании.

При создании статического массива, для указания его размера может использоваться только константа. Размер выделяемой памяти определяется на этапе компиляции и не может изменяться в процессе выполнения.

```
int n;
cin>>n;
String students[n]; /*Неверно*/
```

Выделение памяти в процессе выполнения возможно при работе с динамическими массивами. Но о них немного позже.

Заполнение массива с клавиатуры

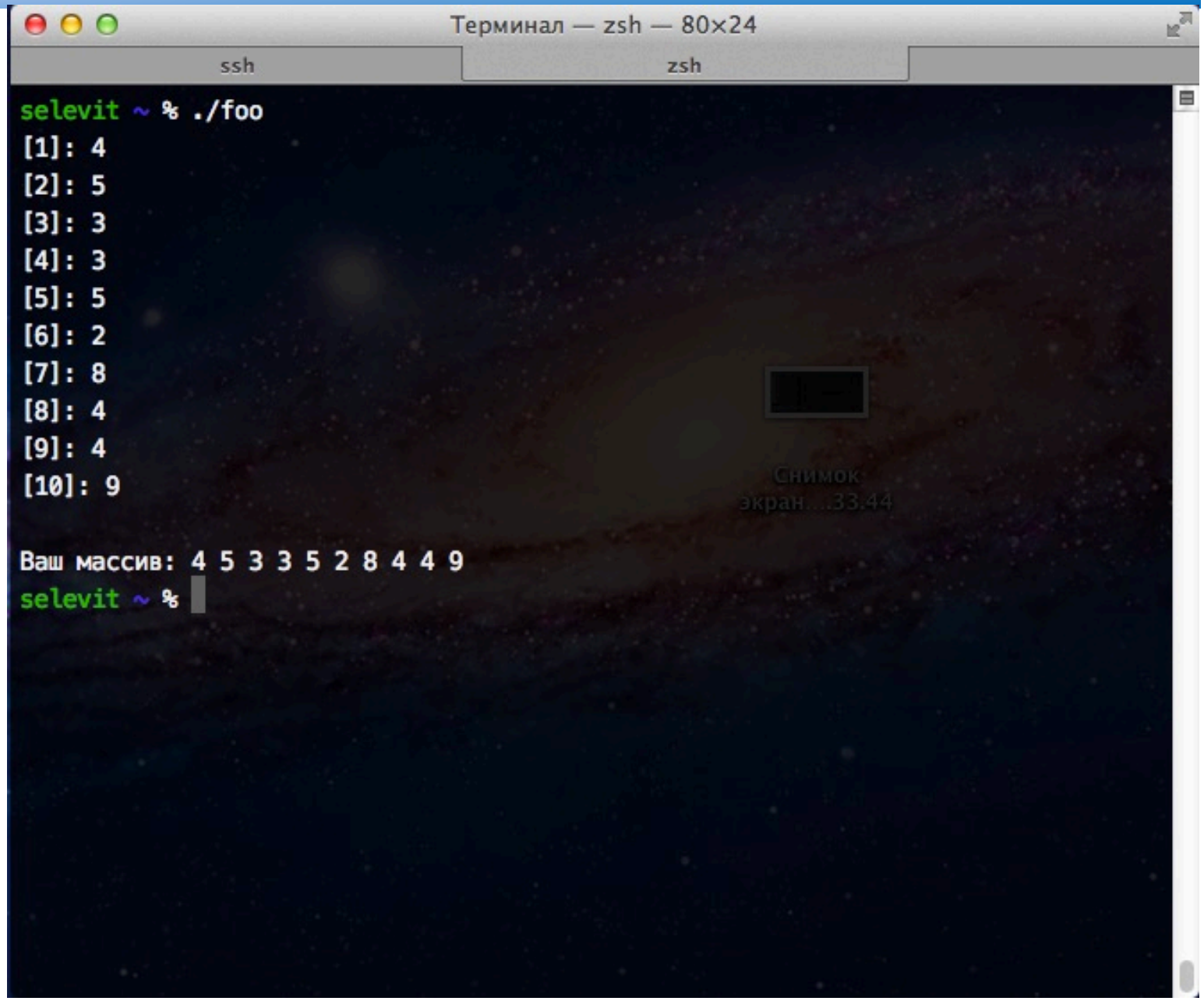
Заполним с клавиатуры пустой массив из 10 элементов.

```
#include <iostream>
#include <string>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int arr[10];

    // Заполняем массив с клавиатуры
     for (int i-0;i<10;i++)
          {
              cout<<“[“<<i+1<<‘[‘<<‘:’;
               cin>>arr[i];
           }
```

```
// И выводим заполненный массив
    cout<<“\nВаш массив:”;

for (int i=0;i<10;++i)
  {
     cout<<arr[i]<<“ “;
  }

 cout<<endl;

return 0;
}
```

Скомпилируем эту программу и проверим ее работу.

Заполнение массива с клавиатуры



Задача 1. Найдите сумму отрицательных элементов массива.

```
int sum=0;
for (int i=0; i<n; i++)
{
   if (a[i]<0)
   {
      sum+=a[i];
   }
}

if (!sum)
{
   cout<<"no numbers < 0";
}
else
{
  cout<<"sum = "<<sum;
}
```

Задача 2. Найдите произведение элементов массива с нечетными номерами.

```
int p=1;
for (int i=1; i<n; i+=2)
{
   p*=a[i];
}
cout<<"answer: "<<p<<endl;
```

Задача 3. Найдите наибольший элемент массива.

```
int max=0;
for (int i=1; i<n; i++) {   if (a[i]>max)
   max=a[i];
}
cout<<"max: "<<max<<endl;
```

Задача 4. Найдите наименьший четный элемент массива. Если такого нет, то выведите первый элемент.

```
int imin=-1;
for (int i=0; i<n; i++) {
   if ((!(a[i]%2) && (imin==-1 || a[imin]>a[i]))
        imin=i;
}
if (imin==-1)
  cout<<a[0];
else
  cout<<a[imin]<<endl;
```

Задача 5. Преобразовать массив так, чтобы сначала шли нулевые элементы, а затем все остальные.

```
int i1=0, i2=n-1;
while (i1<i2)
{
   // www.itmathrepetitor.ru
   while (i1 < i2 && !a[i1])
       i1++;
   while (i2 > i1 && a[i2])
      i2--;
   if (i1 < i2)
   {
      int tmp=a[i1];
      a[i1]=a[i2];
      a[i2]=tmp;
   }
   i1++;
   i2--;
}
```

Задача 6. Найдите сумму номеров минимального и максимального элементов массива.

```
int imax=0, imin=0;
for (int i=1; i < n; i++) {
  if (a[i]>a[imax])
        imax=i;
    if (a[i]<a[imin])
        imin=i;
}
cout<<"answer: "<<imin+imax<<endl;
```

Задача 7. Найдите минимальный по модулю элемент массива.

```
int min=abs(a[0]);
for (int i = 1; i < n; i++) {
  if (min>abs(a[i]))
        min=a[i];
}
cout<<"abs min: "<<min<<endl;
```

Многомерные массивы

C++ позволяет создавать многомерные массивы. Простейшим видом многомерного массива является двумерный массив. Двумерный массив - это массив одномерных массивов. Двумерный массив объявляется следующим образом:

тип имя_массива[количество строк][количество столбцов];

Следовательно, для объявления двумерного массива целых с размером 10 на 20 следует написать:

int d[10] [20];

Посмотрим внимательно на это объявление. В противоположность другим компьютерным языкам, где размерности массива отделяются запятой, C++ помещает каждую размерность в отдельные скобки.

Для доступа к элементу с индексами 3, 5 массива d следует использовать запись d[3] [5]

Многомерные массивы

В C/C++ прямоугольный двумерный массив чисел реализует математическое понятие «матрица». Однако, в общем случае, двумерный массив — понятие гораздо более широкое, чем матрица, поскольку он может быть и не прямоугольным, и не числовым.

Определение автоматических многомерных массивов почти полностью совпадает с определением одномерных, за исключением того, что вместо одного размера может быть указано несколько:

```
const unsigned int DIM1 = 3;
const unsigned int DIM2 = 5;
int ary[DIM1][DIM2];
```

В этом примере определяется двумерный массив из 3 строк по 5 значений типа int в каждой строке. Итого 15 значений типа int.

Инициализация массива

При статической (определяемой на этапе компиляции) инициализации значения элементов массива перечисляются в порядке указания размеров (индексов) в определении массива. Каждый уровень многомерного массива заключается в свою пару фигурных скобок:

```
const unsigned int DIM1 = 3;
const unsigned int DIM2 = 5;
int ary[DIM1][DIM2] = {
    { 1, 2, 3, 4, 5 },
    { 2, 4, 6, 8, 10 },
    { 3, 6, 9, 12, 15 }
};
```

В примере показана статическая инициализация прямоугольного массива. Весь список инициализирующих значений заключён в фигурные скобки. Значения для каждой из 3 строк заключены в свою пару из фигурных скобок, значения для каждого из 5 столбцов для каждой строки перечислены через запятую.

Заполнение массива значениями

Многомерный массив заполняется значениями с помощью вложенных циклов. Причём, как правило, количество циклов совпадает с размерностью массива:

```
#include <iostream>
#include <iomanip>
using namespace std;
const unsigned int DIM1 = 3;
const unsigned int DIM2 = 5;
int ary[DIM1][DIM2];
int main() {
    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            ary[i][j] = (i + 1) * 10 + (j + 1);
        }
    }
    // ...
```

В этом примере каждому элементу массива присваивается значение, первая цифра которого указывает номер строки, а вторая цифра — номер столбца для этого значения (нумерация с 1).

Вывод значений элементов массива на консоль

В продолжение предыдущего примера можно написать:

```
    for (int i = 0; i < DIM1; i++) {
        for (int j = 0; j < DIM2; j++) {
            cout << setw(4) << ary[i][j];
        }
        cout << endl;
    }
    return 0;
}
```

В результате получим следующий вывод на консоль:

```
  11  12  13  14  15
  21  22  23  24  25
  31  32  33  34  35
```

Пример работы с трехмерным массивом

Для трёхмерного массива можно написать код, использующий те же приёмы:

```
#include <iostream>
#include <iomanip>
using namespace std;
const unsigned int DIM1 = 3;
const unsigned int DIM2 = 5;
const unsigned int DIM3 = 2;
int ary[DIM1][DIM2][DIM3];

int main() {
    for (int i = 0; i < DIM1; i++)
       {
        for (int j = 0; j < DIM2; j++)
        {
```

Пример работы с трехмерным массивом

```
        for (int k = 0; k < DIM3; k++)
            {
                ary[i][j][k] = (i + 1) * 100 + (j + 1) * 10 + (k + 1);
                 cout << setw(4) << ary[i][j][k];
            }
            cout << endl;
        }
        cout << endl;
    }
  return 0;
}
```

Здесь присваивание значения элементу массива и вывод на консоль происходят в одной группе циклов.

Первое знакомство с динамическими массивами

Фиксированные массивы обеспечивают простой способ выделения и использования нескольких переменных одного типа данных так как размер массива известен во время компиляции.

Поскольку массивам фиксированного размера память выделяется во время компиляции, то здесь мы имеем два ограничения:

1) массивы фиксированного размера не могут иметь длину, основанную на любом пользовательском вводе или другом значении, которое вычисляется во время выполнения программы;

2) фиксированные массивы имеют фиксированную длину, которую нельзя изменить.

Во многих случаях эти ограничения являются проблематичными. К счастью, C++ поддерживает еще один тип массивов, известный как динамический массив. Размер такого массива может быть установлен во время выполнения программы, и его можно изменить. Однако создание динамических массивов несколько сложнее фиксированных, поэтому мы поговорим об этом попозже.

Литература

1. Процедурное программирование на языках СИ и С++ : учебно-методическое пособие / Л. А. Скворцова [и др.]. — М.: РТУ МИРЭА, 2018. — 238 с. [Электронный ресурс]. Режим доступа: https://library.mirea.ru/books/53585
2. Трофимов В.В., Павловская Т.А. Алгоритмизация и программирование: учебник для академического бакалавриата. М.: Издательство Юрайт, 2017. [Электронный ресурс]. Режим доступа: https://www.intuit.ru/studies/courses/16740/1301/info
3. Уроки С++ с нуля. [Электронный ресурс]. Режим доступа: https://code-live.ru/tag/cpp-manual
4. Введение в языки программирования Си С++. [Электронный ресурс]. Режим доступа: https://www.intuit.ru/studies/courses/1039/231/info