

Динамические структуры данных

Лектор: Артамонов Юрий Николаевич

Университет "Дубна"
филиал Котельники

- 1 Динамические массивы
- 2 Структуры данных
- 3 Динамические структуры данных

Очень часто возникают задачи обработки массивов данных, размерность которых заранее неизвестна. В этом случае возможно использование одного из двух подходов:

- выделение памяти под статический массив, содержащий максимально возможное число элементов, однако в этом случае память расходуется нерационально;
- динамическое выделение памяти для хранения массива данных.

В последнем случае мы приходим к понятию динамического массива. Таким образом, динамический массив - это массив, размеры которого можно задавать в ходе выполнения программы и при необходимости их изменять. Для создания таких массивов используются указатели. Такой указатель ссылается на начальный адрес динамического массива, для которого память выделяется специальными функциями динамического выделения памяти.

Стандартные функции динамического выделения памяти

Функции динамического выделения памяти находят в оперативной памяти непрерывный участок требуемой длины и возвращают начальный адрес этого участка.

В языке C существует 2 разновидности таких функций:

```
void* malloc(Размер_Массива_в_Байтах);
```

```
void* calloc(Число_элементов, Размер_элемента_в_байтах);
```

Поскольку обе представленные функции в качестве возвращаемого значения имеют указатель на пустой тип `void`, требуется явное приведение типа возвращаемого значения.

Для определения размера массива в байтах, используемого в качестве аргумента функции `malloc()` требуется количество элементов умножить на размер одного элемента. Для точного определения размера элемента в общем случае рекомендуется использование функции `sizeof(тип)`, которая определяет количество байт, занимаемое элементом указанного типа.

Стандартные функции динамического выделения памяти

(продолжение)

Память, динамически выделенная с использованием функций `calloc()`, `malloc()`, может быть освобождена с использованием функции `free(указатель)`;

«Правилом хорошего тона» в программировании является освобождение динамически выделенной памяти в случае отсутствия ее дальнейшего использования. Однако, если динамически выделенная память не освобождается явным образом, она будет освобождена по завершении выполнения программы.

Рассмотрим пример создания одномерного динамического массива.

Пример создания и использования одномерного динамического массива

```
#include <stdio.h>
main()
{
    int *Ar1, *Ar2, n,m;
    printf("Введите размерность первого массива n = ");
    scanf("%d",&n);
    Ar1 = (int *)malloc(n*sizeof(int));
    printf("Введите размерность второго массива m = ");
    scanf("%d",&m);
    Ar2 = (int *)calloc(m, sizeof(int));
    int i;
    for (i=0; i<n; i++)
    {
        printf("A[ %d] = ", i);
        scanf("%d", &Ar1[i]); //Можно обращаться, как к обычному массиву
    }
    for (i=0;i<n;i++) printf("A[ %d]=%d\n", i, *(Ar1+i)); //Можно
    использовать индексацию по указателям
```

Пример создания и использования одномерного динамического массива (продолжение)

```
for (i=0; i<m; i++)
{
    printf("B[ %d] = ", i);
    scanf(" %d", Ar2+i); //А здесь используется индексация по указателю
}
for (i=0; i<m; i++) printf("B[ %d]= %d\n", i, Ar2[i]); //И
наоборот, здесь мы получаем элемент массива по индексу
free(Ar1);
free(Ar2);
return 0;
}
```

Иногда даже в ходе выполнения программы размер массива оказывается неизвестен, или его требуется изменить по результатам каких-либо вычислений. В этом случае потребуется перераспределение памяти.

Например, для увеличения размера массива на единицу необходимо выполнить следующие действия:

- Выделить блок памяти размерности $n+1$ (на 1 больше текущего размера массива)
- Скопировать все значения, хранящиеся в массиве во вновь выделенную область памяти
- Освободить память, выделенную ранее для хранения массива
- Переместить указатель начала массива на начало вновь выделенной области памяти
- Дополнить массив последним введенным значением

Перераспределение памяти (продолжение)

Все перечисленные выше действия (кроме последнего) выполняет функция

```
void* realloc (void* ptr, size);
```

`ptr` - указатель на блок ранее выделенной памяти функциями `malloc()`, `calloc()` или `realloc()` для перемещения в новое место. Если этот параметр равен `NULL`, то выделяется новый блок, и функция возвращает на него указатель.

`size` - новый размер, в байтах, выделяемого блока памяти. Если `size = 0`, ранее выделенная память освобождается и функция возвращает нулевой указатель, `ptr` устанавливается в `NULL`.

Размер блока памяти, на который ссылается параметр `ptr` изменяется на `size` байтов. Блок памяти может уменьшаться или увеличиваться в размере. Содержимое блока памяти сохраняется даже если новый блок имеет меньший размер, чем старый. Но отбрасываются те данные, которые выходят за рамки нового блока. Если новый блок памяти больше старого, то содержимое вновь выделенной памяти будет неопределенным.

Пример на перераспределение памяти

```
#include <stdio.h>
main()
{
    int *Ar = NULL, i=0, j;
    do {
        printf("a[ %d]=", i);
        Ar = (int*)realloc(Ar, ++i*sizeof(int));
        scanf(" %d", &Ar[i-1]);
        printf("Вы хотите продолжить (y/n)?\n");
        getchar();
    } while (getchar() == 'y');
    for (j = 0; j < i; j++) printf("A[ %d]= %d\n", j, Ar[j]);
    free(Ar);
    return 0;
}
```

Задание 8.1

Реализовать алгоритм решета Эратосфена получения массива всех простых чисел. Вначале массив A содержит все числа от 1 до n (n требуется ввести в программу). Затем все составные числа заменяются нулями. После этого массив A требуется уменьшить и хранить в нем только простые числа.

Задание 8.2

Реализовать алгоритм пузырьковой сортировки массива длины n (n требуется ввести в программу). После этого требуется расширить массив на m элементов (m требуется ввести в программу), заполнить полученные новые элементы случайными числами и вновь отсортировать массив.

Задание 8.3

С помощью динамического массива создать структуру стека. Стек - это структура, в которой доступно две операции:

pop - извлечение элемента из вершины стека

push - добавление элемента в вершину стека.

Образно стек подобен стопке книг, лежащих на столе. За один раз можно взять только одну книгу с вершины этой стопки. Когда книга добавляется, она также кладется вверх стопки. Такой принцип добавления, извлечение элементов получил название LIFO (last input first output).

Задание 8.4

С помощью динамического массива создать структуру очереди. Очередь - это структура, в которой доступно две операции:

pop - извлечение элемента из начала очереди

push - добавление элемента в конец очереди

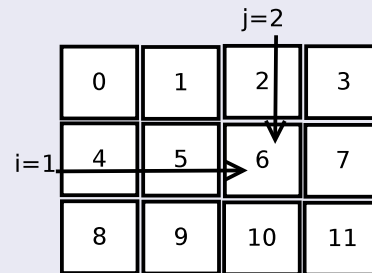
Образно очередь - это стандартная процедура оплата товара в кассе магазина. Кассир обслуживает клиента, стоящего первым в очереди. Последний клиент будет обслужен последним. Такой принцип добавления, извлечение элементов получил название FIFO (first input first output).

Двумерные динамические массивы

Ранее мы рассматривали статические двумерные массивы. Следует иметь в виду, что элементы любых массивов физически располагаются в памяти последовательно. Логический доступ к элементу i -й строки j -го столбца для двумерного массива $n \cdot m$ осуществляется как физический доступ к одномерному массиву с индексом, вычисляемым по формуле:

$$index = m \cdot i + j$$

Например, на рисунке количество строк $n = 3$, количество столбцов $m = 4$.



The diagram shows a 3x4 grid of cells containing numbers from 0 to 11 in row-major order. A horizontal arrow labeled 'i=1' points to the second row, and a vertical arrow labeled 'j=2' points to the third column. Both arrows intersect at the cell containing the number 6.

| | | | |
|---|---|----|----|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |

Если $i = 1, j = 2$, то имеем: $index = 4 \cdot 1 + 2 = 6$. Это можно использовать, чтобы физически организованный одномерный динамический массив представить двумерным динамическим массивом.

Двумерные динамические массивы (продолжение)

Тогда задать такой массив можно с помощью указателя следующим образом:

```
int *a; // указатель на массив
int i, j, n, m;
printf("Введите количество строк: "); scanf("%d", &n);
printf("Введите количество столбцов: "); scanf("%d", &m);
a = (int*)malloc(n*m * sizeof(int)); // Выделение памяти
for (i = 0; i<n; i++) // цикл по строкам
{
    for (j = 0; j<m; j++) // цикл по столбцам
    {
        printf("a[ %d][ %d] = ", i, j);
        scanf("%d", (a + i*m + j));
        printf(" %4d ", *(a + i*m + j));
    }
}
```


Двумерные динамические массивы можно задавать и другим образом, используя динамический массив указателей на динамические массивы строк. Для этого необходимо:

- выделить блок оперативной памяти под массив указателей;
- выделить блоки оперативной памяти под одномерные массивы, представляющие собой строки искомой матрицы;
- записать адреса строк в массив указателей.

Рассмотрим соответствующий пример.

Пример задания двумерного динамического массива с помощью массива указателей

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int i, j, n, m;
    int **a; // указатель на указатели на строки элементов
    printf("Введите количество строк: "); scanf("%d", &n);
    printf("Введите количество столбцов: "); scanf("%d", &m);
    a = (int**)malloc(n * sizeof(int*)); // Выделение памяти под указатели
    на строки
    for (i = 0; i < n; i++) // цикл по строкам
    {
        a[i] = (int*)malloc(m * sizeof(int)); // Выделение памяти под
    хранение строк
        for (j = 0; j < m; j++) // цикл по столбцам
        {
            printf("a[ %d][ %d] = ", i, j); scanf("%d", &a[i][j]);
        }
    }
}
```

Пример задания двумерного динамического массива с помощью массива указателей (продолжение)

```
// Вывод элементов массива
for (i = 0; i < n; i++) // цикл по строкам
{
    for (j = 0; j < m; j++) // цикл по столбцам
    {
        printf(" %5d ", a[i][j]); // 5 знакомест под элемент массива
    }
    printf("\n");
}
// Очистка памяти
for (i = 0; i < n; i++) // цикл по строкам
    free(a[i]); // освобождение памяти под строку
free(a);
return 0;
}
```

В принципе можно создавать массив указателей на строки разной длины. Такой двумерный массив называется свободным двумерным массивом.

Задание 8.5

С помощью двумерного динамического массива реализуйте алгоритм транспонирования матрицы размером $n \cdot m$. В итоге должна получаться матрица $m \cdot n$.

Задание 8.6

Реализуйте свободный двумерный массив, число элементов в строках ввести с клавиатуры.

Задание 8.7

Реализуйте трехмерный динамический массив (его можно задать, как указатель на указатели на указатели строк).

Структуры - это наборы (*агрегаты*) логически связанных переменных, объединенных под одним именем. В отличие от массивов, которые могут содержать элементы только одного типа, структуры могут состоять из переменных различных типов данных. Указатели на структуры могут служить для создания более сложных структур, таких как связанные списки, очереди, стеки и т.п. Чтобы задать структуру, необходимо использовать следующую форму:

```
struct имя
{
    тип1 имя1;
    тип2 имя2;
    ...
    тип_n имя_n;
}
```

Понятие структуры (продолжение)

Например, определим структуру сотрудника фирмы. У сотрудника нас будут интересовать: пол, возраст, должность

```
struct associate
{
    char sex;
    int years;
    char position[100];
};
```

Обратите внимание, каждое определение структуры должно заканчиваться точкой с запятой. Следует понимать, что такое определение не резервирует место в памяти для хранения элементов структуры. Оно просто создает новый тип данных, который можно использовать для объявления переменных этого типа.

Понятие структуры (продолжение)

Переменные структуры объявляются так же, как переменные других типов:

```
struct associate boss, ar_human[10], *PtrAss;
```

В представленном примере объявлена одна переменная boss типа associate, массив из десяти элементов типа associate, а также указатель на структуру типа associate.

Можно объявить переменные данной структуры и по-другому, поместив список соответствующих переменных между закрывающейся скобкой определения структуры и точкой с запятой, завершающей ее определение:

```
struct associate  
{  
    char sex;  
    int years;  
    char position[100];  
} boss, ar_human[10], *PtrAss;
```

Единственно допустимыми операциями над структурами являются:

- присваивание переменных-структур переменным того же типа;
- взятие адреса структуры с помощью &;
- обращение к элементам структуры;
- применение операции sizeof для определения размера структуры.

Структуры нельзя сравнивать, поскольку ее элементы не обязательно хранятся в последовательных байтах памяти. Структуры можно инициализировать, как и массивы, используя список инициализации. Например,

```
struct associate boss = { 'М', 43, "Генеральный директор" };
```

Если инициализаторов в списке меньше, чем элементов в структуре, остальным элементам автоматически присваивается значение 0 (или NULL, если элемент - указатель).

Доступ к элементам структуры

Для обращения к элементам структур используются две операции: *операция элемента структуры - точка (.)*, *операция указателя структуры - стрелка (->)*. Для того, чтобы обратиться к элементу структуры после имени переменной нужно поставить точку, после которой следует указать имя соответствующего поля. Например,

```
printf(" %s", boss.position);
```

Операция указателя структуры, состоящая из знака минус (-) и знака больше (>) без пробела между ними, обращается к элементу через указатель структуры. Например,

```
printf(" %s", PtrAss->position);
```

Выражение `PtrAss->position` эквивалентно в этом случае `(*PtrAss).position`.

Рассмотрим соответствующий пример.

Пример работы со структурами

```
#include <stdio.h>
struct associate
{
    char sex;
    int years;
};
void output_struct_1( struct associate);
void output_struct_2( struct associate *);
void output_struct_1( struct associate elem)
{
    printf("Пол сотрудника: %c\n", elem.sex);
    printf("Возраст сотрудника: %d\n", elem.years);
}
void output_struct_2( struct associate *elem)
{
    printf("Пол сотрудника: %c\n", elem->sex);
    printf("Возраст сотрудника: %d\n", elem->years);
}
```

Пример работы со структурами (продолжение)

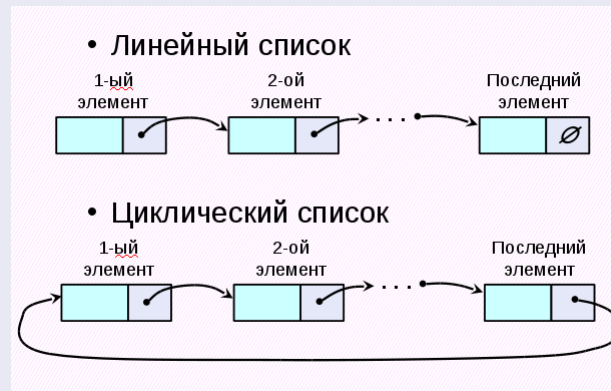
```
int main()
{
    struct associate boss = { 'M', 43 }, *PtrAss;
    output_struct_1(boss);
    output_struct_2(&boss);
    PtrAss = &boss;
    (*PtrAss).sex = 'W';
    PtrAss->years = 55;
    output_struct_1(*PtrAss);
    output_struct_2(PtrAss);
    return 0;
}
```

Структуры, ссылающиеся на себя

Кроме обычных структур, можно создавать структуры, ссылающиеся на себя. Такие структуры содержат в качестве элемента указатель, который ссылается на структуру того же типа. Например,

```
struct node
{
    int data;
    struct node *next;
};
```

Это позволяет создавать произвольные динамические структуры данных. В качестве примера рассмотрим структуру связанного списка:



Связанный список - это линейный набор ссылающихся на себя структур, называемых узлами, и объединенных указателем-связкой. Доступ к связанному списку обеспечивается указателем на первый узел списка. Доступ к следующим узлам производится через связывающий указатель текущего узла. Указатель последнего узла списка устанавливается в `NULL`, отмечая конец списка. Каждый узел создается по мере необходимости. Узел может содержать данные любого типа, в том числе и другие структуры. Списки удобны, когда заранее неизвестно, сколько элементов данных будет содержать структура. Связанные списки являются динамическими, поэтому длина списка при необходимости может увеличиваться или уменьшаться. В отличие от массивов, элементы списка располагаются последовательно только логически. Физически в памяти они могут располагаться любым образом.

Пример определения связанного списка

```
#include <stdio.h>
#include <stdlib.h>
struct lst{
    int data;
    struct lst *next;
};

int main()
{
    struct lst *a,*b,*c, d;

    a = (struct lst *)malloc(sizeof(struct lst));
    b = (struct lst *)malloc(sizeof(struct lst));
    c = (struct lst *)malloc(sizeof(struct lst));
    printf("a.data = ");scanf("%d",&(a->data));
    printf("b.data = ");scanf("%d",&(b->data));
    printf("c.data = ");scanf("%d",&(c->data));
```


Пример определения связанного списка

```
a->next = b;  
b->next = c;  
c->next = &d;  
d.data= 666;  
d.next = NULL;
```

```
struct lst *p;  
p=a;
```

```
while (p != NULL)  
{  
    printf(" %d\n",p->data);  
    p=p->next;  
}
```

```
free(a); free(b); free(c);  
return 0;
```

```
}
```

Пример определения стека с помощью связанного списка

Основная функция

```
#include<stdio.h>
#include<stdlib.h>

struct stack{
    int data;
    struct stack *next;
};

int pop(struct stack **);
void push(struct stack **, int);

int main()
{
    struct stack *top_stack=NULL, *current = NULL;
    int key, a;
    do{
        printf("1 — Добавить в стек, 2 — удалить из стека; 3 — выход\n");
        scanf("%d",&key);
```

Пример определения стека с помощью связанного списка

(продолжение)

Основная функция (продолжение)

```
switch (key){
case 1:
    printf("a = "); scanf("%d",&a);
    push(&top_stack, a);
    break;
    case 2:
        printf("Извлеченный элемент: %d\n", pop(&top_stack));
        break;
case 3:
    break;
default:
    printf("Ошибка ввода\n");}}
while (key != 3);
```

Пример определения стека с помощью связанного списка (продолжение)

Основная функция (продолжение)

```
while (top_stack != NULL)
{
    current = top_stack->next;
    free(top_stack);
    top_stack = current;
}
return 0;
}
```

Пример определения стека с помощью связанного списка

(продолжение)

Реализация pop

```
int pop(struct stack **p)
{
    if ((*p) != NULL)
    {
        int a = (*p)->data;
        *p=(*p)->next;
        return a;
    }
    else
        return 0;
}
```

Пример определения стека с помощью связанного списка (продолжение)

Реализация push

```
void push(struct stack **p, int element)
{
    struct stack *current;
    current = (struct stack *) malloc(sizeof(struct stack));
    current->data = element;
    current->next = *p;
    *p = current;
}
```

Задание 8.8

Реализуйте структуру динамического линейного списка, каждый элемент которого имеет три поля: информационное поле (`data`), ссылка на предыдущий элемент (`pred`), ссылка на следующий элемент (`nex`t). Если нет предыдущего (это первый элемент), то `pred=NULL`, если нет следующего (это последний элемент), то `nex`t=NULL. Реализация должна включать в себя три функции: 1. добавление элемента в начало; 2. добавление элемента в конец; 3. вывод всех элементов списка.

Задание 8.9

Используя результат задачи 8.8, реализуйте функцию добавления элемента на заданную позицию n .

Задание 8.10

Используя результат задачи 8.8, реализуйте функцию удаления элемента с начала списка.

Задание 8.11

Используя результат задачи 8.8, реализуйте функцию удаления элемента с конца списка.

Задание 8.12

Используя результат задачи 8.8, реализуйте функцию удаления элемента с заданного места в списке.

Задание 8.13

Используя результат задачи 8.8, реализуйте функцию нахождения элемента в списке.

Задание 8.14

Используя результат задачи 8.8, реализуйте функцию сортировки элементов списка.