Работа №4

«Массивы данных»

Цель работы

Получение опыта работы с массивами данных

После выполнения работы студенты могут

- Использовать одномерные и многомерные массивы данных
- Считывать и записывать файлы в виде двухмерных массивов
- Осуществлять матричные преобразования с использованием массивов данных

Теоретическая часть

Обработка одиночных переменных не является задачей, для которой применение вычислительных машин является оправданным. Гораздо логичнее использовать ЭВМ там, где требуется обработка больших объемов однотипных данных. Человек в процессе такой обработки допускает ошибки, а машина способна работать часами, не ошибаясь. Простейшим примером машины, обрабатывающей массив данных, является цифровой фильтр. На вход такого фильтра по одному подаются отсчеты, а с выхода фильтра считываются отсчеты обработанного сигнала.

Наиболее полно потенциал вычислительного устройства раскрывается с применением устройства хранения данных с произвольным доступом, то есть **памяти**. Использование памяти позволяет обрабатывать данные в произвольном порядке, что значительно расширяет вычислительные возможности. Сложные данные, такие как изображения, видео или документы, представляются в виде последовательности байт, расположенных в памяти. Для работы с длинными последовательностями однотипных данных используются массивы.

В программировании **массив** — это набор однотипных переменных, расположенных в памяти последовательно друг за другом и объединенных одним идентификатором. Массив характеризуется своими размерами, то есть общим количеством содержащихся в нем переменных, а также размерностью

(одномерный, двухмерный и т.д.). Переменные в массиве являются независимыми, каждая переменная определяется своим индексом. Максимальный пределы параметров массива ограничены только возможностями компилятора. Помимо этого выделяют **статические массивы**, размер которых фиксирован на этапе выполнения, и **динамические массивы**, размер которых может изменяться.

В зависимости от типа вычислителя, работа с массивами может производиться по-разному. В классическом однопоточном режиме может одновременно выполняться только одна операция, поэтому работа с массивами превращается в циклическое повторение однотипных команд:

1. Чтение переменной из ячейки массива;

2. Обработка значения;

3. Запись нового значения в ячейку массива.

Для реализации этого набора команд необходимо записать цикл. Переменная, изменяющаяся в цикле, как правило является индексом элемента массива.

Массивы в языке С

Наиболее низкоуровневый подход к построению массивов, максимально приближенный к аппаратной реализации, предлагается в языке Си. Массив является обобщением обычной переменной и объявляется сходным образом:

```
int a;         // Одиночная переменная
int ar[10];    // Массив из 10 переменных типа int
```

Для Си переменная и массив отличаются только объемом выделенной памяти. Фактически, одиночная переменная является массивом единичной длины. Многомерный массив представляется как одномерный массив, состоящий из других массивов:

```
int R[4][4];  // Массив 4 на 4 элемента
```

Каждый элемент массива является независимой переменной, однако необходимо сперва выделить его из массива. Для этого служит оператор квадратных скобок. Квадратные скобки ставятся сразу после идентификатора массива, а в скобках записывается индекс:

```
int b = ar[3];
ar[0] = 11;
```

Элементы в массиве в языке С всегда нумеруются начиная с нуля. Соответственно, для массива `ar`, содержащего 10 элементов, номера будут лежать в диапазоне от 0 до 9. При этом, в отличие от других высокоуровневых языков, таких как Pascal, Си **не проверяет** индекс на выход за пределы массива. Программисту необходимо самостоятельно следить за тем, чтобы индекс всегда был в пределах от 0 до N-1, где N – размер массива.

Такой подход является отражением внутренней структуры памяти. Массив — это всего лишь адрес первого байта того участка памяти, который обозначен как массив, а оператор квадратных скобок говорит о том, что необходимо перейти по этому адресу, а затем сдвинуться на N элементов вперед. Любой блок памяти определяется двумя числами: адресом начала блока и количеством байт в блоке.

При работе с **многомерными** массивами индексация делается через несколько квадратных скобок. К примеру, для двухмерного массива нужно указывать два набора скобок. Поскольку двухмерный массив является по сути одномерным массивом из одномерных массивов, первые квадратные скобки возвращают адрес подмассива, а вторые возвращают конечный элемент:

```
int Ri = R[0][1];
```

Динамические массивы

Массивы, объявленные показанным выше способом, являются статическими. Их размер задается на этапе компиляции и не меняется в процессе работы. На практике размер обрабатываемого массива данных неизвестен заранее.

Иногда удается разделить обрабатываемые данные на участки небольшого размера и использовать статические массивы для обработки каждого из них. Когда условия задачи не позволяю так поступить, необходимо использовать динамические массивы и задействовать механизм динамической памяти.

В языке С для работы с динамической памятью используется **указатель** — специальная переменная, которая хранит адрес в памяти. Для объявления указателя необходимо поставить символ «*» между типом и именем переменной:

```
int a;      // Обычная переменная
int *pa;    // Указатель на переменную типа int
```

К указателю можно обращаться напрямую, как к обычной переменной, и тогда будет производиться модификация хранимого адреса. А можно произвести **разыменование** и обратиться к памяти, на которую он указывает (адрес которой он хранит). Для этого необходимо использовать тот же символ «*», что и при объявлении переменной:

```
*pa = 12; // Записать 12 по адресу, на который указывает pa
a = *pa;  // Присвоить a значение по адресу, на который указывает pa.
```

Указателю может быть присвоен адрес другой переменной того же типа, используя оператор «&»:

```
pa = &a;
```

Адрес, лежащий в указателе, может быть произвольным, и в общем случае нельзя сказать, принадлежит ли он памяти приложения и можно ли его разыменовывать. Существует специальный нулевой адрес, обозначаемый константой `NULL`. Данное значение следует записывать в указатель, который не указывает ни на какие данные, чтобы можно было быстро сказать, можно разыменовывать такой указатель или нет.

```
int p = NULL;
```

Можно создать указатель на любой тип данных. Технически все указатели ничем не отличаются, разница между ними существует только на этапе компиляции. Тип указателя — это тип данных, лежащий по его адресу. При этом существует указатель общего назначения, который просто хранит адрес, но не позволяет работать с данными по нему — это тип «`void *`». Перед использованием такой указатель нужно преобразовать к указателю на нужный тип:

```
void *v = &a;
int* pa = (int*) v; // Преобразование в тип int*
```

Целевое применение указателей — работа с динамической памятью. Можно запросить у операционной системы участок оперативной памяти произвольного размера для своих нужд, используя функцию `malloc`. Аргументом функции является размер необходимой области памяти в байтах, а результатом ее работы —

адрес этой области в виде `void*`. Для того, чтобы выделить при помощи `malloc` участок памяти, соответствующий массиву данных, необходимо использовать оператор `sizeof` и преобразование (приведение) типов. Нижеприведенная команда выделит 100 переменных типа `int` в динамической памяти:

```
int *par = (int*) malloc ( 100 * sizeof(int) );
```

Динамическая память является ресурсом, занимаемым приложением, поэтому по окончании работы с ней ее необходимо освободить. Для этого необходимо вызвать функцию `free` и передать ей в качестве аргумента адрес выделенной ранее области.

Необходимо, чтобы каждому вызову `malloc` соответствовал вызов `free`!

```
free(par);
```

С выделенным участком памяти можно работать как с массивом. Для доступа к отдельным элементам такого массива используется тот же оператор квадратных скобок:

```
par[0] = 1;
int c = par[3];
```

Более новые стандарты языка позволяют объявлять динамический массив так же, как статический. Для этого необходимо объявить статический массив, размер которого является значением какой-либо перемнной. Следующий пример показывает, как объявить массив, длина которого вводится вручную с клавиатуры:

```
int size;
cin >> size;
int vars[size];
```

Работа с таким массивом полностью аналогична работе с обычным статическим массивом.

В языке C++ данный механизм во многом остался неизменным. Изменился только механизм выделения и освобождения динамической памяти. Для этого используется пара операторов: `new` и `delete`:

```
int* pR = new int[100];
delete [] pR;
```

Число в квадратных скобках указывает не количество байт, а количество **элементов**, которое необходимо разместить в динамической памяти. Фактический объем памяти в байтах будет вычислен автоматически.

Для одиночных переменных синтаксис остается таким же, но указывать квадратные скобки не нужно.

Динамический двухмерный массив

Отдельную сложность в языке С представляет выделение динамических двухмерных массивов. Для этого приходится сначала выделить динамический массив, содержащий адреса строк, а затем выделить каждую строку по отдельности. Переменная, являющаяся таким массивом, должна быть «указателем на указатель». Следующий код иллюстрирует выделение двухмерного массива из 10 строк и 20 столбцов:

```
int **pR;
*pR = (int*) malloc( 10 * sizeof(int*) );
int y;
for(y=0; y<10; y++)
    pR[y] = (int*) malloc (20 * sizeof(int) );
```

Удаление такого массива происходит в обратном порядке. Сначала удаляются отдельные строки, а затем удаляется массив-столбец:

```
int y;
for(y=0; y<10; y++)
    free( pR[y] );
free pR;
```

Для того, чтобы избежать такой рутины, используют специальный прием: производят выделение одномерного массива, представляющего собой массив из отдельных строк, расположенных подряд. Размер такого массива равен M·N, где M и N – количество строк и столбцов соответственно:

```
int *pR = (int*) malloc ( M*N*sizeof(int) );
```

Для того, чтобы получить из такого массива элемент с координатами x и y, необходимо знать размер одной строки. Индекс элемента из его двухмерных

координат вычисляется следующим образом:

```
int el = pR[y*N + x];
```

Диаграмма на рисунке 1 показывает сравнение двух способов выделения двухмерных динамических массивов. В обоих случаях выделяется массив из трех строк и четырех столбцов.

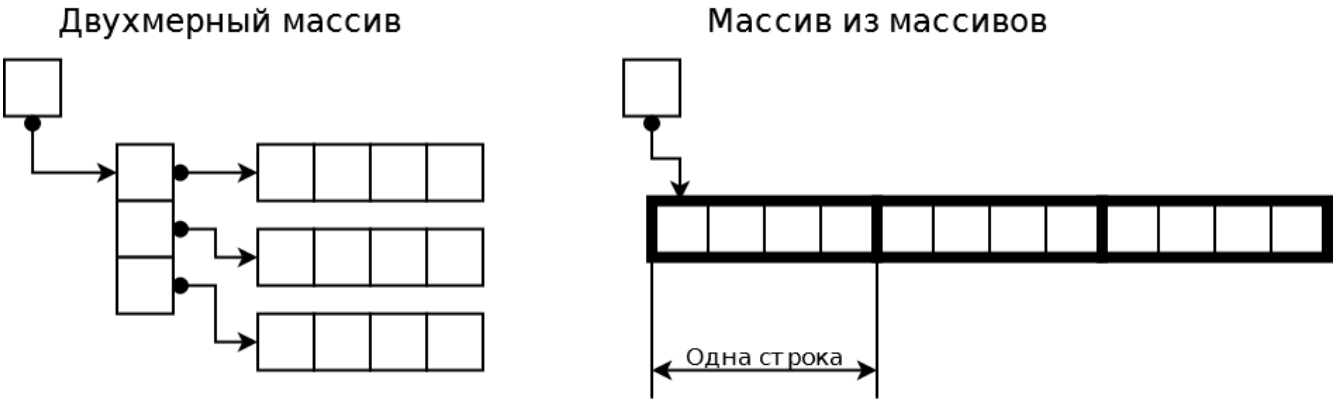

Рисунок 1 - Разные способы создания многомерных массивов

Зачастую можно избежать использования динамических массивов, обрабатывая большие данные небольшими порциями. Классический пример — обработка длинного текстового файла построчно. Для хранения одной строки достаточно выделить массив небольшой длины, не более 1 кБ, в который гарантированно поместится одна строка из файла, и который вполне допустимо сделать статическим. При этом языки C и C++ делают автоматическое преобразование из статического массива в указатель и обратно, если попытаться передать статический массив в функцию, ожидающую указатель:

```
char line[256];
FILE *F = fopen("file.txt","r");
while(!feof(F))
{
    fgets(line, 256, F);
    // Обработка строки
}
```

Динамические структуры данных.

Массивы данных в C и C++ являются очень мощным инструментом, однако работа с указателями и динамической памятью требует внимания и провоцирует на ошибки, способные нарушить работу приложения. Программисту приходится следить за такими рутинными вещами, как выход указателя за границы выделенной памяти, а также не забывать освобождать память по окончании работы с ней. При необходимости изменить размер массива программист вынужден создавать новый массив, скопировать в него элементы старого, и затем удалять старый.

Появление классов и шаблонов в языке C++ привело к разработке новых способов хранения больших объемов данных в памяти. Стандартная библиотека C++ содержит набор специальных шаблонных классов — **контейнеров** — которые могут хранить произвольное количество однотипных элементов. Контейнер берет на себя всю рутинную работу с памятью и обеспечивает проверку ошибок. Платой за это является чуть больший расход процессорного времени и оперативной памяти. Разные контейнеры отличаются по способу хранения данных и по-разному оптимизированы для различных операций.

Одним из основных классов-контейнеров является класс **`vector`.** Он хранит элементы в памяти подряд, аналогично обычному массиву. Как правило, `vector` выделяет память с определенным запасом, чтобы в случае небольшого изменения размера массива не перевыделять память. `Vector` хорошо оптимизирован на быструю выборку значений, однако сравнительно медленно работает с операциями вставки элементов в середину или в начало вектора. Для подключения вектора необходимо записать следующий код:

```
#include <vector>
using std::vector;
```

Другим классом-контейнером, который часто применяется, является **`list`**. Он хранит данные в виде двухсвязного списка: каждый элемент списка является структурой, поля которой — это непосредственно хранимые данные, а также два указателя: на предыдущий и на следующий элемент списка. У крайних элементов соответствующие указатели содержат `NULL`. При этом соседние элементы такого

списка могут располагаться в совершенно разных участках памяти. Структура двусвязного списка приведена на рисунке 2.

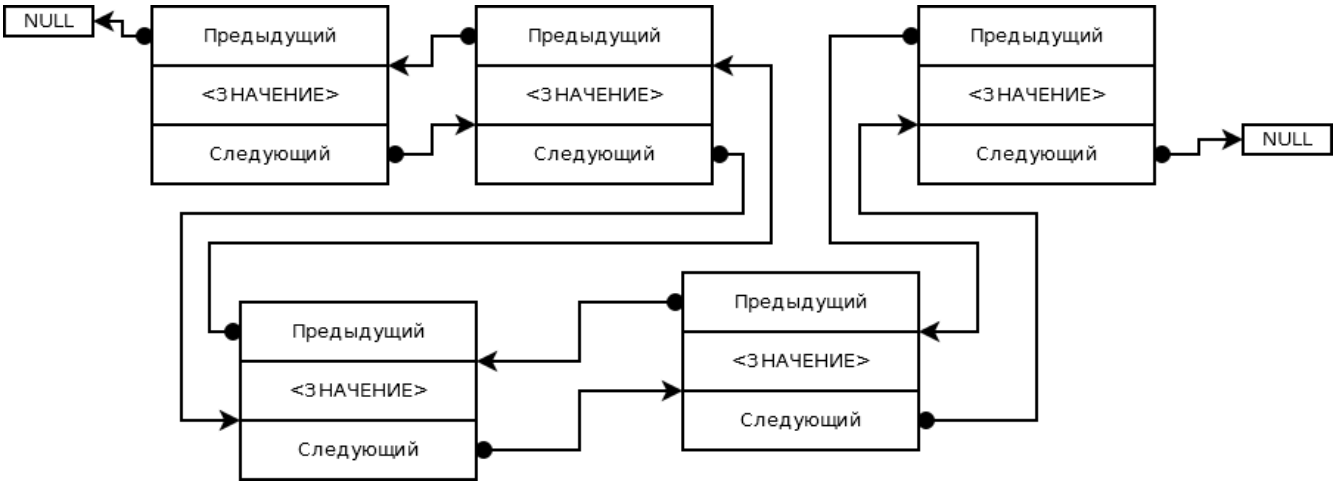

Рисунок 2 - Двусвязный список

Для такого списка операции добавления и удаления новых элементов или даже целых подсписков элементов не представляют никакого труда, так как не требуют перевыделения больших участков памяти. Однако, в отличие от `vector`, поиск в списке нужного элемента — затратная задача, так как необходимо пройти последовательно все элементы списка, чтобы найти нужный.

Помимо `vector` и `list`, существуют и производные от них контейнеры. Так, **`queue`** представляет собой очередь, то есть контейнер, оптимизированный для добавления элементов в начало и удаления их с конца; **`deque`** – двухсторонняя очередь, являющаяся расширенной версией `vector`, так как позволяет быстро добавлять элементы в начало и конец, но плохо справляется с добавлением элементов в середину; **`stack`** – колодец, добавляющий элементы в конец и точно так же убирающий их с конца. Также существуют ассоциативные контейнеры (например, **`map`**), в которых элементы адресуются не по индексу, а по ключу — целочисленной или строковой переменной, при этом ключи не обязательно располагаются по порядку.

Контейнеры C++ являются шаблонными классами, поэтому для того, чтобы использовать контейнер, необходимо при его объявлении записать в угловых скобках тот тип данных, который будет хранить контейнер. Например:

```
vector<double> field;
```

Очевидно, что точно так же можно создать контейнер из контейнеров.

```
vector<vector<double>> field_array;
```

Каждый контейнер является классом. Для контейнеров C++ характерен набор общих методов:

- Функция `size()` возвращает количество элементов, находящихся в контейнере.
- Функция `empty()` позволяет узнать, пуст ли контейнер. Проверка на пустоту происходит быстрее, чем сравнение размера контейнера с нулем.
- Функция `clear()` позволяет очистить контейнер.
- `push_back()` добавляет элемент в конец контейнера.
- `pop_back()` убирает последний по счету элемент из контейнера.
- `insert()` вставляет элемент в середину контейнера.
- `erase()` удаляет элементы из контейнера.
- `swap()` позволяет поменять местами два элемента в контейнере.
- `assign()` используется для заполнения контейнера значениями из другого контейнера или просто для заполнения одинаковыми значениями.
- Оператор прямого присвоения (=) используется для замещения содержимого контейнера содержимым другого такого же контейнера.

Также доступен следующий функционал:

- `push_front()` и `pop_front()` используются для добавления и удаления элементов в начале. Функции недоступны для `vector`.
- Оператор квадратных скобок `[]` используется для прямого доступа к элементам по индексу, как в обычном массиве. Недоступен для `list`.

Для того, чтобы пройти по содержимому контейнера, предусмотрены специальные классы — итераторы. **Итератор** — это объект, предоставляющий доступ к элементу контейнера и обеспечивающий навигацию по элементам.

Существуют функции контейнеров, используемые для получения итераторов

для характерных элементов. Функция `begin()` возвращает итератор, указывающий на элемент в начале контейнера, а `end()` возвращает специальный итератор, указывающий на не существующий элемент сразу после последнего элемента контейнера. Итераторы можно использовать как обычные переменные: увеличение значения итератора на 1 означает переход к следующему элементу, а уменьшение на 1 — переход к предыдущему. Для получения значения элемента следует «разыменовать» итератор как обычный указатель. В примере выводится значение элементов списка `list`:

```
std::list<int> mylist;
std::list<int>::iterator it
std::cout << "mylist contains:";
for (it=mylist.begin(); it != mylist.end(); ++it)
    std::cout << ' ' << *it;
```

В результате работы на экран будет выведена строка, отображающая значения элементов списка.

Начиная со стандарта C++11, можно использовать автоматическое присвоение типа переменной с помощью специального типа данных: `auto`. Использование `auto` упрощает работу с итераторами. Запишем предыдущий пример с его использованием:

```
std::list<int> mylist;
std::cout << "mylist contains:";
for (auto it=mylist.begin(); it != mylist.end(); ++it)
    std::cout << ' ' << *it;
```

Наконец, начиная со стандарта C++11 можно использовать простую форму записи оператора `for` для прохода по всем элементам контейнера:

```
std::list<int> mylist;
std::cout << "mylist contains:";
for (auto& v : mylist)
    std::cout << ' ' << v;
```

В приведенном примере переменная `v` последовательно будет ссылаться на

элементы списка. Поскольку `v` является ссылкой, возможна модификация элементов списка с ее помощью:

```
for (auto& v : mylist)
    v += 1;
```

Массивы в Python

Язык Python отличается тем, что в нем изначально отказались от массивов в стиле C, поскольку они требуют понимания тонкостей работы ЭВМ и являются источником трудноуловимых ошибок. В Python изначально встроены контейнеры трех типов:

Тьюпл представляет собой неизменяемый набор данных. Элементами тьюпла могут быть совершенно разнородные типы данных, в том числе другие тьюплы или списки. В тьюпле элементы нумеруются с нуля, так же как и переменные в массивах языка C. Особенность Python состоит в том, что имеют смысл и отрицательные номера элемента: они используются для нумерации с конца. Элемент с номером «-1» — это последний элемент тьюпла, «-2» — предпоследний и так далее. Для определения тьюпла следует указать несколько элементов через запятую, заключив их в круглые скобки:

```
tuple1 = (1, '2', "string", (2, 'Where the hell am I?'), 6)
```

Доступ к элементам тьюпла осуществляется при помощи оператора квадратных скобок:

```
first = tuple1[1]
last  = tuple1[-1]
```

При необходимости можно выбрать несколько идущих подряд элементов из тьюпла, указав в квадратных скобках два числа, разделенных двоеточием, при этом будут выбраны все элементы, включая первый и не включая последний. Нижеприведенный код сохранит в sub элементы со 2 по 4 включительно:

```
sub = tuple1[2:5]
```

Для определения размера тьюпла следует использовать стандартную функцию `len`:

```
tsize = len(tuple1)
```

Для создания тьюпла, состоящего из одной переменной, необходимо записать в тьюпл эту переменную и поставить после нее запятую:

```
t = (22,)
```

Для создания пустого тьюпла следует использовать пустые круглые скобки, либо встроенную функцию `tuple()`:

```
empty_t = ()
empty_t = tuple()
```

Список представляет собой набор каких-либо элементов, объединенных общим названием и доступным по индексу. Индексы в списке имеют тот же смысл, что и в тьюпле, но, в отличие от тьюпла, элементы списка можно изменять, а также добавлять и удалять элементы в произвольном месте списка. Список определяется так же, как и тьюпл, однако используются квадратные скобки:

```
ar = [1, "2", '3', "Ololo", 4]
```

Добавление элементов в конец списка происходит при помощи метода `append`, для добавления в середину существует метод `insert`. Удаление элементов производится при помощи метода `pop`. Пример ниже показывает, как можно модифицировать список:

```
ar = [1, "2", '3', "Ololo", 4]
ar[1] = 15             # Изменить элемент под номером 1
ar.insert(0, 10)       # Добавить число 10 перед нулевым элементом
ar.append("Last one") # Добавить строку "Last one" в конец списка
ar.pop(3)              # Убрать элемент под номером 3
```

Список может содержать всего один элемент или может вовсе быть пустым. При необходимости можно создать список необходимой длины, используя оператор умножения. Следующий код создаст массив из 100 нулевых элементов:

```
empty = [0] * 100
```

Словарь представляет собой набор элементов, в котором элементы идентифицируются не по индексу, а по специальному ключу. В качестве ключа может выступать все, что угодно, но на практике используются строки или номера. Словарь определяется как набор пар «КЛЮЧ : ЗНАЧЕНИЕ», разделенных запятыми, которые заключены в фигурные скобки:

```
rec = {"one": 1, "two":2}
```

Обращение к словарю происходит при помощи оператора квадратных скобок, в которые заключен ключ:

```
b = rec["one"]
```

Присвоение значения элементу словаря происходит таким же образом. При этом если ключ не существует в словаре, будет создан новый элемент с указанным ключом:

```
rec["three"] = 3 # Создать новый элемент "three":3
```

Для того, чтобы проверить, существует ли элемент с заданным ключом в словаре, существует оператор in:

```
b = ("one" in rec) # b == True если такой элемент есть в словаре
```

Разумеется, контейнеры Python ориентированы в первую очередь на безопасность и удобство использования, поэтому их быстродействие ниже, чем быстродействие контейнеров в компилируемых языках. Считается, что если необходимо максимальное быстродействие, узкие места программного кода следует писать на низкоуровневом языке, таком как C, и вызывать этот код из Python.

Для того, чтобы создать многомерный массив в Python, его нужно, аналогично C, составлять как массив одномерных массивов. Логичным решением кажется создание такого массива следующим образом:

```
ar = 5*[5*[0]]
```

Однако, при попытке присвоить значение одному из элементов массива, значение будет присвоено всем строкам сразу, так как Python не сделает 5 разных строк, а сделает 5 ссылок на одну строку. Для того, чтобы создать массив из независимых элементов, следует использовать следующую запись:

```
ar = [[0 for x in range(5)] for x in range(5)]
```

Получение элемента из такого массива производится через двойные квадратные скобки:

```
ar[0][1] = 1
```

Отдельно следует упомянуть **строки**. В Python они являются отдельным

типом данных, в отличие от C, где строка — это массив из отдельных символов. Со строками в Python можно работать посимвольно, используя оператор квадратных скобок, однако при этом нельзя их изменять. При необходимости изменить строку следует создать новую строку на основе предыдущей:

```
str = "hello"
c = str[1]    #  "e"
# str[0] = "H"  # Такая запись приведет к ошибке
str = "H" + str[1:-1]
```

Текстовый ввод-вывод

Массивы данных практически никогда не вводятся вручную. Как правило, они записываются в файл и затем читаются программой. Файлы можно разделить по способу хранения данных на текстовые и двоичные.

Двоичный файл хранит числа в виде копии участка памяти. Такая форма записи удобна для машины, так как не требует никаких преобразований при чтении или записи, однако непригодна для человека. Двоичный файл можно открыть при помощи шестнадцатеричного редактора, который представляет отдельные байты в виде их шестнадцатеричных двухсимвольных кодов, а также пытается интерпретировать эти байты как печатные символы, что возможно не всегда. Нижеприведенный двоичный файл содержит последовательно два числа: целое число 10 типа `int` и вещественное число 3.5 типа `float`. Каждое из чисел занимает в файле по 4 байта.

```
0a 00 00 00 00 00 60 40                           |......`@|
```

Текстовый файл хранит данные в человекопонятном виде, однако работа с таким файлом требует преобразования данных из текстового формата в двоичный при чтении, а также из двоичного в текстовый при записи. Преобразование из текстового формата возможно не всегда, так как строка может не соответствовать числу (например, содержит не только цифры).

Язык C содержит встроенные средства для работы с текстовыми и двоичными файлами. Набор функций для работы с файлами объявлен в заголовочном файле `stdio.h`. Для работы с файлом необходимо объявить специальную переменную — файловый дескриптор:

```
FILE *F;
```

Для того, чтобы можно было работать с файлом, его необходимо сначала открыть, то есть установить соответствие между дескриптором и файлом на диске. Для этого используется функция `fopen`, аргументами которой являются путь к файлу и режим, в котором следует открыть файл. Режим задается в виде строки:

- `"r"` — открыть файл для чтения
- `"w"` — открыть файл для записи. Если файл существует, он будет очищен. Если не существует — будет создан.
- `"a"` — открыть файл для дозаписи в конец файла.
- Суффикс `"b"` существует для чтения-записи в двоичном режиме, то есть спецсимволы записываются в файл "как есть", без преобразования. При его отсутствии файл открывается в текстовом режиме. К примеру, режим `"wb"` попытается открыть файл для записи в двоичном режиме. Выбор двоичного или текстового режима не ограничивает использование каких-либо функций, приведенных ниже.

Таким образом, для того, чтобы открыть файл на чтение в текстовом режиме, необходимо вызвать fopen со следующими аргументами:

```
F = fopen("file.txt", "r");
```

Чтение и запись двоичных данных осуществляются функциями `fread` и `fwrite`. Их аргументы следующие:

- Указатель на область данных, в которую будет производиться чтение, или из которой будет производиться запись.
- Размер одной переменной для чтения-записи.
- Количество записываемых или читаемых переменных.
- Файловый дескриптор.

```
int a=10;
fwrite(&a, sizeof(a), 1, F);
```

Возвращаемым значением `fread` и `fwrite` является количество

прочитанных или записанных элементов.

В текстовом виде с файлом можно работать при помощи функций `fprintf` и `fscanf`. Они работают точно так же, как и обычный `printf` и `scanf`, однако в качестве первого аргумента принимают файловый дескриптор:

```
fprintf(F, "%d\n", a);
```

Наконец, для построчной работы с файлом существуют функции `fgets` и `fputs`. Первая из них производит чтение из файла и записывает байты по указанному адресу до тех пор, пока не закончится участок памяти (его размер указывается вторым аргументом), либо пока не будет встречен символ переноса строки (`'\n'`), который также будет записан в указанный буфер. Вторая точно так же записывает одну строку в файл, добавляя после нее символ переноса.

```
FILE* F1 = fopen("file1.txt", "r");
FILE* F2 = fopen("file1.txt", "w");
char buf[256];
fgets(buf, 256, F1);
fputs(buf, F2);
```

Для определения окончания файла существует функция `feof`. При необходимости прочитать весь файл от начала до конца она используется следующим образом:

```
while(!feof(F))
{
    // Чтение из файла
}
```

По окончании работы с файлом следует закрыть его при помощи `fclose`:

```
fclose(F);
```

В языке **Python** ввод-вывод осуществляется аналогичным образом. Сначала следует открыть файл при помощи встроенной функции `open`. Аргументы данной функции — это имя файла и режим, в котором его следует открыть. Режим открытия файла полностью повторяет логику работы функции `fopen` в C. Для открытия файла на чтение в текстовом режиме необходимо выполнить следующий

код:

```
F = open("file.txt", "r")
```

С этого момента F становится специальным файловым объектом, через методы которого осуществляется ввод-вывод. Для чтения данных существует функция read, аргументом которой является количество байт, которое необходимо прочитать. Если аргумент не указан, read прочитает весь файл:

```
s = F.read(10)
fdump = F.read()
```

Запись в файл осуществляется при помощи функции write:

```
F.write("Hello\n")
```

Существует также функция readLine, считывающая одну строку из файла. Также можно прочитать файл построчно, объявив цикл по файловому дескриптору:

```
for a in F:
    print(a)
```

В python отсутствует функция проверки файла на окончание. Вместо этого следует ориентироваться на результат, возвращаемый функцией чтения. Если в результате read будет прочитано меньше байт, чем было запрошено или будет возвращена пустая строка, значит чтение файла закончено.

При работе **в двоичном режиме** существует некоторая разница между Python 2 и Python 3. Вышеприведенный пример будет работать во второй версии Python «как есть», если заменить "r" на "rb", однако в Python 3 двоичный ввод-вывод непосредственно через строку невозможен. Вместо этого необходимо преобразовать строку в объект типа bytes, указав кодировку для преобразования (например, ASCII или UTF-8):

```
F = open("file.bin", "wb")
F.write( bytes("Hello", "ASCII") )
```

При чтении данных в двоичном режиме необходимо декодировать данные из массива байт обратно в строку, используя кодировку:

```
F = open("file.bin", "rb")
```

```
s = F.read().decode("ASCII")
```

По окончании работы с файлом необходимо закрыть его, используя метод `close`:

```
F.close()
```

Порядок проведения работы

В данной работе необходимо реализовать алгоритм «Жизнь». Данный алгоритм представляет собой простой клеточный автомат, придуманный Д. Конвеем в 1970 году и заключается в последовательном рекуррентном расчете матрицы на основе ее предыдущего состояния. Алгоритм состоит из следующих правил:

- Место действия алгоритма — двухмерное поле, состоящее из ячеек — клеток.
- Каждая клетка на поле может находиться в одном из двух состояний: в живом или в мертвом. В силу конечных размеров поля, крайние клетки поля всегда находятся в «мертвом» состоянии. Каждая клетка, кроме крайних, имеет 8 соседей.
- Распределение живых клеток (начальное состояние поля) называется первым поколением. Каждое следующее поколение рассчитывается из предыдущего следующим образом:
  - Если мертвая клетка содержит ровно три живых соседа, то на следующем ходу она становится живой.
  - Если живая клетка содержит менее 2 или более 3 соседей, то на следующем ходу она умирает.
  - Все остальные клетки сохраняют свое состояние с предыдущего хода.

Пользователь задает начальную конфигурацию ячеек (первое поколение) и дальше просто наблюдает за работой алгоритма. Функционирование продолжается до тех пор, пока пользователь не прервет работу программы. Ввод данных в программу осуществляется при помощи текстового файла. Вывод данных производится в консоль.

Игра «Жизнь» содержит множество различных примечательных фигур. Самой известной из них является «планер». Эта фигура изменяет свою конфигурацию таким образом, что через 4 хода приходит в начальное состояние, но оказывается сдвинутой на 1 клетку. Последовательные стадии планера приведены на рисунке 3.

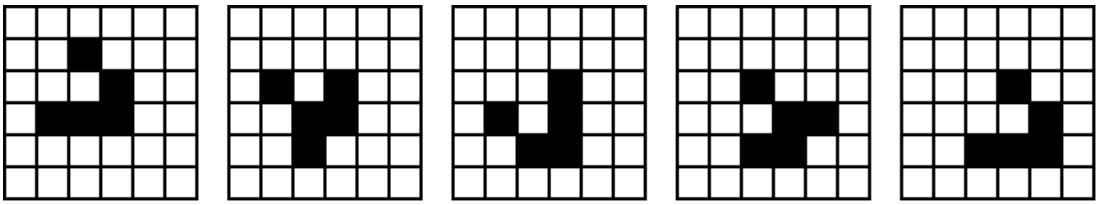

Рисунок 3 - Стадии жизни планера

При расчете следующего хода не следует производить преобразование исходной матрицы, так как это приведет к наслоению и к получению неверных результатов. Вместо этого следует создать вспомогательную матрицу с таким же размером, как и оригинальная, создать в ней состояние, соответствующее следующему ходу, на основе исходной матрицы, а затем по окончании расчета скопировать содержимое вспомогательной матрицы в основную.

В рамках данной работы начальная конфигурация будет задаваться в виде текстового файла, состоящего из символов «-», означающих пустую клетку, и символов «*», означающих живую клетку. Символы будут записаны в файл в виде прямоугольной матрицы. При этом граничные элементы матрицы должны соответствовать мертвым клеткам:

```
------
------
--***-
------
------
------
```

Для выполнения работы необходимо сделать следующее:

1. Создать основу приложения

   1.1. Создать новый проект и подключить к нему необходимые заголовочные файлы или модули.

   1.2. Объявить массив из строк или динамическую структуру под названием `field`, в который будет прочитан файл. При создании массива необходимо выделить его «с запасом», чтобы он гарантированно вместил в себя все

строки из файла.

В языке C++ рекомендуется объявить глобальный массив размером 100 на 100 элементов (заведомо больше массива, записанного в файле), а также две целочисленные переменные, хранящие фактический размер массива в ширину и высоту (количество занятых строк и столбцов в массиве):

```
char field[100][100];
int W=0, H=0;
```

В Python достаточно создать пустой список и затем заполнять его по мере чтения данных.

```
field = []
```

1.3. Объявить вспомогательный массив или динамическую структуру под названием `field2`, через которую будет производиться расчет. `field2` по своим размерам должен повторять `field`.

1.4. Организовать построчное чтение файла. Сохранить прочитанные строки в созданный массив или динамическую структуру.

В C++ элементами массива `field` являются строки, элементы которых — символы. Запись вида `field[a]` представляет собой строку, которую можно использовать в качестве аргумента таких функций как `fgets` или `fread`.

В Python строка представляет собой неизменяемую последовательность символов, что будет неудобно при последующей работе. Рекомендуется после чтения строки из файла преобразовать ее в список, состоящий из отдельных символов. Пример иллюстрирует преобразование строки в список:

```
s = "----**--"
ls = list(s)
print(ls)      # ['-', '-', '-', '-', '*', '*', '-', '-']
```

После чтения файла следует закрыть его.

1.5. Вывести на экран прочитанные данные и убедиться, что они соответствуют содержимому файла.

В Python перед выводом строки на экран необходимо собрать цельную строку

из отдельных символов, хранящихся в списке, чтобы избежать отображения оформления списка (квадратные скобки и кавычки):

```
s = ""
for c in ls:
    s += c
```

2. Подготовить расчет

2.1. Создать функцию `numNear`, рассчитывающую и возвращающую число «живых» соседей у клетки. Аргументами функции должны являться координаты клетки, а также сам массив.

2.2. При расчете количества соседей рекомендуется сделать цикл по координатам x и y в пределах [x-1, x+1] и [y-1, y+1] и исключить из этого цикла расчет для точки (x, y).

2.3. Проверить функционирование алгоритма. Для этого необходимо заменить в `field2` все элементы на числа, равные количеству соседей в соответствующей клетке `field`. При этом крайние элементы `field2` не принимают участия в расчете. Пример правильного расчета выглядит следующим образом:

Вывод исходного массива	**Вывод вспомогательного массива**
`------`	`------`
`------`	`-1232-`
`--***-`	`-1121-`
`------`	`-1232-`
`------`	`-0000-`
`------`	`------`

В языке C можно преобразовать число к цифре, пользуясь тем, что символы, соответствующие цифрам, расположены в таблице символов подряд, начиная с кода 0x30. Для получения символа из числа можно записать:

```
int d = 4;            // Число, полученное от numNear
char c = 0x30 + d;    // Символ, записываемый в ячейку массива
```

В языке C++ можно воспользоваться функцией `std::to_string()` для

преобразования из числа в строку типа `std::string`. Для получения символа, пригодного для вставки в массив, необходимо взять нулевой символ данной строки:

```
int d = 4;                          // Число, полученное от numNear
char c = std::to_string(d)[0];      // Символ, записываемый
                                    // в ячейку массива
```

2.4. В языке Python преобразование из числа в строку осуществляется созданием объекта типа `str`:

```
d = 4;          # Число, полученное от numNear
c = str(d)      # Символ, записываемый в ячейку массива
```

3. Реализовать алгоритм игры

3.1. Объявить бесконечный цикл.

3.2. В теле цикла необходимо рассчитать новое состояние игрового поля в массиве `field2`, используя данные из массива `field` и функцию `numNear`, в соответствии с правилами. При этом первая и последняя строка `field2`, а также первый и последний элементы всех остальных строк должны быть равны `'-'`, что соответствует пустой (мертвой) клетке.

3.3. По окончании расчета `field2` следует скопировать его содержимое в `field` и вывести его на экран.

3.4. В конце тела цикла следует добавить задержку для того, чтобы данные не выводились слишком быстро. Для этого можно использовать пустой цикл, количество итераций которого достаточно велико (необходимое количество итераций следует определить опытным путем).

В C++ можно использовать следующую конструкцию для создания паузы на указанное число миллисекунд:

```
#include <chrono>
#include <thread>
// …
std::this_thread::sleep_for(std::chrono::milliseconds(300));
```

В Python можно использовать функцию `time.sleep()`, обеспечивающую паузу на указанное число секунд (требуется импорт модуля `time`).

3.5. Запустить алгоритм и убедиться в его функциональности.

4. ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ. Модифицировать программу таким образом, чтобы возможно было существование нескольких конфликтующих популяций клеток:

4.1. Ввести дополнительный символ `'0'`, соответствующий клетке другой популяции.

4.2. Ввести новую функцию `numPattern`, аналогичную функции `numNear`, которая принимает дополнительный аргумент — символ популяции и возвращает количество соседей, равных данному символу.

4.3. Изменить правила генерации следующего хода:

- Новая клетка популяции X возникает на пустой клетке в том случае, если рядом с ней находится в точности три живые клетки популяции X и меньше трех клеток любой другой популяции.
- Остальные правила остаются неизменными.

4.4. Запустить полученный алгоритм и убедиться в его работе.

Вопросы для самоконтроля и подготовке к защите работы №4

1. Чем отличаются статические массивы от динамических?
2. Какие контейнеры в C++ вы знаете?
3. Какие существуют способы создания двухмерных массивов?
4. Как прочитать массив из файла?
5. Чем отличаются текстовые и двоичные файлы?