Лекция 4.6 «Структуры данных»

1. ОПРЕДЕЛЕНИЯ
2. КЛАССИФИКАЦИЯ
3. ПРОСТЫЕ БАЗОВЫЕ СТРУКТУРЫ ДАННЫХ
4. УКАЗАТЕЛИ
5. ПЕРЕЧИСЛЕНИЯ
6. МАССИВЫ
7. СТРУКТУРЫ
8. СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ C++

1. Определения

Данные – это представленная в формализованном виде информация, над которой можно выполнять операции: сбора, преобразования, передачи, хранения, обработки, отображения, сжатия, защиты и др.

Структура данных – множество элементов данных и множество связей между ними.

Тип данных определяет:

- **внутреннее представление данных** в памяти ПК, количество байт выделенное под переменную в оперативной памяти;
- **множество значений**, которые могут принимать величины данного типа;
- **операции и функции**, которые можно применить к величинам данного типа.

2. Классификация

Структуры данных

- Физическая структура – способ физического представления данных в памяти машины (структура хранения, внутренняя структура, структура памяти – синонимы).
- Логическая (абстрактная) структура – структура данных без учета ее представления в машинной памяти.

Структуры данных

- Процедуры отображения логической структуры в физическую
- Процедуры отображения физической структуры в логическую

Физическое представление обычно не соответствует логическому, и, кроме того, может существенно различаться в разных программных системах. Степень различия зависит от самой структуры и особенностей среды, в которой она должна быть отражена. Вследствие этого различия существуют процедуры, осуществляющие отображение логической структуры в физическую и наоборот. Эти процедуры обеспечивают доступ к физическим структурам и выполнение над ними различных операций, причем каждая операция рассматривается применительно к логической или физической структуре.

ОПЕРАТИВНАЯ ПАМЯТЬ

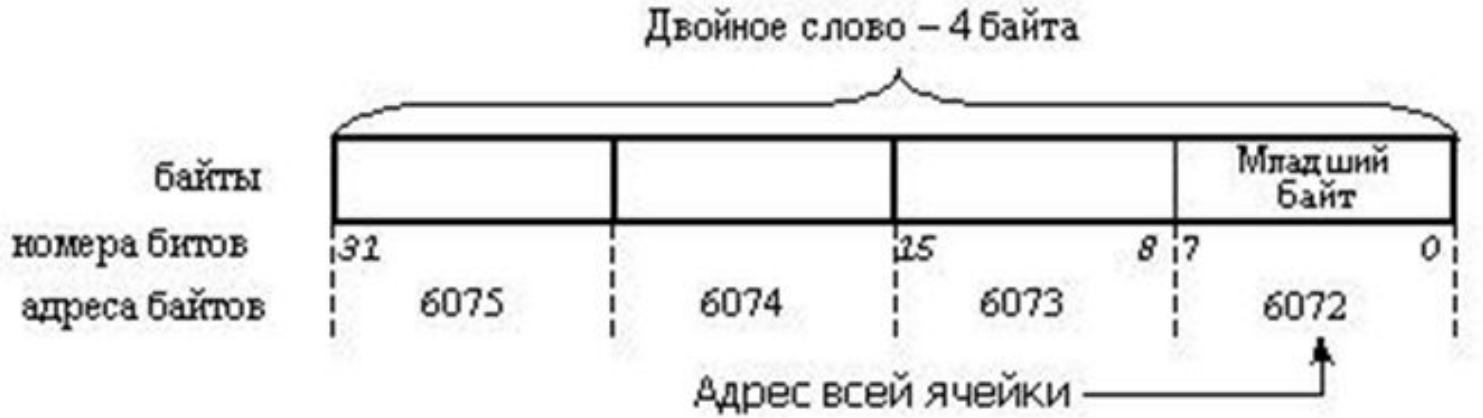
Оперативная память – последовательность пронумерованных ячеек, в которых может храниться двоичный код (в каждой ячейке хранится 1 байт информации).

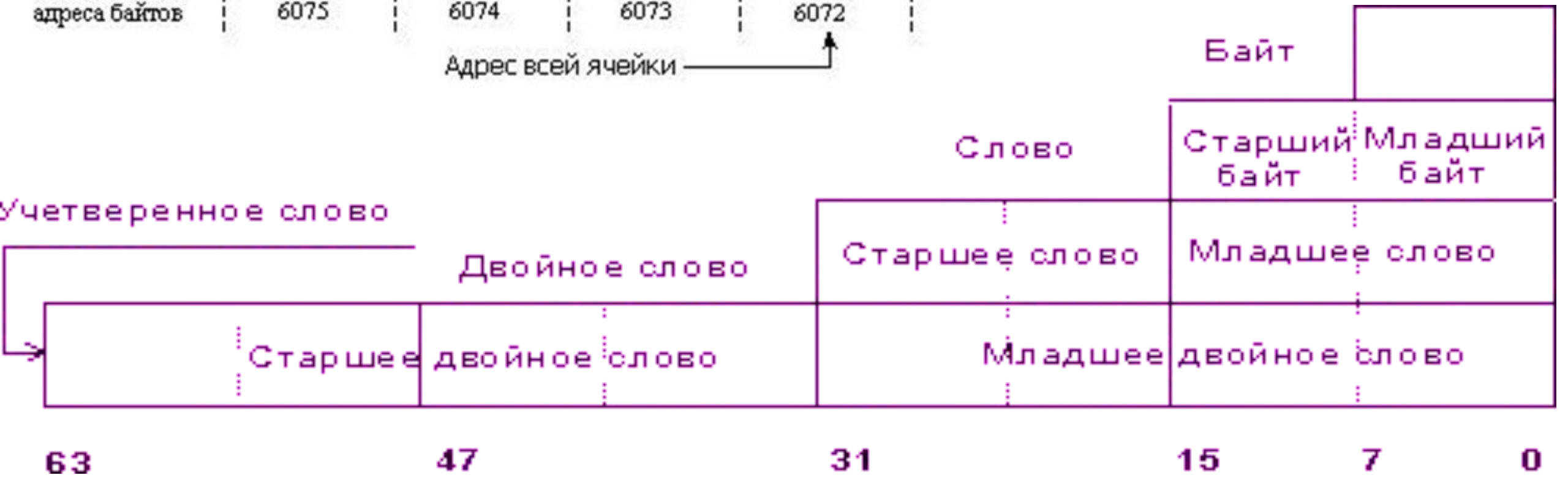
Номер ячейки	Информация в ячейке
268435456	11111111
...	...
4	00000000
3	11110000
2	00001111
1	10101010
0	01010101

Ячейки оперативной памяти

Модуль оперативной памяти – плоская пластина с электрическими контактами, по бокам размещаются БИС памяти. Может иметь информационную емкость 128, 256, 512 Мбайт.

Структуры данных в оперативной памяти






Двойное слово по адресу A
содержит 7AFE0636
Слово по адресу B содержит
FE06
Байт по адресу 9
содержит 1F
Слово по адресу 2 содержит
Слово по адресу 2 содержит
74CB
Слово по адресу 1 содержит
CB31
7A
FE
06
36
1F
23
0B
74
CB
31
E
D
C
B
A
9
8
7
6
5
4
3
2
1
0

Структуры данных

- Простые (базовые, примитивные) – не могут быть расчленены на составные части, большие чем биты.
- Интегрированные (структурированные, композитные, сложные) – состоят из простых и/или интегрированных структур

Структуры данных

Отсутствие или наличие явно заданных связей.

Несвязные структуры – векторы, массивы, строки, стеки, очереди.

Связные структуры – связные списки.

Структуры данных

Линейные структуры – структуры, в которых связи элементов не зависят от выполнения какого-либо условия

- **Строчные структуры**
- **Стек**
- **Очередь**
- **Дек**

Нелинейные структуры – у которых связи между элементами зависят от выполнения определенного условия.

- **Графы**
- **Деревья**
- **Плексы (сплетения)**

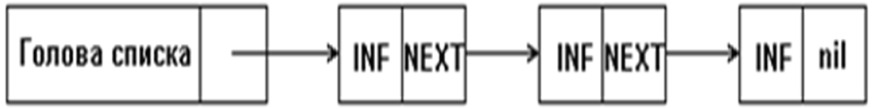
Структуры данных

Изменчивость – изменение числа элементов и/или связей между элементами структуры.

- Статические
- Полустатические
- Динамические

Структуры данных

Простые базовые	Статические	Полустати-ческие	Динамичес-кие
Числовые	Вектор	Стеки	Линейные связные списки
Символьные	Массивы	Очереди	Разветвленные связные списки
Логические	Множества	Деки	Графы
Перечисления	Записи	Строки	Деревья
Интервал	Таблицы		
Указатели			



Статические

Вектор — Вектор – структура данных с фиксированным количеством элементов одного и того же типа

Массивы — Массив – последовательность элементов одного базового типа *

Множества — Множество – набор неповторяющихся данных одного и того же типа

Записи — Запись – конечное упорядоченное множество полей, характеризующихся различным типом данных

Таблицы — Таблица – последовательность записей, которые имеют одну и ту же организацию **

* Массив – вектор с индексами элементов

**Таблица – записи с индексами записей

Полустати-ческие

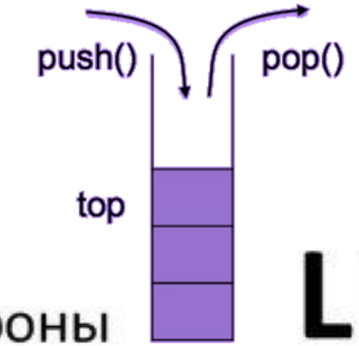
Стеки

Очереди

Деки

Строки

Stack in Data Structure



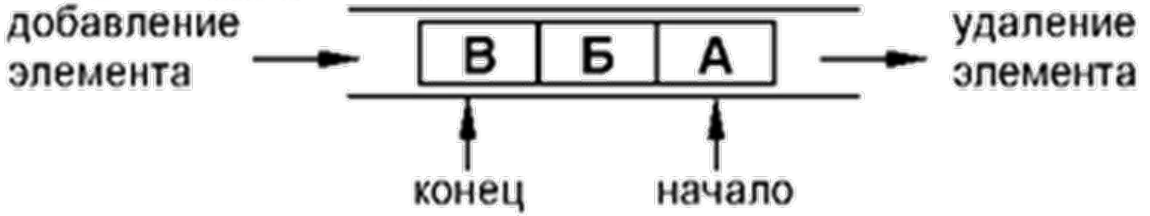
Последовательность, в которой включение и исключение элемента происходит с одной стороны

LIFO

Последовательность, в которую включают элементы с одной стороны, а исключают - с другой

Последовательность, в которой включение и исключение элементов происходит с двух стороны

FIFO



Строчные структуры – одномерные, динамические изменяемые структуры данных, различающиеся способами включения и исключения элементов

Структуры данных

Динамичес-кие

Линейные связные списки

Разветвленные связные списки

Графы

Деревья

Связный список – структура данных, элементами которой являются записи, связанные между собой указателями, хранящимися в самих элементах.

Граф – совокупность двух множеств: вершин и ребер.

Дерево – совокупность элементов, называемых *узлами* (один из которых определен как *корень*), и отношений, образующих иерархическую структуру узлов.

3. Простые базовые структуры данных

char	1 байт	Символы, целые числа от 0 до 255.
int	4 байта	Целые числа от -2147483648 до 2147483647.
float	4 байта	Числа с плавающей точкой от $\pm(3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{38})$
double	8 байт	Числа с плавающей точкой от $\pm(1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{308})$
bool	1 байт	true / false (0..255)
void	0 байт	Пустой тип

Приставки к простым типам данных

unsigned	без знака	*char, int*
long	длинное	*int, float*
short	короткое	*int*

4. Указатели

Указатели — это с самого начала переменные, уже в которых хранится адрес других переменных.

Чтобы пользоваться указателями, вам нужно использовать два оператора:

***** — показывает значение переменной по заданному адресу (показывает, кто живет в этом номере). Если вы используете оператор *, то вы занимаетесь операцией ***разыменование указателя***.

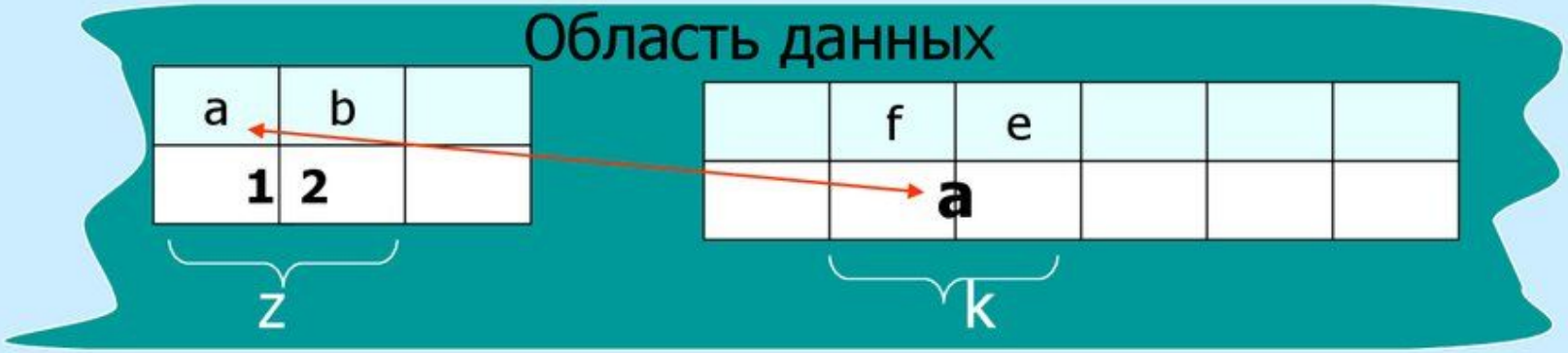
& — показывает адрес переменной (говорит, по какому адресу проживает этот человек).

& (амперсанд)

3.4.2. Ссылки

Ссылка – это адрес существующей переменной. Ссылка формируется добавлением знака «&» к имени переменной слева.

```
int z = 12; //объявлена и задана целая переменная
int *k = &z; // указателю k присваивается значение
//адреса переменной z.
```



* — используется, когда вам нужно значение переменной.

& — используется, когда вам понадобилось узнать адрес переменной.

Указатели в Си

Указатель - это специальное данное, которая содержит адрес другого данного.

Основные операции для работы с указателями:

* - взятие содержимого по адресу (*i - содержимое переменной с адресом i)

& - взятие адреса (&a - адрес переменной a).

Описание имеет вид:

*тип *имя_указателя;*

При описании указателя задается тип значения, на которое он указывает.

Примеры описаний: int *i, j, *pointj;

int v1, *pointv1=&v1, *p=(int*)200;

Операции над указателями

*p – разименование

p++ - переход к следующему элементу того типа, на который указывает указатель (p--)

p+i - сложение (вычитание) с целым – получаем указатель на i элементов правее (левее).

p – t - вычитание двух указателей одного типа - целое число, количество элементов между указателями.

p==t, p< t, p<=t, p>t, p>=t, p!=t

0 – указатель имеющий значение 0.

Указатели

```
int x = 5;
int *p = &x;          // указатель указывает на
память переменной x (его значение - адрес x)


cout << *p << endl;     //5 - разыменование
(*p)++;                  // увеличили значение
cout << *p << endl;     //6
cout << x << endl;      //6 изменилась x
```

5. Перечисления

- Перечисляемый тип представляет собой тип значений, содержащий конечное число именованных констант
- Синтаксис определения перечисления

```
enum <имя> [ : базовый тип]
{список-перечисления констант(через запятую)};
```

- Создание перечисления

```
enum DayTime { morning, day, evening, night };
```

- Использование перечисления

```
DayTime current;
if (current != night)
    // выполнить работу
```

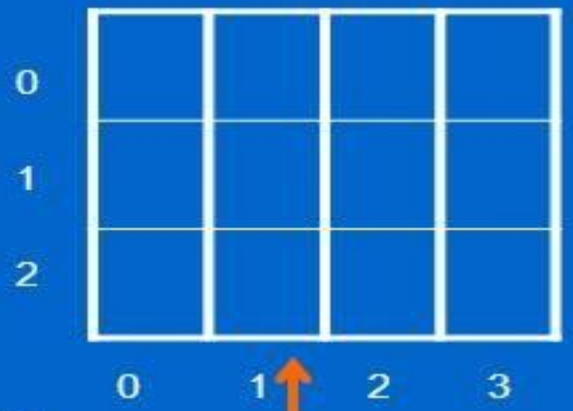
6. Массивы

Массив представляет собой переменную, содержащую упорядоченный набор данных одного типа.
В языке C++ массив не является стандартным типом данных. Напротив, он сам имеет тип: *char*, *int*, *float*, *double* и т.д.
Существует возможность создавать массивы массивов, указателей, структур и др.

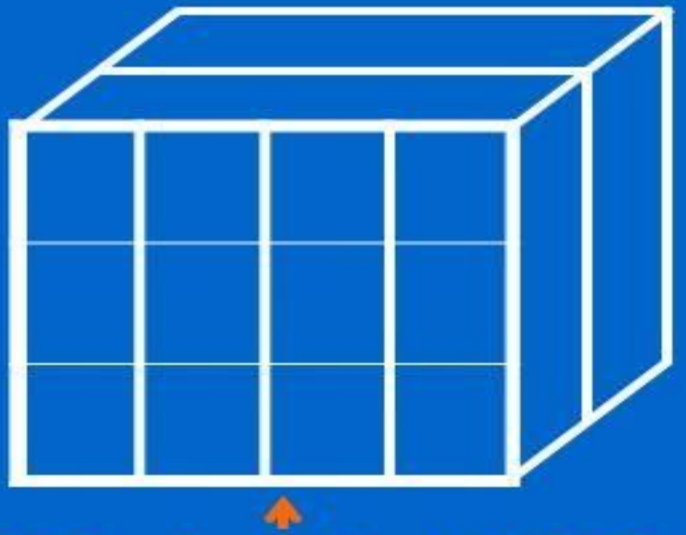
Многомерные массивы

Одномерный массив

Двумерный массив



Трехмерный массив



<Список размерностей>:

(0 To 7)

(0 To 2, 0 To 3)

(0 To 2, 0 To 3, 0 To 1)

Массивы могут иметь ***любые имена***, допустимые для переменных.

Размером массива называется количество его элементов, указываемое в квадратных скобках.

Например:

```
const int n=25;
double D[n]; // объявлен массив D из 25 чисел
             // типа double
int M[n+5];  // объявлен массив M из
             // 30 чисел типа int
```

Способы задания одномерных массивов

Явная инициализация

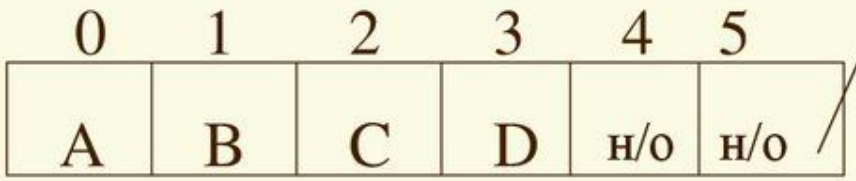
```
int I_Array[5] = {1, 2, 3, 4, 5};
I_Array[0]=1
I_Array[3]=4
```

Инициализация массивов при описании в Си

Инициализация - задание начальных значений.

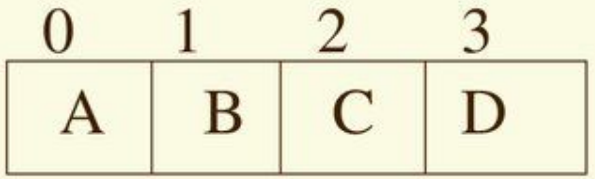
Одномерные массивы

```
char a[6]={'A', 'B', 'C', 'D'};
```



если a - локальная переменная

```
char a[ ]={'A', 'B', 'C', 'D'};
```



Размер массива определяется количеством инициализирующих значений

Доступ к элементам массива

К каждому из элементов можно обратиться по его номеру, расположенному в квадратных скобках после имени массива. Номера элементов массива начинаются ***с нуля***! Следовательно, **первым** элементом массива I_Array будет

I_Array[0],

вторым – I_Array[1] и т.д.

Многомерные массивы

Под **размерностью** массива понимают число индексов, которые необходимо указать для получения доступа к отдельному элементу массива.

Инициализация многомерных массивов

Например, для объявления массива содержащего
4 строки и 3 столбца, можно использовать следующую запись:
int M_Array [4][3];

```
int M_Array[4][3] = {1,7,8,-4,5,3,9,6,0,15,11,14};

int M_Array[4][3] = { {1, 7, 8},
                      {-4, 5,3},
                      { 9, 6, 0},
                      {15, 11,14} };
```

M_Array[2][1]=

M_Array[1][2]=

$$M\_Array[\,][\,] = \begin{pmatrix} 1 & 7 & 8 \\ -4 & 5 & 3 \\ 9 & 6 & 0 \\ 15 & 11 & 14 \end{pmatrix}$$

Для определения **размера массива** можно воспользоваться следующей функцией: ***sizeof*(*I_Array*)**, которая возвращает размер массива ***I_Array*** в **байтах**.
Если этот размер разделить на размер одного элемента массива, то получим количество элементов массива.

Например, если массив ***I_Array*** объявлен как массив целых чисел (***int***), то количество его элементов (размерность) *k* можно вычислить по формуле:

Использование массивов с динамическим выделением памяти

Массивы с динамическим выделением памяти используют, когда размер массива не известен на этапе компиляции. Реальный размер массива может вычисляться в программе или вводиться пользователем — неважно.

Память для массива выделяется оператором **new** в форме

new тип [количество_элементов].

Тип выражения, определяющего размер (количество элементов) массива должен быть целочисленным. Также это выражение может быть и константным.

Когда работа с массивом закончена, память, выделенную под массив необходимо освободить. Это делается с помощью оператора **delete** в форме

delete [] имя_переменной.

После того, как память освобождена, работать с массивом нельзя.

Динамические массивы в C++

Можно выделять сразу несколько ячеек динамической памяти, получая динамический массив. Для этого его размер указывается в квадратных скобках после типа **int [n]**. Чтобы удалить динамический массив и освободить память используется оператор **delete[]**.

```
int* p;
p = new int [12];
for (int i=0; i<12; i++) {
        *(p+i) = i + 1;
        cout << *(p+i) << ' '; // 1 2 3 ... 12
}
delete [] p; // память освобождена
```

Сразу после создания динамический массив автоматически заполняется нулями.

```
#include <iostream>
using namespace std;
int main()
{
    int num; // размер массива
    cout << "Enter integer value: ";
    cin >> num; // получение от пользователя размера массива
    int *p_darr = new int[num]; // Выделение памяти для массива
   // Заполнение массива и вывод значений его элементов
  for (int i = 0; i < num; i++) {
        p_darr[i] = i;
        cout << "Value of " << i << " element is " << p_darr[i] << endl;
    }
    delete [] p_darr; // очистка памяти
    return 0; }
```

7. Структуры

Структуры

- Создание структуры

```
public struct Employee
{
    string firstName;
    int age;
}
```

- Использование структуры

```
Employee companyEmployee;
companyEmployee.firstName = "Joe";
companyEmployee.age = 23;
```

Структуры могут хранить элементы разных типов

Структуры. Понятию структуры можно легко найти аналог в повседневной жизни. Обычная **записная книжка**, содержащая

- адреса друзей,
- телефоны,
- дни рождений и прочую информацию,

по сути своей является структурой взаимосвязанных элементов. Список файлов и папок в окне *Windows* – тоже структура.

***Структура* – это группа переменных *разных* типов, объединенных в единое целое**.

Структура создается с помощью ключевого слова ***struct***, за которым следует имя структуры, а затем – список членов структуры. ***ИмяСтруктуры*** становится именем нового типа данных и может использоваться для создания переменных этого типа.

Описание структуры в общем виде выглядит следующим образом:

```
struct ИмяСтруктуры
{  тип1 имя1;
   тип2 имя2;
   тип3 имя3;
   . . .
 тип-n имя- n };
```

Например, создадим структуру ***student***:

```
struct student
{   char Fam [20];
    char Imay [15];
        char Otch [20];
    int Nomer_zach;
        int Ex_math;
        int Ex_inf;
        int Ex_fiz;    };
```

В C++ создать **экземпляр** рассмотренной выше структуры ***student*** (переменную нового типа данных) с именем ***Ivanov*** можно следующим образом:

```
student   Ivanov;
```

Доступ к членам структур. Доступ к отдельному члену структуры можно получить с помощью оператора точки:

имя_переменной.член_структуры

Например, в языке C++ записать информацию в поле ***God_rogden*** экземпляра структуры ***student Ivanov*** можно с помощью следующего выражения:

cin >> Ivanov.God_rogden;

Вывести полученное значение на экран можно так:

cout << Ivanov.God_rogden;

Указатели на структуры.

Для получения доступа к отдельным членам структуры могут применяться **указатель** и **оператор –>** (селектор выбора).

```
struct student {
    char Fam [20]; char Imay [15];
    char Otch [20];
    int Nomer_zach; int Ex_math; int Ex_inf;     int Ex_fiz;};
int main()
{
student Ivanov;
cout << "Vvedite Familiu="; cin >> Ivanov.Fam;
cout << "Vvedite Imay=";    cin >> Ivanov.Imay;
cout << "Vvedite Otchestvo=";
cin >> Ivanov.Otch;
```

```
Vvedite Familiu=Alekseev
Vvedite Imay=Sergey
Vvedite Otchestvo=Viktorovich
Vvedite nomer zachetnoy knigki=1021
Vvedite ocenku po matematike=3
Vvedite ocenku po informatike=2
Vvedite ocenku po fizike=2

Student Alekseev S.V. ne sdal:
-  exzamen po fizike
-  exzamen po informatike
```

```
struct sotrudnik {
                char Fam [20];
                int Nomer_udost;
                int Stag;};
int main()
{
sotrudnik Sev_ROVD[5]; int k=0;
for (int i=0; i<5; i++)
  {cout << "Vvedite Familiu=";
   cin >> Sev_ROVD[i].Fam;
   cout << "Vvedite Nomer slug. udostovereniay=";
   cin >> Sev_ROVD[i].Nomer_udost;
   cout << "Vvedite stag slugby v OVD=";
   cin >> Sev_ROVD[i].Stag; }
```

```
Uvedite Familiu=Alekseev
Uvedite Nomer slug. udostovereniay=109023
Uvedite stag slugby v OVD=8
Uvedite Familiu=Antonov
Uvedite Nomer slug. udostovereniay=289910
Uvedite stag slugby v OVD=12
Uvedite Familiu=Sidorenko
Uvedite Nomer slug. udostovereniay=001928
Uvedite stag slugby v OVD=18
Uvedite Familiu=Pronin
Uvedite Nomer slug. udostovereniay=290121
Uvedite stag slugby v OVD=15
Uvedite Familiu=Takaev
Uvedite Nomer slug. udostovereniay=100928
Uvedite stag slugby v OVD=11

Rezul'taty poiska:
1. Antonov=12 let
2. Pronin=15 let
3. Takaev=11 let
```

```
struct sotrudnik {
char Fam [20];
int Nomer_udost;
int Stag;
};
void vvod_dannyh (sotrudnik *v_point);
void poisk (sotrudnik *p_point);
```

```
int main()
{
sotrudnik Sev_ROVD[5], *point;
point=&Sev_ROVD[0];
vvod_dannyh (point);
cout << endl;
cout << "Rezul'taty poiska:" << endl;
poisk (point);
return 0;
}
```

```
void vvod_dannyh (sotrudnik *v_point)
{
for (int i=0; i<5; i++)
  {cout << "Vvedite Familiu=";
   cin >> v_point->Fam;
  cout << "Vvedite Nomer slug. udostovereniay=";
  cin >> v_point->Nomer_udost;
  cout << "Vvedite stag slugby v OVD=";
  cin >> v_point->Stag;
  v_point++;}
}
```

```
void poisk (sotrudnik *p_point)
{
int k=0;
for (int i=0; i<5; i++)
  {
   if (p_point->Stag >= 10 && p_point->Stag <= 15)
    {k++;
     cout << k <<". "<< p_point->Fam<<"=";
     cout <<p_point->Stag<<" let";
     cout << endl; }
   p_point++;
  }
if (k==0)
  cout<<"Sotrudnikov s ukazamnnym stagem net!"<< endl;
}
```

```
Vvedite Familiu=Alekseev
Vvedite Nomer slug. udostovereniay=109023
Vvedite stag slugby v OVD=8
Vvedite Familiu=Antonov
Vvedite Nomer slug. udostovereniay=289910
Vvedite stag slugby v OVD=12
Vvedite Familiu=Sidorenko
Vvedite Nomer slug. udostovereniay=001928
Vvedite stag slugby v OVD=18
Vvedite Familiu=Pronin
Vvedite Nomer slug. udostovereniay=290121
Vvedite stag slugby v OVD=15
Vvedite Familiu=Takaev
Vvedite Nomer slug. udostovereniay=100928
Vvedite stag slugby v OVD=11

Rezul'taty poiska:
1. Antonov=12 let
2. Pronin=15 let
3. Takaev=11 let
```

8. Стандартная библиотека шаблонов C++

STL(англ. **Standard Template Library**): стандартная библиотека шаблонов C++

STL обеспечивает стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных.

Ядро библиотеки образуют три элемента:

Контейнеры (containers) - это объекты, предназначенные для хранения набора элементов.

Алгоритмы (algorithms) выполняют операции над содержимым контейнера (инициализации, сортировки, поиска, замены содержимого контейнеров).

Итераторы (iterators) - это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера и сканировать его элементы.

Литература:

https://ru.cppreference.com/w/

Типы итераторов

Итераторы ввода (input iterator) поддерживают операции равенства, разыменования и инкремента: ==, !=, *i, ++i, i++, *i++. Специальным случаем итератора ввода является istream_iterator.

Итераторы вывода (output iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента: ++i, i++, *i = t, *i++ = t. Специальным случаем итератора вывода является ostream_iterator.

Однонаправленные итераторы (forward iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание: ==, !=, =, *i, ++i, i++, *i.

Двунаправленные итераторы (bidirectional iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (--i, i--, *i--), что позволяет им проходить контейнер в обоих направлениях.

Итераторы произвольного доступа (random access iterator) обладают всеми свойствами bidirectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ к элементу по индексу: i += n, i + n, i -= n, i - n, i1 - i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2.

Классы-контейнеры STL

Наименование класса	Описание	Заголовочный файл
bitset	множество битов	**<bitset>**
vector	динамический массив	**<vector>**
list	линейный список	**<list>**
deque	двусторонняя очередь	**<deque>**
stack	стек	**<stack>**
queue	очередь	**<queue>**
priority_queue	очередь с приоритетом	**<queue>**
map	ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано одно значение	**<map>**
multimap	с каждым ключом связано два или более значений	**<map>**
set	множество	**<set>**
multiset	множество, в котором каждый элемент не обязательно уникален	**<set>**

Итераторы STL
#include <iterator>

Итератор	Описание
begin()	указывает на первый элемент
end()	указывает на элемент, следующий за последним
rbegin()	указывает на первый элемент в обратной последовательности
rend()	указывает на элемент, следующий за последним в обратной последовательности

Типы STL

Идентификатор типа	Описание
value_type	тип элемента
allocator_type	тип распределителя памяти
size_type	тип индексов, счетчика элементов и т.д.
iterator	ведёт себя как **value_type ***
reverse_iterator	просматривает контейнер в обратном порядке
reference	ведёт себя как **value_type &**
key_type	тип ключа (только для ассоциативных контейнеров)
key_compare	тип критерия сравнения (только для ассоциативных контейнеров)
mapped_type	тип отображенного значения

Методы доступа к элементам STL

Метод	Описание
front()	ссылка на первый элемент
back()	ссылка на последний элемент
operator [](i)	доступ по индексу без проверки
at(i)	доступ по индексу с проверкой

Методы для включения и исключения элементов

Метод	Описание
insert(p, x)	добавление **x** перед элементом, на который указывает **p**
insert(p, n, x)	добавление **n** копий **x** перед **p**
insert(p, first, last)	добавление элементов из диапазона **[first:last]** перед **p**
push_back(x)	добавление **x** в конец
push_front(x)	добавление нового первого элемента (только для списков и очередей с двумя концами)
pop_back()	удаление последнего элемента
pop_front()	удаление первого элемента (только для списков и очередей с двумя концами)
erase(p)	удаление элемента в позиции **p**
erase(first, last)	удаление элементов из диапазона **[first:last]**
clear()	удаление всех элементов

Другие операции STL

Операция	Описание
size()	количество элементов
empty()	определяет, пуст ли контейнер
capacity()	память, выделенная под вектор (только для векторов)
reserve(n)	выделяет память для контейнера под n элементов
resize(n)	изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)
swap(x)	обмен местами двух контейнеров
==, !=, <	операции сравнения
operator =(x)	контейнеру присваиваются элементы контейнера **x**
assign(n, x)	присваивание контейнеру **n** копий элементов **x** (не для ассоциативных контейнеров)
assign(first, last)	присваивание элементов из диапазона **[first:last]**
operator [](k)	доступ к элементу с ключом **k**
find(k)	находит элемент с ключом **k**
lower_bound(k)	находит первый элемент с ключом, меньшим **k**
upper_bound(k)	находит первый элемент с ключом, большим **k**
equal_range(k)	находит **lower_bound** (нижнюю границу) и **upper_bound** (верхнюю границу) элементов с ключом **k**