

Структурированные (составные) типы данных

В языках C, C++

- Массивы, C-строки (C strings)
- Пользовательские типы данных
 - ✓ Структура (`struct`) – набор переменных сгруппированных под одним именем, `class` в C++
 - ✓ Битовое поле (`bit field`) – структурированный доступ к отдельным битам
 - ✓ Объединение (`union`) – структура, позволяющая «интерпретировать» один участок памяти для нескольких типов переменных
 - ✓ Перечисление (`enum`) – список целых констант, `class enum` в C++11
- Структуры данных в STL C++

Массивы

Массив это простейшая вид **структурированных данных**, набор переменных одного типа; вектор – одномерный массив, матрица – двумерный массив

Инициализация:

```
int a[3];           // a[i] неопределенны
int b[3] = {2,3,4}; // b[0]=2; b[1]=3; b[2]=4
int c[] = {5,6,7};  // размерность задана списком
int d[5] = {5,6,7}; // d[3],d[4] неопределенны
static int e[10];    // e[i] инициализуются нулями из-за static
```


Доступ к элементам с помощью индекса:

```
a[0] = -1;          // первый элемент массива - индекс=0
a[1] = b[2];        // b[2] - последний элемент массива b[3]
for(i=0; i<100; i++) a[i]=0; // Error: нарушение границ массива
```

Адреса, указатели и массивы

Схема организации памяти

content	address	variable
0000	0008	← b
	0007	
0002	0006	← c
	0005	
	0004	
	0003	
1101	0002	← a
	0001	



Пример с указателем

```
// variables a and b
```

```
int a = 0x1101;
```

```
int b = 0;
```

```
// c is the pointer to an int
```

```
// assign address of 'a' to 'c'
```

```
int* c = &a;
```

```
// to dereference (косвенный доступ)
```

```
b = *c;    // now 'b' is 0x1101
```

```
*c = 0;    // now 'a' is 0
```

```
c = NULL;  // now 'c' points nowhere
```

Массив хранится в памяти непрерывно в порядке индексации:

	char a[3];			int b[3];		
Элемент	a[0]	a[1]	a[2]	b[0]	b[1]	b[2]
Адрес	1001	1002	1003	2000	2004	2008

Доступ к элементам массива char a[...] с помощью указателей

```
char* p1 = &a[0]; // указатель на нулевой элемент: p1=1001
char* p = a;      // то же самое: p = 1001
p++;              // p=1002 указывает на элемент a[1]
*(p+1) = 3;       // изменяем: a[2] = 3;
*p = 2;           // изменяем: a[1] = 2;
```

Доступ к элементам массива int b[...] с помощью указателей

```
int* p1 = &b[0]; // указатель на нулевой элемент: p1=2000
int* p = b;      // то же самое: p=2000
p++;             // p=2004 указывает на элемент b[1]
*(p+1) = 3;      // изменяем b[2] = 3;
*p = 2;          // изменяем b[1] = 2;
```

Полезные сведения о массивах и указателях

- Обращение к элементу массива возможно двумя эквивалентными способами: $p[i] \Leftrightarrow *(p+i)$

✓ имя массива указывает на нулевой элемент массива:

$\&p[i] \Leftrightarrow p+i; \Rightarrow \&p[0] \Leftrightarrow p;$

✗ допустимо, но уродливо: $\underbrace{(1+p)}[i] \Leftrightarrow *(1+p+i) \Leftrightarrow p[i+1];$
 $\underbrace{2[p]} \Leftrightarrow *(2+p) \Leftrightarrow p[2];$

- Имя массива **не является** указателем

- нельзя изменять
- всегда указывает на реальную память

- Объём памяти занимаемой массивом:

количество байтов = `sizeof(базовый тип)` * длина массива

вычисление длины массива

```
double c[] = {1.5, 2.6, 3.7};  
int nc = sizeof(c)/sizeof(c[0]); // nc = 3
```

Строки (C-strings)

- Строка – массив символов с завершающим нуль-символом

```
char str1[] = {'S','t','r','i','n','g',0};
```

```
char str2[] = "String"; // запись с двойными кавычками
```

- Для C-strings часто используют конструкцию:

```
const char* str3 = "String"; // сравните str3 с str2
```

👉 обратите внимание:

'\0' (a zero) – символ конца строки

'\n' (a newline) – может быть частью C-string

Пример: нахождение длины строки

```
const char* funny_str = "Fanny\n String";
```

```
int len_str = 0;
```

```
for( ; funny_str[len_str]; len_str++);
```

```
printf("length is %d\n",len_str); // length is 13
```

Библиотечные функции обработки строк

```
#include <string.h>
```

<code>strlen(s1)</code>	Возвращает длину строки <code>s1</code>
<code>strcpy(s1,s2)</code>	Копирование <code>s2</code> в <code>s1</code>
<code>strcat(s1,s2)</code>	Присоединение (concatenation) <code>s2</code> в конец <code>s1</code>
<code>strcmp(s1,s2)</code>	Лексикографическое сравнение: возвращает <code>0</code> если <code>s1</code> совпадает с <code>s2</code> , отрицательное значение если <code>s1<s2</code> и положительное если <code>s1>s2</code>
<code>strchr(s1,ch)</code>	Возвращает указатель на первое вхождение символа <code>ch</code> в строку <code>s1</code>

👉 `s1` и `s2` – указатели на C-strings

👉 `s1` в `strcpy()` и `strcat()` должен иметь достаточную длину

👉 `strcmp()` возвращает ноль (**FALSE**), если строки совпадают:

```
if( !strcmp(s1,s2) ) printf("s1 and s2 are equal\n");
```

Пример: добавить к имени файла "_new" перед точкой

```
char name[256]; // max. filename length
printf(" Type a file name: ");
scanf("%s",name); // %s for C-string
size_t len_name=strlen(name);
char* dot = strchr(name,'.'); // position of '.'
if( !dot ) dot = name+len_name; // '.' absent in name
char new_name[len_name+4+1]; // dynamic allocation C99
strcpy(new_name,name);
strcpy(new_name + (dot - name), "_new");
strcat(new_name,dot);
printf(" new_name = %s\n",new_name); // print result
```

тестирование

```
Type a file name: test.txt
new_name = test_new.txt
Type a file name: test
new_name = test_new
```


Структура (struct)

Объявление структуры

```
struct element {  
    char name[50]; // latin name  
    int number;    // Z number  
    double A;      // atomic weight  
};
```

- имя структуры (tag name) — `element`
- имена членов структуры (members name) — `name`, `number`, `A`

👉 никакая переменная не создается, а определяется новый тип данных:
`struct element`

👉 Члены структуры хранятся в памяти «не плотно»:

`sizeof(struct element) = 64`

`sizeof(char[50]) + sizeof(int) + sizeof(double) = 62`

Создание структурных переменных

```
struct element dump; // Неинициализированная переменная
```

```
struct element H = {"Hydrogenium",1,1.00794}; // Инициализация
```

```
// Массивы структур
```

```
struct element elements[5];
```

```
struct element halogens[] = {  
    {"Fluorum",      9,      18.9984},  
    {"Clorum",       17,     35.4527},  
};
```

```
// Указатели на структуру
```

```
struct element * ptr = NULL; // нулевой указатель
```

```
struct element * ptrH = &H;
```

Инициализация по указанию (designated initializer)

- В структурах:

```
struct element He = {.name="Helium",.number=2,.A=4.0026};  
struct element Li = {.number=3,.A=6.941,.name="Lithium"};
```

- В массивах:

```
int ar[] = { [1]=1, [9]=1 }; // ar[] = 0 1 0 0 0 0 0 0 0 1
```

👉 неуказанные элементы получают значение ноль

Обратите внимание

- 👉 До C++20 нет инициализации по указанию, используйте конструкторы
- 👉 Начиная с C++20 имеется, **но с большими ограничениями**

Доступ к членам структуры

оператор точка (.) 📌 слева от точки всегда стоит объект

```
struct element dump;  
strcpy(dump.name,"Test");  
dump.number = -1;  
dump.A = -1.;  
printf("atomic weight of element %s (%d) is %f\n",  
       dump.name,dump.number,dump.A);  
atomic weight of element Test (-1) is -1.000000
```

оператор стрелка (->) 📌 слева от стрелки всегда стоит указатель

```
struct element* pd = &dump; // pointer to dump  
strcpy(pd->name,"Test ptr");  
pd->number = 12;  
pd->A = 12.;  
printf("atomic weight of element «%s» (%d) is %f\n",  
       pd->name,pd->number,pd->A);  
atomic weight of element «Test ptr» (12) is 12.000000
```

👉 операторы точка и стрелка должны применяться к правильным типам:

```
dump->A = 1;           // ERROR
```

```
pd.A = 12;             // ERROR
```

👉 приоритет оператора точка выше чем у оператора звездочка:

```
(*pd).A = 12.;        // скобки обязательны
```

Копирование структур (оператор =)

- ✓ Это единственная операция в С применяемая к структуре целиком: вся структура копируется в другую **того же типа**

Пример

```
dump = H;  
printf("atomic weight of element %s (%d) is %f\n",  
       dump.name, dump.number, dump.A);  
atomic weight of element Hydrogenium (1) is 1.007940
```

Структуры с указателями на себя


Структура может содержать

- Переменные, массивы, структуры другого типа ...
- Указатели на структуру возможно того же типа

```
struct element {  
    char name[50];           // latin name  
    int number;              // = Z  
    double A;                // atomic weight  
    struct element* prev;    // pointer on previous element in table  
};
```

```
struct element H = {"Hydrogenium",1,1.00794,NULL};  
struct element He = {"Helium",2,4.0026,&H};  
struct element* ptr = &He;  
printf("number of previous element is %i\n",ptr->prev->number); // 1
```

Оператор typedef

 **typedef** определяет новое имя (синоним) для уже известного типа данных

```
typedef int Length;           // Length синоним int
Length len = 10;
Length *ptr = &len;

typedef char* Str;            // Str синоним 'char*'
Str line = "I am string\n";

typedef struct element Element; // Element синоним 'struct element'
struct element {
    ...
    Element* prev; /* pointer on previous element */
};
Element tmp = H;
```


Перечисления (enum)

Перечисление это список именованных целых констант:

```
enum tag_name {list of constants} variables;
```

- Значения констант должны быть внутри диапазона `int`
- Любому элементу можно указать желаемое значение
- Если явное значение отсутствует, то:
 - 👉 первый элемент перечисления имеет значение 0
 - 👉 следующий элемент на единицу больше чем предыдущий

Декларация и использование перечислений

```
enum boolean {FALSE, TRUE};  
enum boolean OK = TRUE;    // OK is a variable  
enum months {JAN=1,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};  
enum months winter = DEC;  // winter is a variable
```

```
enum color { WHITE=0, BLACK, RED, GREEN, BLUE, NAVY=4 };

void f(enum color Color) {
    switch(Color) {
        case WHITE: printf("White\n");      break;
        case BLACK: printf("Black\n");      break;
        case BLUE:  printf("Blue\n");       break;
        default:    printf("Color# %i\n", Color);
    }
}

enum color A = BLACK;
f(A);          // Black
f(NAVY);       // Blue
// 🖱️ нет проверки правильности типа
f(32);         // Color# 32
```

Введение в абстрактные типы данных

Структура данных в программировании

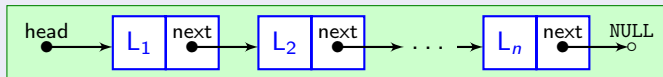
- Структура данных (data structure): специальный способ хранения данных удобный для построения эффективных алгоритмов
- Абстрактные типы данных: математические модели структур данных

Некоторые абстрактные типы данных

- Список (List)
- Стек (Stack)
- Очередь (Queue)
- Дерево (Tree)

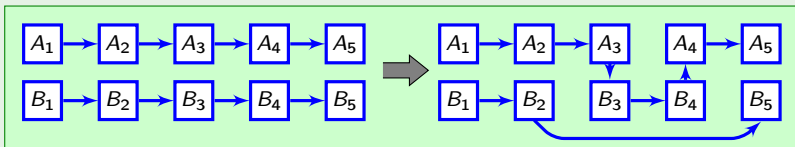
Список (List)

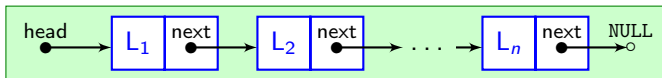
Принцип организации односвязного списка



- Последовательность узлов (*nodes*) списка в которых содержится:
 - ✓ данные: L_i
 - ✓ указатель на следующий узел: *next*
- имеется указатель на самый первый элемент списка: *head*


👉 основное свойство: легко добавлять и удалять элементы





Функциональная спецификация: ожидаемый набор функций

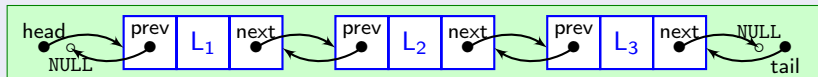
- 1 Создание \rightarrow функция создающая пустой список
- 2 Первый элемент \rightarrow возвращает указатель на первый элемент (или **NULL**)
- 3 Следующий элемент \rightarrow возвращает указатель на элемент за L_i
- 4 Вставка элемента \rightarrow вставляет новый элемент после L_i
- 5 Удаление элемента \rightarrow удаляет элемент после L_i

 Функция для числа элементов имеет сложность $O(n)$

Python3: список (list) объявлен как массив указателей на PyObjects

Другие разновидности списков

Двусвязный список (Doubly linked list)

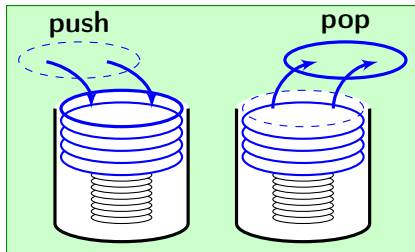


- + в каждом узле имеется ссылка на предыдущий элемент: **prev**
- + два указателя на первый и последний элементы: **head** и **tail**

Кольцевой список (Circularly linked list)

- + **next** в последнем элементе списка указывает на первый элемент
- + **prev** в первом элементе двусвязного списка указывает на последний

Стек (Stack)



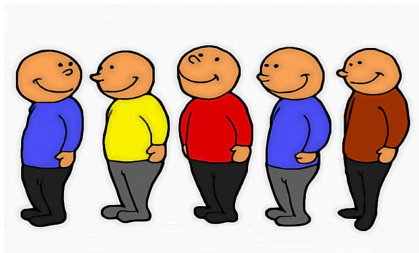
Принцип организации

- Последним пришел, первым вышел (last-in, first-out: LIFO)
- Доступ, извлечение только последнего элемента

Функциональная спецификация

- 1 Создание стека → функция создающая пустой стек
- 2 Проверка → является ли стек пустым
- 3 Добавление элемента → сохранение элемента в стеке: **push**
- 4 Извлечение элемента → новый стек без верхнего элемента: **pop**

Очередь (Queue)



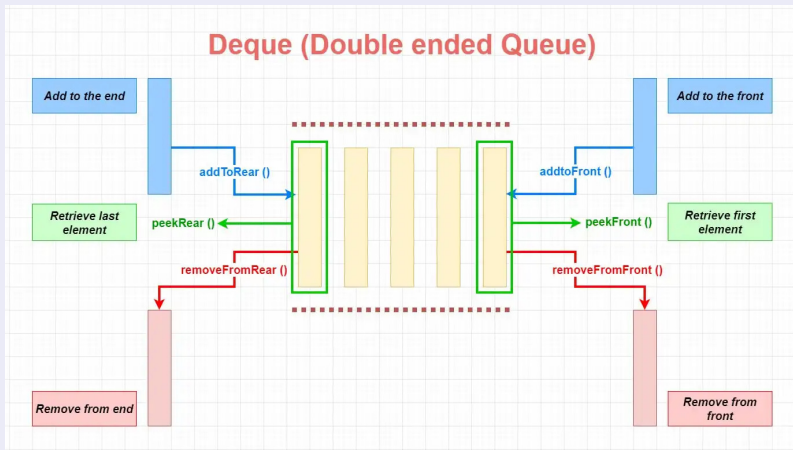
Принцип организации

- Первым пришел – первым вышел (first-in, first-out: FIFO)
- Доступ, извлечение только первого элемента

Функциональная спецификация

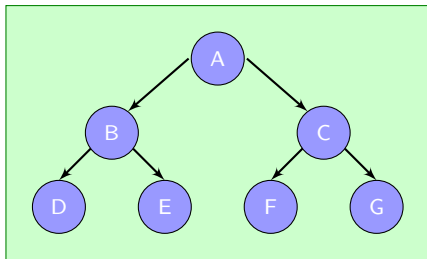
- 1 Создание → функция создающая пустую очередь
- 2 Проверка → является ли очередь пустой
- 3 Добавление элемента → сохранение элемента в очередь
- 4 Извлечение элемента → получение первого, самого давнего, элемента

Дек (Double-Ended Queue): двухсторонняя очередь



- Обобщает концепции и стека и очереди
- Элементы можно добавлять и удалять как в начало, так и в конец

Дерево (Tree)



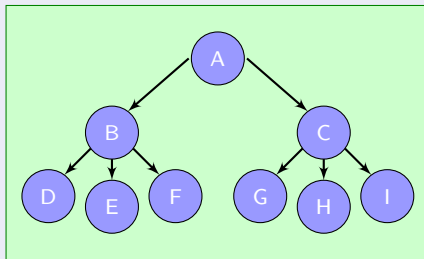
Основное свойство

👉 Дерево – иерархический набор узлов (**node**), связанных ребрами (**edge**), организованный так, что имеется единственный путь (**path**) от одного узла к другому

Терминология

- **A** – корень (**root node**)
- Соединение двух узлов – ребро (**edge**); для N узлов всего $N - 1$ ребро
- Путь – последовательность узлов и ребер между узлами: **path(AD) = A-B-D**, длина пути – полное число узлов в пути (**3**)
- Глубина дерева (**Depth of Tree**) – самый длинный путь из корня

Дерево общего вида



Терминология

- A,B,C – родительские (**parent**) узлы; B и C – родственники (**siblings**) и дети (**childes**) узла A
- Бинарное дерево – число детей любого узла не больше двух
- узлы D,E,F,G,H,I без детей – листья (**leaf**) или терминальные (**terminal**) узлы
- Каждый дочерний узел формирует поддереву (**sub-tree**)

Дополнительные слайды

Двухмерные массивы

Определение и инициализация

```
double a[5][8]; /* 5 строк по 8 элементов (8 столбцов) */
int b[2][3] = { {1,2,3},
               {4,5,6} }; /* инициализация */
int c[][4] = { {1,2,3,4}, /* первую размерность можно */
              {4,5,6,7} }; /* не указывать */
int d[][3] = { 1,2,3,5,6 }; /* допустимо, но уродливо */
```

- Адрес элемента: $a[i][j] = *(a + i * (\text{max value of } j) + j)$

```
_____ fill 2D-matrix _____
int num[3][5];
for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 5; j++) {
        num[i][j] = i+j+1;
    }
}
```

Указатели и двумерные массивы

Указатель на элемент массива

```
int a[5][8];  
int* p1 = &a[1][0]; /* указатель на начало первой строки */  
int* p2 = a[1];      /* то же самое: p2 == p1                */  
p1++;               /* переходим к следующему элементу    */  
p2 += 8;            /* перепрыгиваем на размер строки */
```

Указатели на строки

```
int (*pa1)[8] = &a[0]; /* указатель на всю нулевую строку */  
/* (на 1-d массив из 8 int) */  
int (*pa2)[8] = a;     /* то же самое: pa2 == pa1        */  
pa2++;                /* переходим к следующей строке: */  
/* теперь pa2 == &a[1]      */
```

Имя двумерного массива — указателей на нулевую строку!

Многомерные массивы и массивы указателей

Массив из массивов меньшей размерности: `int mm[5][8][3][4][3]`

☞ адрес одного элемента массива:

`mm[i][j][k][l][m] = *(mm + i*(8*3*4*3) + j*(3*4*3) + k*(3*4) + l*3 + m`

Массивы указателей – альтернатива многомерным массивам

```
_____ print text line by line _____  
void print_text(const char * text[]) {  
    for(int i = 0; text[i] != NULL; i++) printf("%s",text[i]);  
}
```

```
// s - array of pointers on C-strings  
char * s[] = {"Hello, world\n","Goodbye world\n",NULL};  
print_text(s); // Hello, world  
               // Goodbye world
```

Битовые поля (bit field)


Пример из документации

CSR (статусный регистр) доступен по адресу 0xD8000. Для генерации цикла КАМАК номер функции (0...31) нужно занести в биты 1-5 статусного регистра, а в бит 0 занести 1. После завершения цикла бит 0 будет очищен контроллером.

Декларации структуры с битовыми полями для этого примера

```
struct CSR {  
    unsigned busy : 1; /* Бит, запускающий цикл */  
    unsigned f    : 5; /* 5 битов под функцию */  
    int unused    : 2; /* Два неиспользуемых старших бита */  
};
```

Пояснение

 Заданы переменные **битовой** длины: **тип имя : длина_в_битах;**
для доступа к отдельным битам

Пример реализации через структуру с битовыми полями

```
volatile struct CSR * mycsr = (struct CSR*)0xD8000; /* address */
mycsr->f = 8; /* функция 8 */
mycsr->busy = 1; /* взвести бит 0 */
while ( mycsr->busy ); /* ждать когда контроллер очистит бит 0 */
```

Реализация через битовые операции

```
volatile char * csr = (char*)0xD8000; /* address */
*csr = (8<<1)|1; /* функция 8 и взвести бит 0 */
while( (*csr & 1) != 0 ); /* ждать когда контроллер очистит бит 0 */
```

Особенности использования битовых полей

- ☞ Поля могут быть только целочисленного типа
- ☞ Переменные битовых полей не имеют адресов
- ☞ Нет массивов битовых полей
- ☞ **Аппаратно зависимое расположение полей: на одних машинах слева направо, на других справа налево**

Объединения (union)

- Объединение позволяет хранить в одной области памяти различные типы данных

Пример декларация объединения + переменной `my_data`

```
union u_tag {  
    char  cval;  
    int   ival;  
    float fval;  
} my_data;
```

🔗 `sizeof(union u_tag) = 4` – размер определяется наибольшим членом

🔗 В C89 инициализировать можно только первый член:

```
union u_tag data2 = {'c'};
```

🔗 В C99 возможна инициализация по указанию (**designated initializer**):

```
union u_tag test = {.ival=3};
```

Пример: доступ к отдельным полям объединения

```
my_data.cval = 'a';          /* ival and fval not valid */  
printf("Here is u_tag: cval= %c ival= 0x%8x fval= %11.3f\n",  
      my_data.cval, my_data.ival, my_data.fval);
```

```
Result> Here is u_tag: cval= a ival= 0xc3109661 fval=      -144.587
```

```
my_data.ival = 65;           /* cval and fval not valid */  
printf("Here is u_tag: cval= %c ival= 0x%8x fval= %11.3f\n",  
      my_data.cval, my_data.ival, my_data.fval);
```

```
Result> Here is u_tag: cval= A ival= 0x      41 fval=      0.000
```

```
my_data.fval = 1.0;          /* cval and ival not valid */  
printf("Here is u_tag: cval= %c ival= 0x%8x fval= %11.3f\n",  
      my_data.cval, my_data.ival, my_data.fval);
```

```
Result> Here is u_tag: cval=      ival= 0x3f800000 fval=      1.000
```