Министерство образования Республики Беларусь

ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

**Методические указания к лабораторной работе № 2\_3  
 по курсу «Основы алгоритмизации   
и программирования»**

«Динамическое распределение памяти»

Преподаватель: Войтехович   
Агния Витольдовна

Составитель: Войтехович   
Агния Витольдовна

Полоцк, 2017

# **ЦЕЛЬ РАБОТЫ**

Изучить принципы работы с памятью и функции для работы с ней. Научиться разрабатывать программы с использованием динамических структур на смежной памяти.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

# 1 Понятие динамической памяти

Память, которую использует программа делится на три вида:

1. Статическая память (static memory):

* хранит глобальные переменные и константы;
* размер определяется при компиляции.

1. Стек (stack):

* хранит локальные переменные, аргументы функций и промежуточные значения вычислений;
* размер определяется при запуске программы (обычно выделяется   
  4 Мб).

1. Куча (heap):

* динамически распределяемая память;
* ОС выделяет память по частям (по мере необходимости).

**Динамическая память** - это область (блок) памяти выделенный для нужд программы в процессе работы программы (а не заранее).

Динамически распределяемую память следует использовать в случае если мы заранее (на момент написания программы) не знаем сколько памяти нам понадобится (например, размер массива зависит от того, что введет пользователь во время работы программы) и при работе с большими объемами данных (например, массив из 1 000 000 int`ов не поместится на стеке).

Основными двумя действиями над динамической памятью являются: выделение и освобождение. В языке С++ функции для осуществления этих действий описаны в библиотеке **cstdlib**.

# 2 Стандартные функции динамического выделения памяти

Для использования функций динамического выделения памяти необходимо описать указатель, представляющий собой начальный адрес хранения элементов массива.

int \*p; // указатель на тип int

Начальный адрес статического массива определяется компилятором в момент его объявления и не может быть изменен.

Для динамического массива начальный адрес присваивается объявленному указателю на массив в процессе выполнения программы.

Основу системы динамического распределения в С++ составляют функции ***malloc()*** и ***free().*** Эти функции работают совместно. Функция ***malloc()*** выделяет память, а ***free()*** — освобождает ее. Это значит, что при каждом запросе функция ***malloc()*** выделяет требуемый участок свободной памяти, a ***free()*** освобождает его, то есть возвращает системе.

Кроме того, есть еще 2 «модернизированные» функции работы с памятью - ***calloc()*** и ***realloc()****.* Функция ***calloc()*** работает аналогично ***malloc()***, но отличается синтаксисом и тем, что выделенная память будет обнулена, а функция ***realloc()*** изменяет размер выделенной памяти (на которую указывает **ptr**, ранее полученный из вызова malloc, calloc или realloc).

В программу, использующую эти функции, должен быть включен заголовочный файл **<** **cstdlib>**.

Работа с памятью также практически невозможна без унарного оператора ***sizeof()***, возвращающего длину в байтах переменной или типа, помещенных в скобки. Рассмотрим эти функции более подробно.

## 2.1 Оператор sizeof

***sizeof*** - это унарный оператор, возвращающий длину в байтах переменной или типа, помещенных в скобки. Например:

float f;

cout << sizeof f << “ ”;

cout << sizeof(int) << endl;

выдает ***4 2***.

Помните, что для вычисления размера типа следует поместить имя типа в круглые скобки, а имена переменных можно писать и без скобок.

Использование **sizeof** помогает создавать переносимый код для тех случаев, когда код зависит от размера стандартных типов данных. Например, представим, что программе, работающей с базой данных, необходимо сохранять 6 целочисленных значений в записи. Для того, чтобы сделать эту программу переносимой, не следует предполагать, что размер целочисленного типа - 2 или 4 байта, следует самостоятельно определить настоящую длину, используя **sizeof**. Следующая подпрограмма может быть использована для сохранения записи в файле:

/\* Запись записи в файл \*/

void put\_rec(FILE \*fp, int rec[6])

{

int size, num;

size = sizeof(int) \* 6;

num = fwrite(rec, size, 1, fp) ;

if(num!=1) cout << "ошибка записи";

}

Самое главное в этом примере то, что ***put\_rec()*** будет компилироваться и корректно запускаться на любом компьютере, включая компьютеры, использующие 2- и 4-байтные целые.

## 2.2 Функция malloc()

Прототип функции **malloc()** следующий:

void \*malloc(size\_t количество\_байтов);

Здесь ***количество\_байтов*** — размер памяти, необходимой для размещения данных. (Тип ***size\_t*** определен в **<cstdlib>** как некоторый целый без знака.) Функция **malloc()** возвращает указатель типа **void \***, поэтому его можно присвоить указателю любого типа. При успешном выполнении **malloc()** возвращает указатель на первый байт непрерывного участка памяти, выделенного в динамически распределяемой области памяти. Если в динамически распределяемой области памяти недостаточно свободной памяти для выполнения запроса, то память не выделяется и **malloc()** возвращает нуль.

При выполнении следующего фрагмента программы выделяется непрерывный участок памяти объемом 1000 байтов:

char \*p;

p = malloc(1000); /\* выделение 1000 байтов \*/

После присвоения указатель **p** ссылается на первый из 1000 байтов выделенного участка памяти.

В следующем примере выделяется память для 50 целых. Для повышения мобильности (переносимости программы с одной машины на другую) используется оператор **sizeof**.

int \*p;

p = malloc(50\*sizeof(int));

Поскольку динамически распределяемая область памяти не бесконечна, при каждом размещении данных необходимо проверять, состоялось ли оно. Если **malloc()** не смогла по какой-либо причине выделить требуемый участок памяти, то она возвращает нуль. В следующем примере показано, как выполняется проверка успешности размещения:

p = malloc(100);

if(!p) {

cout << "Нехватка памяти." << endl;

exit(1);

}

Конечно, вместо выхода из программы **exit()** можно поставить какой-либо обработчик ошибки. Обязательным здесь можно назвать лишь требование не использовать указатель **р**, если он равен нулю.

## 2.3 Функция сalloc()

#include <cstdlib>

void \*calloc(size\_t num, size\_t size);

Функция **calloc()** выделяет память, размер которой равен значению выражения ***num \* size***, т.е. память, достаточную для размещения массива, содержащего **num** объектов размером **size**. Все биты распределенной памяти инициализируются нулями.

Функция **calloc()** возвращает указатель на первый байт выделенной области памяти. Если для удовлетворения запроса нет достаточного объема памяти, возвращается нулевой указатель. Перед попыткой использовать распределенную память важно проверить, что возвращаемое значение не равно нулю.

**ПРИМЕР**

Эта функция возвращает указатель на динамически распределенный блок памяти для массива из 100 чисел типа **float**:

#include <cstdlib >

#include <oistream>

float \*get\_mem(void)

{

float \*p;

p = calloc(100, sizeof(float));

if(!p) {

cout << "Ошибка при распределении памяти" << endl;

exit(1);

}

return p;

}

## 2.4 Функция realloc()

#include <cstdlib>

void \*realloc(void \*ptr, size\_t size);

Функция realloc() возвращает указатель на блок памяти указанного размера, в который копируется содержимое блока **ptr** (до size байтов). Причем допускается, чтобы новый и старый блоки начинались с одинакового адреса (т.е. указатель, возвращаемый функцией **realloc()**, может совпадать с указателем, переданным в параметре **ptr**).

Если указатель **ptr** нулевой, функция **realloc()** просто выделяет **size** байтов памяти и возвращает указатель на эту память. Если значение параметра **size** равно нулю, память, адресуемая параметром **ptr**, освобождается.

Если в динамически распределяемой области памяти нет достаточного объема свободной памяти для выделения **size** байтов, возвращается нулевой указатель, а исходный блок памяти остается неизменным.

**ПРИМЕР**

Эта программа сначала выделяет блок памяти для 17 символов, копирует в них строку "Это - 17 символов", а затем использует **realloc()** для увеличения размера блока до 18 символов, чтобы разместить в конце точку.

#include <cstdlib>

#include <iostream>

#include <cstring>

int main(void)

{

char \*p;

p = malloc(17);

if(!p) {

cout << "Ошибка при распределении памяти" << endl;

exit(1);

}

strcpy(p, "Это - 17 символов");

p = realloc(p, 18);

if(!p) {

cout << "Ошибка при распределении памяти" << endl;

exit(1);

}

strcat(p, ".");

printf(p);

free(p);

return 0;

}

## 2.5 Функция free()

#include <cstdlib >

void free(void \*ptr);

Функция **free()** возвращает в динамически распределяемую область памяти блок памяти, адресуемый указателем **рtr**, после чего эта память становится доступной для выделения в будущем.

В качестве входного параметра в **free()** нужно передать указатель, значение которого получено из функции **malloc**. Вызов **free** на указателях полученных не из **malloc** (например, ***free(p+10)***) приведет к неопределенному поведению. Это связанно с тем, что при выделении памяти при помощи **malloc** в ячейки перед той, на которую указывает возвращаемый функцией указатель операционная система записывает служебную информацию. При вызове ***free(p+10)*** информация находящаяся перед ячейкой ***(p+10)*** будет трактоваться как служебная.

**ПРИМЕР**

Эта программа распределяет блок памяти для вводимых пользователем строк, а затем освобождает блок памяти:

#include <cstdlib>

#include <iostream>

int main(void)

{

char \*str[100];

int i;

for(i=0; i<100; i++) {

if((str[i] = malloc(128))==NULL) {

cout << "Ошибка при распределении памяти" << endl;

exit(1);

}

gets(str[i]);

}

/\* Освобождение блока памяти \*/

for(i=0; i<100; i++) free(str[i]);

return 0;

}

# 3 Динамическое выделение памяти для одномерных массивов

Форма обращения к элементам массива с помощью указателей имеет следующий вид:

int a[10], \*p; // описываем статический массив и указатель

int b;

p = a; // присваиваем указателю начальный адрес массива

... // ввод элементов массива

b = \*p; // b = a[0];

b = \*(p+i) // b = a[i];

**ПРИМЕР**

Организация динамического одномерного массива и ввод его элементов.

#include <cstdlib>

#include <iostream>

#include <clocale>

using namespace sdt;

int main() {

setlocale(LC\_ALL, “”);

int \*a; // указатель на массив

int i, n;

cout << "Введите размер массива: ";

cin >> n;

// Выделение памяти

a = (int\*) malloc(n\*sizeof(int));

// Ввод элементов массива

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

{

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

cin >> a[i];

}

// Вывод элементов массива

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

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

free(a);

getchar();

return 0;

}

Результат выполнения программы представлен на рисунке 3.1.

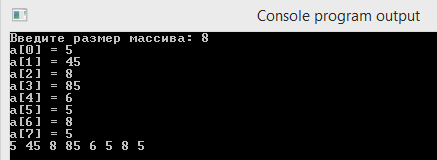


Рисунок 3.1 – Результат работы программы  
 с динамическим выделением памяти для массива

# 4 Динамическое выделение памяти для двумерных массивов

Пусть требуется разместить в динамической памяти матрицу, содержащую **n** строк и **m** столбцов. Двумерная матрица будет располагаться в оперативной памяти в форме ленты, состоящей из элементов строк. При этом индекс любого элемента двумерной матрицы можно получить по формуле

index = i\*m+j;

где **i** - номер текущей строки; **j** - номер текущего столбца.

Рассмотрим матрицу 3x4 (смотрите рисунок 4.1).

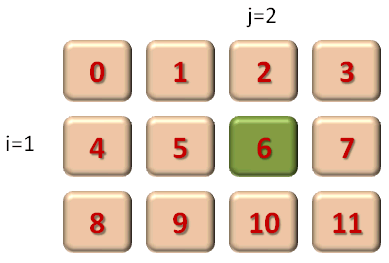


Рисунок 4.1 – Пример матрицы 3х4

Индекс выделенного элемента определится как

index = 1\*4+2=6

Объем памяти, требуемый для размещения двумерного массива, определится как

n \* m \* (размер элемента)

Однако поскольку при таком объявлении компилятору явно не указывается количество элементов в строке и столбце двумерного массива, традиционное обращение к элементу путем указания индекса строки и индекса столбца является некорректным:

a[i][j] - некорректно.

Правильное обращение к элементу с использованием указателя будет выглядеть как

\*(p+i\*m+j),

где **p** - указатель на массив, **m** - количество столбцов, **i** - индекс строки, **j** - индекс столбца.

**ПРИМЕР**

Ввод и вывод значений динамического двумерного массива

#include <cstdlib>

#include <iostream>

#include <clocale>

using namespace sdt;

int main() {

setlocale(LC\_ALL,"");

int \*a; // указатель на массив

int i, j, n, m;

cout << "Введите количество строк: ";

cin >> n;

cout << "Введите количество столбцов: ";

cin >> m;

// Выделение памяти

a = (int\*) calloc(n\*m, sizeof(int));

// Ввод элементов массива

for(i=0; i<n; i++) // цикл по строкам

{

for(j=0; j<m; j++) // цикл по столбцам

{

cout << "a[" << i << "][" << j << "] = ";

cin >> (a+i\*m+j);

}

}

// Вывод элементов массива

for(i=0; i<n; i++) // цикл по строкам

{

for(j=0; j<m; j++) // цикл по столбцам

{

cout << \*(a+i\*m+j) << " ";

}

cout << endl;

}

free(a);

getchar();

return 0;

}

Результат выполнения программы представлен на рисунке 4.2.

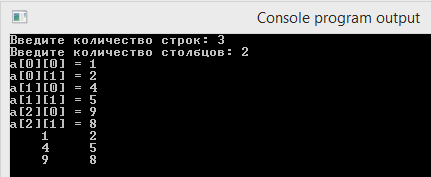


Рисунок 4.2 – Результат работы программы  
 с динамическим выделением памяти для двумерного массива

Возможен также другой способ динамического выделения памяти под двумерный массив - с использованием массива указателей. 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.

Графически такой способ выделения памяти можно представить так, как показано на рисунках 4.3 и 4.4.

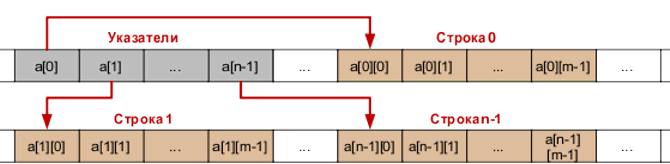


Рисунок 4.3 – Графическое представление  
массива указателей на строки (1 вариант)

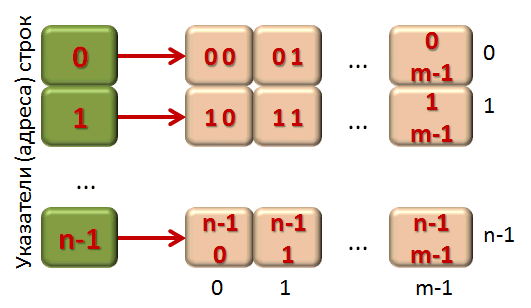


Рисунок 4.4 – Графическое представление  
массива указателей на строки (2 вариант)

При таком способе выделения памяти компилятору явно указано количество строк и количество столбцов в массиве.

#include <cstdlib>

#include <iostream>

#include <clocale>

using namespace sdt;

int main() {

setlocale(LC\_ALL,"");

int \*\*a; // указатель на указатель на строку

int i, j, n, m;

cout << "Введите количество строк: ";

cin >> n;

cout << "Введите количество столбцов: ";

cin >> m;

// Выделение памяти под указатели на строки

a = (int\*\*)malloc(n\*sizeof(int\*));

// Ввод элементов массива

for(i=0; i<n; i++) // цикл по строкам

{

// Выделение памяти под хранение строк

a[i] = (int\*)malloc(m\*sizeof(int));

for(j=0; j<m; j++) // цикл по столбцам

{

cout << "a[" << i << "][" << j << "] = ";

cin >> a[i][j];

}

}

// Вывод элементов массива

for(i=0; i<n; i++) // цикл по строкам

{

for(j=0; j<m; j++) // цикл по столбцам

{

cout << a[i][j];

}

cout << endl;

free(a[i]); // освобождение памяти под строку

}

free(a);

getchar();

return 0;

}

Результат выполнения программы аналогичен предыдущему случаю.

С помощью динамического выделения памяти под указатели строк можно размещать свободные массивы. **Свободным** называется двухмерный массив (матрица), размер строк которого может быть различным. Преимущество использования свободного массива заключается в том, что не требуется отводить память компьютера с запасом для размещения строки максимально возможной длины. Фактически свободный массив представляет собой одномерный массив указателей на одномерные массивы данных.

Для размещения в оперативной памяти матрицы со строками разной длины необходимо ввести дополнительный массив **m**, в котором будут храниться размеры строк.

**ПРИМЕР**

#include <cstdlib>

#include <iostream>

#include <clocale>

using namespace sdt;

int main() {

setlocale(LC\_ALL,"");

int \*\*a;

int i,j, n,\*m;

cout << "Введите количество строк: ";

cin >> n;

a = (int\*\*)malloc(n\*sizeof(int\*));

m = (int\*)malloc(n\*sizeof(int));

// массив кол-ва элементов строк

// Ввод элементов массива

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

cout << "Введите количество столбцов строки " << i<< " ";

cin >> m[i]);

a[i] = (int\*)malloc(m[i]\*sizeof(int));

for(j = 0; j<m[i]; j++) {

cout << "a[" << i << "][" << j << "] = ";

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

}

}

// Вывод элементов массива

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

for(j = 0; j<m[i]; j++) {

cout << a[i][j];

}

cout << endl;

}

getchar();

return 0;

}

Результат выполнения программы представлен на рисунке 4.5.

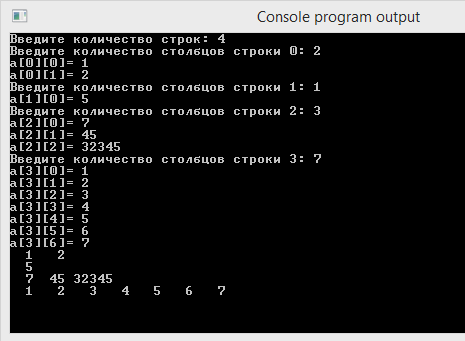


Рисунок 4.5 – Результат работы программы  
 с динамическим выделением памяти для свободного массива

# 5 Перераспределение памяти

Если размер выделяемой памяти нельзя задать заранее, например при вводе последовательности значений до определенной команды, то для увеличения размера массива при вводе следующего значения необходимо выполнить следующие действия:

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

Все перечисленные выше действия (кроме последнего) выполняет функция

void\* realloc (void\* ptr, size\_t size);

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**

Выделить память для ввода массива целых чисел. После ввода каждого значения задавать вопрос о вводе следующего значения.

#include <cstdlib>

#include <iostream>

int main() {

int \*a=NULL, i=0, elem;

char c;

do {

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

cin >> elem;

a = (int\*)realloc(a, (i + 1) \* sizeof(int));

a[i] = elem;

i++;

getchar();

cout << "Next (y/n)? ";

c = getchar();

} while (c=='y');

printf("\n");

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

cout << a[j];

if (i>2) i -= 2;

cout << "\n\nn-2: \n " << endl;

a = (int\*)realloc(a, i \* sizeof(int));

// уменьшение размера массива на 2

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

cout << a[j];

getchar();

return 0;

}

Результат выполнения программы представлен на рисунке 5.1.

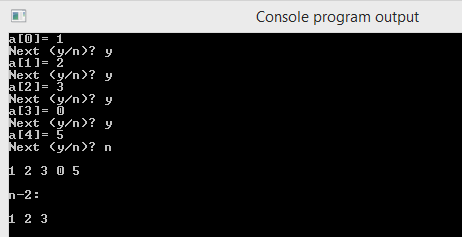


Рисунок 5.1 – Результат работы программы  
 с перераспределением памяти

ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНОЙ РАБОТЫ №12

**Задание 1.** Создать указатель на переменную типа а), выделить для нее динамическую память, считать ее значение и вывести на экран. Освободить память.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **1.12** | а) char  б) double | **2.12** | а) char  б) float | **3.12** | а) float  б) int |

**Задание 2.** Создать указатель на строку длиной **n** символов. Ввести строку с клавиатуры.

1. Уменьшить длину строки на 5 символов в конце добавить символ ‘\0’. Вывести результат на экран.
2. Увеличить длину строки на m символов. Заполнить последние **m** элементов строки символом ‘\*’. Вывести результат на экран.

Очистить память.

**Задание 3.** Создать массив типа б), выделить память на n элементов. Считать элементы с клавиатуры и вывести полученный массив на экран. Очистить память.

**\*Задание 4.** Создать свободный массив типа а). Считать количество элементов в строке и значения элементов с клавиатуры. Вывести результат на экран. Очистить память.

**\*\*Задание 5.** В динамический свободный массив записывать строки, заранее считанные с клавиатуры. То есть, сначала считывается строка, затем определяется ее длина, после чего она заносится в строку динамического массива. Вывести содержимое массива в виде текста. Очистить память.

# ВАРИАНТЫ ЗАДАНИЙ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Вариант** | **Задание** | **Вариант** | **Задание** | **Вариант** | **Задание** |
| **1.1** | а) int  б)char | **2.1** | а) int  б) double | **3.1** | а) int  б) float |
| **1.2** | а) float  б)char | **2.2** | а) float  б) int | **3.2** | а) float  б) int |
| **1.3** | а) char  б) int | **2.3** | а) char  б) float | **3.3** | а) char  б) float |
| **1.4** | а) double  б)char | **2.4** | а) double  б) int | **3.4** | а) double  б) int |
| **1.5** | а) int  б) double | **2.5** | а) int  б)char | **3.5** | а) int  б) double |
| **1.6** | а) float  б) int | **2.6** | а) float  б)char | **3.6** | а) float  б) int |
| **1.7** | а) char  б) double | **2.7** | а) char  б) int | **3.7** | а) char  б) double |
| **1.8** | а) double  б) int | **2.8** | а) double  б)char | **3.8** | а) double  б)char |
| **1.9** | а) int  б) float | **2.9** | а) int  б) float | **3.9** | а) int  б)char |
| **1.10** | а) float  б) int | **2.10** | а) float  б) int | **3.10** | а) float  б)char |
| **1.11** | а) char  б) float | **2.11** | а) double  б)char | **3.11** | а) char  б) int |
| **1.12** | а) char  б) double | **2.12** | а) char  б) float | **3.12** | а) float  б) int |
| **1.13** | а) int  б)char | **2.13** | а) double  б) int | **3.13** | а) int  б) double |
| **1.14** | а) int  б) double |  |  |  |  |