Содержание:

Введение

В повседневной жизни встречаются довольно часто действия, которые повторяются. Выполняя их, тратится время, и как-то упростить этот алгоритм не всегда получается.

В теории программирования существуют циклические алгоритмы, обеспечивающие выполнение действий, которые повторяются. Этим сокращается объем кода и облегчается написание программ.

В мире существует огромное количество языков программирования, хотя большинство из них отличается лишь методом компиляции и синтаксисом.

Циклические алгоритмы, которые используются в них, имеют одинаковый смысл и практически не отличаются методами применения.

Век высоких технологий принес свои плоды и теперь с помощью языков программирования есть возможность описать и даже упростить любые действия. При этом нет необходимости описывать действия, которые повторяются шаг за шагом, а достаточно только использовать циклические алгоритмы.

Актуальность данной темы состоит в том, что циклические алгоритмы являются одними из основных составляющий почти всех программ. И их изучение является одним из краеугольных камней в подготовке квалифицированных программистов.

Целью работы является рассмотрение циклических алгоритмов и их использования в различных заданиях с программирования.

В соответствии к цели работы поставлены такие основные задания:

– рассмотреть основные алгоритмические структуры;

– проанализировать использование циклических алгоритмов в C++;

– рассмотреть особенности обработки массивов;

– привести примеры базовых алгоритмов для обработки одномерных массивов;

– привести примеры базовых алгоритмов для обработки двумерных массивов.

В данной курсовой работе используется язык программирования C++, поскольку он в нынешнее время является одним из наиболее распространенных и используемых.

Основные понятия о циклических алгоритмах

1.1. Алгоритмы, их свойства

В повседневной деятельности постоянно приходится сталкиваться с разными правилами, которые описывают последовательность действий с целью достичь определенного необходимого результата. Данные правила являются многочисленными. Например, мы придерживаемся определенных правил, чтобы позвонить, приготовить лекарство, сварить суп или вычислить какое-то математическое уравнение. Указанные примеры можно объединить понятием «алгоритм». [2]

Понятие алгоритма, относится к фундаментальным концепциям информатики и возникло задолго до появления персональных компьютеров и стало основным понятием математики. Понятие «алгоритм» происходит от имени узбекского математика Мугаммада бен Муса аль-Хорезми. Algorithmi – это латинское транскрипция имени аль-Хорезми, данное слово использовалось для обозначения правил выполнения таких арифметических действий: вычитание, сложение, умножение и деление над числами. Совокупность этих правил в Европе стали называть «алгоризм». Впоследствии это слово переродилось в алгоритм и стало собирательным названием отдельных правил определенного вида и не только правил арифметических действий. В течение длительного времени его употребляли только математики, обозначая правила решения различных задач.

Алгоритм – это точное правило, определяющее процесс преобразования исходных данных для получения конечных результатов при решении определенного класса задач. [1] В этом правиле задаются указания к выполнению некоторой системы операций в определенном порядке и правила их применения к начальным данным для решения задачи.

Алгоритм является фундаментальным понятием программирования. Ученые выделяют три основных класса алгоритмов: вычислительные, информационные и

управляющие [3].

Вычислительные алгоритмы – это алгоритмы, которые работают со сравнительно простыми типами данных, например числами, векторами, матрицами.

Информационные алгоритмы представляют собой набор простых процедур, которые обрабатывают большие объемы информации. Примером такой процедуры может быть поиск необходимой числовой или символьной информации, что соответствует определенным критериям. Эффективность работы данных алгоритмов зависит от организации данных.

Управляющие алгоритмы характерны тем, что данные для них поступают от внешних процессов, которыми они управляют. Результатами работы этих алгоритмов является выработка своевременной управляющей реакции на быстрое изменение входных данных.

Попытки найти и сформулировать в жестких терминах приемы и формы построения алгоритмов привели к возникновению самостоятельной дисциплины – теории алгоритмов, в которой раскрываются теоретические возможности разработки эффективных алгоритмов вычислительных процессов и их применения в прикладных приложениях.

Любой алгоритм обладает следующими свойствами:

1. Массовость – применимость алгоритма к любым задачам определенного класса. Это свойство обеспечивает решение любой задачи из класса однотипных задач при любых исходных данных. Так, алгоритм вычисления площади треугольника применим к любым треугольникам. Вместе с тем, существуют и такие алгоритмы, которые применяются только к единому набору исходных данных. К их числу относятся алгоритмы, используемые различными автоматами, например, автоматом для продажи газет или телефоном-автоматом.

2. Определенность (детерминированность) – набор указаний должен быть точен и не зависеть от исполнителя. Эта характеристика обеспечивает определенность и однозначность результата процесса, описываемого при заданных исходных данных. Каждый шаг должен быть четко и недвусмысленно определен и не должен допускать произвольной трактовки исполнителем.

3. Дискретность – разбиение процесса, который определяется алгоритмом, на отдельные элементарные операции, возможность выполнения которых человеком

или машиной не вызывает сомнений. Процесс, который определяется алгоритмом, должен иметь дискретный характер, то есть представлять собой последовательность отдельных шагов.

4. Ясность – понимание исполнителя о том, что надо делать для выполнения этого алгоритма. При этом исполнитель алгоритма, выполняя его, действует «механически», поэтому формулировка алгоритма должна быть очень точной и однозначной.

5. Результативность – это завершение процесса преобразования входной информации в результатную. Данное свойство указывает на то, что применение алгоритма к любому допустимому набору исходных данных за конечное число шагов обеспечивает получение определенного результата.

6. Формальность – результат выполнения алгоритма не должен зависеть от каких-либо факторов, которые не являются частью этого алгоритма. Любой исполнитель, способный воспринимать и выполнять указания алгоритма (даже не понимая их содержания), действуя по нему, может выполнить поставленную задачу.

Чтобы довести к пользователю алгоритмы, они должны быть формализованы по определенным правилам с помощью конкретных изобразительных средств. Методы, используемые для записи алгоритмов, в значительной степени определяются тем, для какого исполнителя назначается алгоритм. К основным методам записи алгоритмов относятся: словесный, формульно-словесный, блок-схемы алгоритмов, языки программирования.

В словесной записи алгоритма каждая операция формулируется на естественном языке в виде правила. Правила нумеруются, чтобы иметь возможность на них ссылаться, и указывается порядок их выполнения.

Формульно-словесный метод записи алгоритма основывается на инструкциях о выполнении конкретных действий в определенной последовательности с использованием математических символов и выражений пояснениями на естественном языке.[10]

Блок-схема алгоритма представляет собой графическое изображение процесса решения задачи в виде последовательности блоков специального вида, отражающих специфику преобразования информации и соединенных между собой линиями или стрелками. Внутри каждого блока кратко описывается содержание конкретного этапа решения алгоритма задачи.

Рассмотрим основные геометрические фигуры, применимые в блок-схемах (таблица 1)

Таблица 1.

Основные геометрические фигуры, применимые в блок-схемах

Фигура	Применение
	Начало или конец алгоритма
	Операции вычисления или присваивания
	Ввод или вывод данных
	Проверка условия

Различают обобщенную и подробную блок-схемы алгоритма. Обобщенная схема алгоритма изображает вычислительный процесс в общем плане на уровне типовых процессов обработки данных, например, во время обработки экономических данных типичными процессами является ввод данных, корректировка, сортировка, обработка данных и т. д. К составлению подробных схем приступают после тщательного анализа обобщенных схем алгоритмов.

Сейчас самым совершенным методом для записи алгоритма являются языки программирования, которые позволяют автоматизировать вычислительный процесс за счет того, что перевод инструкции с алгоритмического языка на язык персонального компьютера осуществляется автоматически с помощью специальных программ-трансляторов.

Появление языков программирования сблизила понятия алгоритма и программы.

1.2. Базовые структуры алгоритмов. Циклические структуры алгоритмов

Любой алгоритм содержит описание команд и определяет последовательность их выполнения. На первый взгляд кажется, что команды алгоритма всегда выполняются одна за другой, однако это не так. Для обеспечения такого свойства алгоритма, как массовость, его создают с учетом ввода любого набора допустимых данных. Из-за этого во многих случаях нельзя заранее предсказать, каким именно должен быть следующий шаг алгоритма. Отсюда возникает потребность в таких указаниях исполнителю, которые позволяли бы управлять его действиями по складывающейся ситуацией в процессе выполнения алгоритма.

По характеру управления различают три основных вида алгоритмов: линейные, разветвляющиеся и цикличные.

В простейшем случае алгоритм осуществляет одновременное выполнение всех по очереди заданных действий независимо от значений входных данных задачи. Например, для нахождения объема призмы нужно найти площадь ее основания, определить высоту призмы, найти их произведение. Эти действия нужно выполнить для расчета объема любой призмы. Алгоритм, который осуществляет выполнение одной и той же последовательности действий при любых допустимых входных данных задачи, называется линейным (рисунок 1).

Действие 1

Действие 2

Действие n

…

Рис 1. Линейная структура алгоритмов

Алгоритмы, которые предусматривают два возможных варианта действий, являются более сложными, чем линейные. Например, в алгоритме решения квадратного уравнения сначала нужно найти значение дискриминанта, а затем, в зависимости от знака, или сообщить об отсутствии действительных корней (если значение дискриминанта отрицательное) или найти их по соответствующим формулам (в противном случае).

Алгоритм, который осуществляет выполнение тех или иных действий в зависимости от результата проверки условия, называется алгоритмом с разветвлением или разветвляющимся (рисунок 2): [8]

Условие

Действие 1

Действие 2

Да

Нет

Рис. 2. Структура разветвляющегося алгоритма

Поскольку такой алгоритм содержит описание действий для обоих возможных вариантов, при каждом его выполнении реализуется только один из них. Который именно – зависит от заданного набора входных данных.

Третий вид алгоритмов составляют в случае повторного выполнения определенной последовательности действий.

Например, для подсчета суммы двух целых чисел (в столбик) нужно сначала вычислить сумму последних цифр чисел-слагаемых, записать последнюю цифру результата и перенести, если нужно, единицу в следующий разряд. Далее по аналогичному правилу нужно вычислить сумму предпоследних цифр чисел-слагаемых и т.д. Процедура повторяется, пока все цифры чисел-слагаемых не будут исчерпаны. Количество повторений зависит от количества цифр в заданных числах.

Алгоритм, осуществляющий повторное выполнение действий, называется алгоритмом с повторением или циклическим алгоритмом. Повторяющееся действие или группа действий называется телом цикла. Количество повторений тела цикла определяется поставленным условием, которое называется условием цикла. По результатам проверки условия осуществляется выбор: еще раз повторить тело цикла или перейти к другим действиям.

Различают две основные разновидности циклов: циклы с предусловием (рисунок 3) и циклы с постусловием (рисунок 4).

В цикле с предусловием условие цикла формулируется таким образом, чтобы повторное выполнение операций производилось, пока проверка условия дает результат «да». Поэтому такие циклы называют еще циклами «пока».

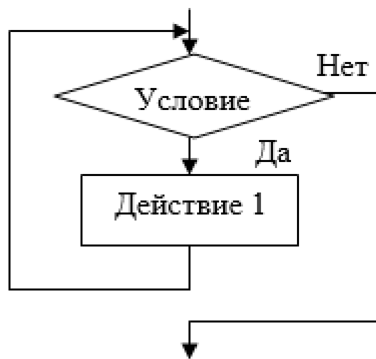

Рис.3. Циклическая структура с предусловием

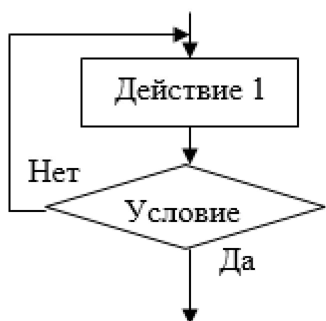

Рис.4. Циклическая структура с постусловием

Цикл с предусловием можно прочитать следующим образом:

пока проверка условия дает результат «да», выполнять действие.

Если при очередной проверке условия будет получено результат «нет», повторное выполнение действия будет прекращено и произойдет выход из цикла.

Например, для подсчета остатка от деления целого числа t на целое число n с помощью вычитания можно воспользоваться циклом:

пока t> n, уменьшить t на n.

В цикле с постусловием (рисунок 4) условие цикла формулируется противоположным образом: если очередная проверка условия дает результат "да", происходит выход из цикла. Цикл с постусловием можно сокращенно прочитать так:

повторять действие до получения результата «да» при проверке условия.

Например, подсчет остатка от деления целого числа t на целое число n (t> n) можно реализовать с помощью цикла: [9]

повторять уменьшить t на n до t <n.

Стоит обратить внимание на то, что является общим для обоих типов цикла:

– обе базовые структуры цикла являются замкнутыми;

– количество повторений цикла определяется его условием;

– выход из цикла происходит только через проверку условия цикла.

Наиболее существенная разница между типами циклов заключается в том, что тело цикла с постусловием обязательно выполняется хотя бы один раз – до первой проверки условия, а цикл с предусловием может не выполнятся ни разу, если при первой же проверке условия имеем результат «нет». Поэтому рассмотренные типы циклов не являются взаимозаменяемыми: цикл с постусловием можно заменить циклом с предусловием, а наоборот – нет.

Использование циклических алгоритмов при обработке массивов

2.1. Циклические алгоритмы в C++

Рассмотрим реализацию циклических алгоритмов на языке C++, в котором существую три разновидности операторов цикла:

– оператор цикла for;

– оператор цикла с предусловием while;

– оператор цикла с постусловием do … while.

Все операторы цикла непременно содержат следующие составные части:

– присваивания исходных значений (инициализация);

– условие продолжения цикла;

– тело цикла;

– изменение параметра (счетчика) цикла.

Оператор for обычно используется, когда есть заранее известное количество повторений или, когда условие продолжения выполнения цикла записывается кратким выражением. Примерами использования данного оператора являются вычисления сумм заданного количества слагаемых, поиск минимального (максимального) элемента последовательности чисел, сортировки элементов массива по возрастанию (убыванию) и т.д. [7]

Синтаксис оператора следующий:

```
for (<инициализация>; <условие>; <модификации>)

{

<тело цикла>;

}
```

Конструкция этого оператора состоит из трех основных блоков, размещенных в круглых скобках и отделенных друг от друга точкой с запятой (;) и команд (тела цикла), которые могут многократно повторятся. В начале выполнения оператора цикла, однократно, в блоке инициализации задаются начальные значения переменных, которые управляют циклом. Затем проверяется условие и, если оно выполняется, то управление переходит к выполнению тела цикла. Блок модификации меняет параметры цикла и, в случае истинности условия, выполнение цикла продолжается. Если условие не выполняется (false или равно нулю), то цикл прерывается и управление передается на оператор, следующий за оператором for. Существенным является то, что проверка условия выполняется в начале цикла. Это значит, что тело цикла может не выполниться ни разу, если условие сначала ошибочное. Каждое повторение (шаг) цикла называется итерацией.

Простой пример для вычисления суммы проиллюстрирует использования оператора for:

```
int s = 0;

for (int i = 1; i <= 10; i ++)

s + = i;
```

Этот оператор цикла можно прочитать так: "выполнить команду s + = i 10 раз (для значений i от 1 до 10 включительно, где i при каждой итерации увеличивается на 1)". В этом примере есть два присваивания начальных значений: s = 0 и i = 1, условие продолжения цикла: (i <= 10) и изменение параметра: i ++. Телом цикла является команда s + = i.

Порядок выполнения этого цикла компьютером такой:

1) присваиваются начальные значения (s = 0, i = 1);

2) проверяется условие (i <= 10);

3) если условие истинное (true), выполняется команда (или команды) тела цикла: к сумме, полученной на предыдущей итерации, добавляется новое число;

4) параметр цикла увеличивается на 1.

Далее возвращаемся к пункту 2. Если условие в пункте 2 не будет выполнено (false), произойдет выход из цикла. [5]

В операторе возможные конструкции, когда отсутствует тот или иной блок: инициализация может отсутствовать, если начальное значение задать предварительно; условие – если предполагается, что условие всегда истинно, то есть следует непременно выполнять тело цикла, пока не встретится оператор break; а модификации – если прирост параметра осуществлять в теле цикла. В этих случаях само выражение блока упускается, но точку с запятой (;) обязательно нужно оставить.

Стоит отметить, что оператор for допускает запись тела цикла непосредственно в своих параметрах. Например, сумму чисел можно вычислить следующим образом:

```
for (int s = 0, i = 1; i <= 10; s + = i ++)
```

В этом примере отсутствует тело цикла, а в блоке инициализации находится два оператора, которые разделены операцией "запятая" и задают начальные значения переменных s и i.

Рассмотрим простые примеры использования оператора цикла for. [6]

1) Вывод на экран чисел от 1 до 10 с их квадратами:

```
for (int i = 1; i <11; i ++)
```

```
cout << i <<" " << i * i;
```

Цикл выполняется при значениях i = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. После этого i примет значение 11 и условие (11 <11) не выполнится (false) – произойдет выход из цикла.

2) Вывод положительных нечетных целых от 1 до 100 с их квадратами. Цикл записывается аналогично предыдущему, но параметр будет увеличиваться на 2, то есть цикл будет выполняться для значений i = 1, 3, 5, 7, ..., 97, 99.

```
for (int i = 1; i <100; i + = 2)

cout << i <<" " << i * i;
```

3) Вычисление факториала F = n! (напомним, что факториал вычисляется по формуле n! = 1 * 2 * 3 * ... * (n-2) * (n-1) * n, например: 4! = 1 * 2 * 3 * 4 = 24). Приведем три аналогичных по действию формы записи оператора for:

```
1) int F = 1, n = 5,

for (int i = 1; i <= n; i ++)

F * = i;

2) int F, i, n = 5,

for (F = 1, i = 1; i <= n; F * = i ++)

3) int F = 1, i = 1, n = 5,

for (; i <= n;)

F * = i ++;
```

Для досрочного начала очередной итерации цикла можно использовать оператор перехода к следующей итерации continue, например:

```
for (i = 0; i <20; i ++)

{

if (array [i] == 0) continue;

array [i] = 1 / array [i];
```

```
}
```

Для заблаговременного выхода из цикла применяют операторы break (выход из конструкции) или return (выход из текущей функции). [19]

Некоторые варианты применения оператора for повышают его гибкость за счет возможности использования нескольких переменных-параметров цикла.

Например:

```
int top, bot;

char string [100], temp;

for (top = 0, bot = 98; top <bot; top++, bot– –)

{

temp = string [top];

string [top] = string [bot];

string [bot] = temp;

}
```

В этом примере для реализации записи строки символов в обратном порядке параметрами цикла являются две переменные top и bot, значение которых движутся навстречу друг другу. Заметим, что в блоке инициализации оператора for через запятую задаются начальные значения сразу двух переменных. Так же через запятую записаны модификации этих переменных после условного выражения.

Еще одним интересным вариантом применения оператора for является бесконечный цикл. Для организации такого цикла можно записать пустое условное выражение, а для выхода из цикла воспользоваться условным оператором if вместе с оператором break. [20]

Например:

```
for ( ; ; )

{
```

```
if (<некоторое условие>) break;

}
```

Циклы могут быть вложены друг в друга. При использовании вложенных циклов надо составлять программу таким образом, чтобы внутренний цикл полностью укладывался в тело внешнего цикла, то есть циклы не должны пересекаться. В свою очередь, внутренний цикл может содержать собственные вложенные циклы. Имена параметров внешнего и внутреннего циклов должны быть разными. Допускаются следующие конструкции:

```
foг(k = 1; k <= 10; k ++)

{

foг (i = 1; i <= 10; i ++)

{

foг (t = 1; t <= 10; t ++)

{

...

}

}

}
```

Вложенные циклы используются, например, для вывода на экран треугольника "звездочек":

*

**

***

****

Для вывода одной строки следует сформировать отдельный цикл, а для вывода нескольких таких строк необходимо вложить первый цикл в цикл, который будет

формировать переход к следующей строке. В первой строке должна быть только одна "звездочка", во второй – две, в третьей – три и т.д. То есть строка с номером i должна состоять из i "звездочек", поэтому внутренний цикл будет выполняться i раз. [18]

```
int i, j, m;

cout << "Ввести количество строк m:";

cin >> m;

for (i = 1; i <= m; i ++)

{

for (j = 1; j <= i; j ++) cout << "*";

cout << endl;

}
```

Операторы с предусловием и постусловием используются для организации циклов и являются альтернативными к оператору for. Обычно цикл с предусловием используется, если количество повторений заранее неизвестно, а для многократного повторения тела цикла известно условие, при истинности которого цикл продолжает выполнение. Это условие следует проверять каждый раз перед очередной итерацией. Например, при считывании данных из файла условием цикла является наличие данных непосредственно в файле, то есть повторять чтение данных следует до тех пор, пока указатель не будет указывать на конец файла.

Синтаксис цикла с предусловием следующий:

```
while (<условие>)

{

<тело цикла>

};
```

Последовательность операторов (тело цикла) выполняется пока условие является истинным, а выход из цикла осуществляется, когда условие станет ложным. Если условие является ошибочным при вхождении в цикл, то последовательность

операторов ни разу не выполнится, а управление будет передаваться к следующему оператору программы.

Цикл с постусловием используется, если есть необходимость проверять истинность условия каждый раз после очередной итерации. Как отмечалось выше, отличие цикла с предусловием от цикла с постусловием заключается в первой итерации: цикл с постусловием всегда выполняется по крайней мере один раз независимо от условия. [17]

Синтаксис цикла с постусловием следующий:

```
do

{

<тело цикла>

}

while (<условие>);
```

Последовательность операторов (тело цикла) выполняется один или несколько раз, пока условие не станет ложным. Оператор цикла do ... while используется в тех случаях, когда есть необходимость выполнить тело цикла хотя бы один раз, поскольку проверка условия осуществляется после выполнения операторов.

Если тело цикла состоит из одного оператора, то операторные скобки {} не является обязательны.

Операторы while и do ... while могут преждевременного завершиться при выполнении операторов break и return внутри тела цикла.

Рассмотрим отличие работы разных операторов цикла на примере вычисления суммы всех нечетных чисел в диапазоне от 10 до 100:

1) с использованием оператора for:

```
int i, s = 0;

for (i = 11; i <100; i + = 2)

s + = i;
```

2) с использованием оператора while:

```
int s = 0, i = 11;

while (i <100)

{

s + = i;

i + = 2;

}
```

3) с использованием оператора do ... while:

```
int s = 0, i = 11;

do

{

s + = i;

i + = 2;

}

while (i <100);
```

2.2. Особенности обработки одномерных массивов

Часто, в процессе разработки программы, возникает потребность хранить большое количество однотипных значений, которые должны поддаваться одинаковым методам обработки. Например, экзаменационные оценки студентов, следует вводить, выводить, анализировать (сравнение с "2" или "5"), изменять при необходимости и тому подобное. Такие однотипные значения имеет смысл хранить в одной переменной, перенумеровать их внутри этой переменной, предоставить доступ к этим значениям по индексу и обрабатывать эти значения в цикле (номер индекса должен совпадать с значением параметра цикла).

Массив – это упорядоченная совокупность однотипных элементов. Массивы широко применяются для хранения и обработки однородной информации, например таблиц, векторов, матриц, коэффициентов уравнений и т.д. [16]

Каждый элемент массива однозначно можно определить по имени массива и индексе. Имя массива (идентификатор) подбирают по тем же правилам, что и для переменных. Индексы определяют местонахождение элемента в массиве. К примеру, элементы вектора имеют один индекс – номер по порядку; элементы матриц или таблиц имеют по два индекса: первый означает номер строки, второй – номер столбца. Количество индексов определяет размерность массива. Например, векторы в программах – это одномерные массивы, матрицы – двумерные.

Индексами могут быть только переменные, константы или выражения целого типа. Значения индексов записывают после имени массива в квадратных скобках. При объявлении массивов в квадратных скобках указывается количество элементов, а нумерация элементов всегда начинается с нуля.

Различия массива от обычных переменных следующие: [14]

– общее имя для всех значений;

– доступ к конкретному значению по его номеру (индексу)

– возможность обработки в цикле.

Одномерный массив объявляется в программе следующим образом:

<тип данных> <имя_массива> [<размер массива>];

Тип данных задает тип элементов массива. Элементами массива не могут быть функции и элементы типа void. Размер массива в квадратных скобках задает количество элементов массива. В отличие от других языков, в C++ не проверяется выход за пределы массива, поэтому, чтобы избежать ошибок в программе, следует следить за размерностью объявленных массивов. Значение размера массива при объявлении может быть не указано в следующих случаях: [15]

– при объявлении массив инициализируется;

– массив объявлен как формальный параметр функции;

– массив объявлен как ссылка на массив, явно определенный в другом модуле.

Используя имя массива и индекс, можно обращаться к элементам массива:

<имя_массива> [<значение_ индекса>]

Значения индексов должны находиться в диапазоне от нуля до величины, на единицу меньше размера массива, который определен при его объявлении, поскольку в C ++ нумерация индексов начинается с нуля.

Например,

int A[10];

объявляет массив с именем А, содержащий 10 целых чисел; при этом выделяет и закрепляет за этим массивом оперативную память для всех 10-ти элементов соответствующего типа (int – 4 байта), то есть 40 байтов, следующим образом (рисунок 5):

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Рисунок 5 Изображение одномерного массива

Следовательно, при объявлении массива выделяется память, необходимая для размещения всех его элементов. Элементы массива с первого до последнего запоминаются в последовательно возрастающих адресах памяти. Между элементами массива в памяти промежутков нет. Элементы массива записываются один за другим поэлементно. [13]

Для получения доступа к i-му элементу массива А, можно написать А[i], где i – переменная цикла (счетчик цикла), которая может принимать значения от 0 до 9. При этом величина i умножается на размер типа int и представляет собой адрес i-го элемента массива А от его начала, после чего осуществляется выбор элемента массива А по сложившейся адресу. [12]

При объявлении массивов можно присваивать начальные значения его массива (необязательно всем), которые в дальнейшем в программе могут быть изменены. Если реальное количество инициализированных значений является меньше размерность массива, то остальные элементы массива приобретают значение 0.

Например,

```
int a[5] = {9, 33, -23, 8, 1}; // a [0] = 9, a [1] = 33, a [2] = - 23, a [3] = 8, a [4] = 1;
```

```
float b[10] = {1.5, -3.8, 10}; // b [0] = 1.5, b [1] = - 3.8, b [2] = 10, b [3] = ... = b [9] = 0;
```

Все операции с массивами выполняются с помощью операторов цикла. Например, ввод информации в массив:

```
for (int i = 0; i <n; i ++)

cin >> a [i];
```

Вывод информации из массива:

```
for (int i = 0; i <n; i ++)

cout << a [i] << "";
```

2.3. Особенности обработки двумерных массивов

Для многих структур данных изображения в виде одномерного массива является неприемлемым. Например, результаты матчей футбольного чемпионата удобнее подавать в виде квадратной таблицы. Для хранения таких структур данных применяют многомерные массивы, среди которых наиболее широко используются двумерные массивы (матрицы).

Как отмечалось в данном разделе, размерность массива определяется количеством индексов. Элементы одномерного массива имеют один индекс, двумерного массива (матрицы, таблицы) – два индекса: первый из них – номер строки, второй – номер столбца. При размещении элементов массива в памяти компьютера сначала меняется крайний правый индекс, затем остальные - справа налево. [11]

Многомерный массив объявляется в программе следующим образом:

<тип> <имя_массива> [<размерность1>] [<размерность2>] ... [<размерностьN>];

Количество элементов массива равна произведению количества элементов по каждому индексу. Например,

```
int a [3][4];
```

объявлено двумерный массив из 3-х строк и 4-х колонок (12-ти элементов) целого типа:

a [0] [0] a [0] [1], a [0] [2], a [0] [3],

a [1] [0], a [1] [1], a [1] [2], a [1] [3],

a [2] [0], a [2] [1], a [2] [2], a [2] [3];

Приведем еще несколько примеров объявления массивов:

float Mas1 [5] [5]; // Матрица 5x5 = 25 элементов вещественного типа.

char Mas2 [10] [3]; // Двумерный массив с 10x3 = 30 элементов символьного типа

double Mas3 [4] [5] [4]; // Трехмерный массив с 4x5x4 = 80 элементов вещественного типа

При объявлении массива можно присваивать начальные значения его элементов, причем необязательно всех, например:

1) int w [3] [3] = {{2, 3, 4}, {3, 4, 8}, {1, 0, 9}};

2) float C [4] [3] = {1.1, 2, 3, 3.4, 0.5, 6.8, 9.7, 0.9};

В первом примере объявлено и инициализирован массив целых чисел w[3][3]. Элементам массива присвоено значение из списка: w[0][0] = 2, w[0][1] = 3, w[0][2] = 4, w[1][0] = 3 и т.д.

Рассмотрим примеры для ввода и вывода элементов матрицы a.

Инициализировать матрицу a размерности nxm можно с помощью следующего кода:

```
for (i = 0; i <n; i ++)

for (j = 0; i <m; j ++)

cin >> a [i] [j];
```

Для вывода двумерного массива на экран используются следующие операторы:

```
for (i = 0; i <n; i ++)
```

```
{
for (j = 0; i <m; j ++)
cout << a [i] [j] << "";
cout << endl;
}
```

Типовые алгоритмы обработки массивов

3.1. Базовые алгоритмы обработки одномерных массивов

К базовым алгоритмам обработки одномерных массивов относят:

– вывод на экран элементов или их количества, соответствующие определенному критерию;

– нахождение минимального или максимального элемента массива;

– нахождение суммы, произведения элементов и т.д.

Приведем типичные примеры для обработки каждого из этих типов алгоритмов.

Пример 1. Вывести на экран все четные числа массива и их количество.

```
int i, n, k = 0, a [10]; // Объявление переменных
cout << "n =";
cin >> n; // Введение размерности
cout << "Элементы массива:";
for (i = 0; i <n; i ++) // Введение элементов массива
cin>>a[i];
for (i = 0; i <n; i ++)
```

```
if (a [i]% 2 == 0) // Нахождение парных элементов

{

cout << a [i] << ""; //вывод на экран парного элемента

k ++; //вычисление количества парных элементов

}

cout << "k =" << k; //вывод количества парных элементов
```

Пример 2. Найти значение минимального элемента одномерного массива и его порядковый номер.

```
int i, n, p;

float min, a [10], d;

cout << "n ="; //ввод размерности

cin >> n;

cout << endl;

cout << "Элементы массива:";

for (i = 0; i <n; i ++) //ввод элементов массива

cin>>a[i];

min = a [0]; //присваивание начального значения для минимума

for (i = 1; i <n; i ++) //вычисляет минимум массива

if (a [i] <min)

{

min = a [i]; //присваиваем значение в переменную min

p = i; //запоминаем индекс минимального элемента

}
```

```
cout << endl;

cout << "min =" << min << "," << "p =" << p << endl; //выводим результат
```

Пример 3. Найти сумму элементов массива.

```
int i, n;

float sum = 0, a [10];

cout << "n ="; //вводим размерность массива

cin >> n;

cout << endl;

cout << "Элементы массива:"; //вводим элементы массива

for (i = 0; i <n; i ++)

cin>>a [i];

for (i = 0; i <n; i ++) //находим сумму элементов

sum + = a [i];

cout << endl;

cout << "Summa =" << sum; //выводим сумму
```

3.2. Базовые алгоритмы обработки двумерных массивов

Приведем сначала перечень базовых операций над матрицами и их элементами. К таким операциям относятся:

– ввод и вывод матриц – рассматривался в разделе 2;

– создание новой матрицы по заданному алгоритму;

– поиск элементов матрицы по определенному критерию;

– выполнение определенных операций над компонентами матриц (перестановка строк и столбцов, умножение матриц и т.д.).

Приведем несколько примеров использования базовых алгоритмов обработки двумерных массивов.

Пример 4. Вычислить количество положительных элементов квадратной матрицы, расположенных по периметру и на ее диагоналях.

```
int k, i, j, N, a [20] [20];

cout << "N ="; //ввод размерности квадратной матрицы

cin >> N;

cout << "Input Matrix A" << endl; //ввод элементов матрицы A

for (i = 0; i <N; i ++)

for (j = 0; j <N; j ++)

cin >> a [i] [j];

// цикл прохода по главной и боковой диагоналям

for (i = k = 0; i <N; i ++)

{

if (a [i] [i]> 0)

k ++;

if (a [i] [N-i-1]> 0)

k ++;

}

// цикл прохода по периметру матрицы

for (i = 1; i <N-1; i ++)

{
```

```
if (a [0] [i]> 0)

k ++;

if (a [N-1] [i]> 0)

k ++;

if (a [i] [0]> 0)

k ++;

if (a [i] [N-1]> 0)

k ++;

}

// проверка, пересекаются ли диагонали, если размерность матрицы – нечетное число

if ((N% 2! = 0) && (a [N / 2] [N / 2]> 0))

k--;

cout << "k =" << k << endl; //вывод результат
```

Пример 5. Найти все элементы матрицы, значения которых больше нуля.

```
int k, i, j, N, a [20] [20];

cout << "N ="; //ввод размерности квадратной матрицы

cin >> N;

cout << "Input Matrix A" << endl; //ввод элементов матрицы A

for (i = 0; i <N; i ++)

for (j = 0; j <N; j ++)

cin >> a [i] [j];

// циклы прохода по элементах матрицы
```

```
for (i = 0; i <N; i ++)

for (j = 0; j <N; j ++)

if (a [i] [i]> 0)

cout<<a[i][j];
```

Пример 6. Вывести на экран транспонированную квадратную матрицу.

```
int i, j, N, a [20] [20];

cout << "N ="; //ввод размерности квадратной матрицы

cin >> N;

cout << "Input Matrix A" << endl; //ввод элементов матрицы A

for (i = 0; i <N; i ++)

for (j = 0; j <N; j ++)

cin >> a [i] [j];

// транспонирование матрицы

for (i = 0; i <N; i ++)

{

for (j = 0; j <N; j ++)

cout<<a[j][i];

cout<<endl;

}
```

Заключение

Современные концепции типов данных развиваются на протяжении последних 40 лет. В ранних языках программирования все структуры данных, соответствующие конкретным задачам, моделировались небольшим количеством основных структур

данных, поддерживаемых этими языками. Например, в версиях языка FORTRAN, разработанных до языка FORTRAN 90, связные списки и двоичные деревья обычно моделировались с помощью массивов.

На практике, при функционировании автоматизированных систем управления, информационных систем, измерительных комплексов и др., возникает необходимость обрабатывать большое количество различной информации. Например, показания температуры воздуха окружающей среды, стоимость товаров, значения координат движущихся объектов, характеристики приборов и других технических устройств и т.д. Программное обеспечение таких систем должно обеспечивать обработку, хранение, ввод-вывод больших объемов всевозможных данных. Язык программирования C++ позволяет эффективно разрабатывать, тестировать и отлаживать программы, связанные с обработкой массивов данных самой различной структуры.

В языке C++ под массивом понимается упорядоченный набор фиксированного количества однотипных данных.

Массивы, наряду с записями, строками, множествами, относятся к структурированному типу данных языка. Они могут быть одномерные и многомерные. При этом размер массива не ограничивается. Размерность на практике ограничивается лишь объемом рабочей памяти конкретного компьютера.

Список использованных источников

1. Джесс Либерти. Освой самостоятельно C++ за 21 день. Издательский дом «Вильямс». – 2014. – 230 с.
2. Борис Пахомов. C/C++ и MS Visual C++ 2010 для начинающих. БХВ-Петербург. – 2013. – 436 с.
3. Бьерн Страуструп. Программирование. Принципы и практика использования C++. Издательский дом «Вильямс». – 2016. – 258 с.
4. Айвор Хортон. Visual C++ 2010. Полный курс. Издательский дом «Вильямс». – 2015. – 300 с.
5. Дэвид Гриффитс, Дон Гриффитс. Изучаем программирование на C. Издательство «Эксмо». – 2013. – 400 с.
6. Прата С. Язык программирования C++. Издание 6. Издательский дом «Вильямс» – 2015. – 304 с.

7. Брайан Керниган, Деннис Ритчи. Язык программирования С++. Издательство «Невский диалект». – 2015. – 320 с.
8. Р. Лафоре. Объектно-ориентированное программирование в С++. Издательство «Питер». Издание 4. – 2014. – 628 с.
9. Хусаинов Б.С. Структуры и алгоритмы обработки данных. Примеры на языке Си. Учеб. пособие. – Финансы и статистика, 2014. – 464с.
10. Кубенский А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на С++. – СПб.: БХВ-Петербург, 2014. – 464с.
11. Седжвик Роберт. Фундаментальные алгоритмы на С++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./ Седжвик Роберт. К.: Издательство «ДиаСофт», – 2014. – 500 с.
12. Язык С++: Учеб. Пособие /И.Ф. Астахова, С.В. Власов, В.В. Фертиков, А.В. Ларин. – Мн.: Новое знание, 2013. – 203 с.
13. Лаптев В.В., Морозов А.В., Бокова А.В. С++. Объектно-ориентированное программирование. Задачи и упражнения. – СПб.: Питер. 2017. – 288 с.
14. Кнут, Дональд, Эрвин. Искусство программирования. Том 1. Основные алгоритмы. 3-е изд. Пер. с англ. –: Уч. пос. М.: Издательский дом. «Вильямс», 2014. – 720с.
15. С++ Стандартная библиотека. Для профессионалов. /Н. Джосьютис. – СП Питер, 2015. – 350 с.
16. Динман М.И. С++. Освой на примерах. – СПб.: БХВ-Петербург, 2016. – 260 с.
17. Харви Дейтел, Пол Дейтел. Как программировать на С++. Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2015. – 430 с.
18. Майерс С. Эффективное использование С++. 50 рекомендаций по улучшению ваших программ и проектов. Пер. с англ. – М.: ДМК Пресс; – СПб.: Питер. 2016. – 240с.
19. Штерн Виктор. Основы С++: Методы программной инженерии. – Издательство «Лори», 2013. – 860с.
20. Скляров В.А. Язык С++ и объектно-ориентированное программирование. Справочное пособие. – Минск. «Вышейшая школа». – 2014. – 478с.

Размещено на Allbest.ru