НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Дисциплина «Информатика» (углубленный уровень), 1-й курс, 2-й семестр.

Методические указания

Тематическое занятие 11

Динамический массив: работа с элементами.

Содержание

Динамическое распределение памяти

Динамически распределяемая память (куча)

При разработке программы не всегда можно точно предсказать реальное количество данных, а, значит, и объем необходимой памяти. Поэтому в большинстве языков программирования имеется возможность *создавать и удалять переменные во время выполнения программы*. Такие переменные называются ***динамическими***. Они не описываются заранее в разделе описаний и не имеют своего собственного имени, а память под них выделяется по ходу выполнения программы.

Динамические переменные создаются в отдельной специальной области памяти, которая называется ***динамически распределяемой областью***, или ***кучей (heap)***. Работать с переменными в куче можно только через указатели.

Всю доступную для программы память можно условно разделить на три раздела:

- ***раздел статических переменных*** – память под каждую статическую переменную выделяется в момент начала выполнения программы и освобождается в момент завершения работы программы;
- ***раздел автоматических переменных*** – память под автоматическую переменную выделяется, когда происходит вход в блок кода, который содержит объявление этой переменной (например, вызов функции), и освобождается в момент выхода из этого блока;
- ***раздел динамически распределяемой памяти*** – память выделяется при вызове функции `malloc()` и освобождается при вызове `free()`.

Операция `sizeof`

Унарная операция **`sizeof`** возвращает **количество байт**, требуемое для хранения того типа, который имеет ее операнд. Операнд представляет собой выражение или имя типа, записанное в скобках:

```
sizeof выражение
sizeof (ИмяТипа)
```

Применение операции **`sizeof`** к массиву дает общее количество байт в массиве.

Функции `malloc()` *и* `free()`

В языке С имеются средства, позволяющие выделять и освобождать участки динамически распределяемой памяти во время выполнения программы – функции **`malloc()`** и **`free()`**, прототипы которых описаны в заголовочном файле `<stdlib.h>`.

Для выделения памяти служит функция `malloc()`, которая имеет один аргумент – количество выделяемых байтов памяти. Функция отыскивает подходящий **непрерывный фрагмент** (блок) в свободной области памяти, и возвращает адрес первого байта этого блока в виде ***общего указателя*** (указателя на `void`). При этом значение этого общего указателя можно присвоить указателю любого другого типа без конфликта типов.

Если функции `malloc()` не удастся найти затребованный фрагмент памяти, то она возвращает нулевой указатель `NULL`.

Функция `free()` принимает в качестве аргумента адрес, который возвратила функция `malloc()`, и освобождает выделенную ей память. Обе эти функции следует **всегда использовать в паре**.

Динамические переменные

Динамические переменные не имеют собственного имени, поэтому обращаться к ним нужно через имена их указателей.

Переменная-указатель может находиться в одном из трех состояний:

- содержать адрес переменной, память под которую уже выделена;
- содержать нулевой (пустой) адрес `NULL`;
- находиться в неопределенном состоянии.

В *неопределенном состоянии* указатель находится в начале работы программы до первого присваивания ему конкретного адреса (или нулевого адреса NULL), а также после освобождения области памяти, на которую он указывает.

Пример создания динамической переменной:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
     double *p; /*Указатель в неопределенном состоянии.*/
     p = (double *) malloc(sizeof (double));  /*    Указатель */
     *p = 123.4567;          /* содержит адрес блока памяти для */
     printf("%f", *p);       /* хранения переменной типа double.*/
     free(p); /*Указатель в неопределенном состоянии.*/
     return 0;
}
```

Здесь при выделении памяти `p=(double*)malloc(...)` происходит явное приведение типа, **необходимое** в стандарте ANSI C. Однако присваивание значения указателя на `void`, который возвращает функция `malloc()`, указателю другого типа не рассматривается как конфликт типов.

Следует учитывать, что функция `malloc()` выделяет блок памяти, но **не присваивает ему имени**, а лишь возвращает адрес первого байта этого блока. Этот адрес записывается в переменную-указатель (в приведенном примере – переменная `p`).

Если в процессе работы программы значение переменной-указателя `p` будет изменено до освобождения выделенного блока памяти, то адрес этого блока будет утерян, и доступ к нему станет невозможен. Мы уже не сможем освободить данный блок.

Поэтому ***программисту необходимо следить за тем, чтобы ссылки на каждую выделенную ячейку памяти не терялись***.

Динамические массивы

Массив переменной длины

Стандарт ANSI языка C не позволяет объявить массив переменной длины, используя для обозначения размера массива выражения, содержащие переменные. При объявлении массива в стандарте ANSI C допустимы только константные выражения:

```
#include <stdio.h>
int main(void) {
     int n;
     printf("Введите количество элементов массива: ");
     scanf("%d", &n);
     double a[n]; /* НЕДОПУСТИМО в ANSI C, допустимо в C99 */
     ...
     return 0;
}
```

Такая возможность предусмотрена только **начиная со стандарта C99**.

Однако даже в случае использования компилятора C99, при таком объявлении память выделяется однократно сразу для всего массива при некотором конкретном значении переменной `n`. После изменения значения `n` размер массива не меняется. Вся выделенная для массива память освобождается целиком после завершения работы программы.

Динамический массив

Создание динамического массива возможно с помощью динамического распределения памяти. При этом вместо обычного объявления массива используется указатель, для которого из динамически распределяемой области выделяется нужное количество байт (например, функцией `malloc()`).

Пример создания массива чисел типа `double` в ходе выполнения программы:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
     int i,n;
     double *pa; /*Указатель pa – динамический массив.*/
     printf("Введите количество элементов массива: ");
     scanf("%d", &n);
     pa = (double *) malloc(n * sizeof (double)); /*Выделение
               памяти (n ячеек размером double) для массива pa.*/
     if (pa == NULL) { /*Проверка успешности выделения памяти.*/
          printf("Не удалось выделить память.");
          exit(EXIT_FAILURE); /*Аварийное завершение программы.*/
     }
     for (i=0; i<n; ++i) {
          pa[i] = (double) rand() / (RAND_MAX + 1.0);
     }
     for (i=0; i<n; ++i) {
          printf("%7.5f ",pa[i]);
     }
     free(pa); /*Освобождение памяти, выделенной для массива pa.*/
     return 0;
}
```

Если в динамически распределяемой области отсутствует свободный непрерывный блок памяти необходимого размера, то выполнение функции `malloc()` закончится неудачей – память не будет выделена, а сама функция вернет нулевой указатель (значение `NULL`).

Запись в ячейку по указателю `NULL` приводит к непредсказуемым последствиям и является запрещенной операцией:

```
double *p = NULL; /* Нулевой указатель. */
*p = 123.4567; /* ЗАПРЕЩЕНО! */
```

Поэтому ***результат, возвращаемый функциями выделения памяти всегда необходимо проверять*** на равенство `NULL`.

Функции динамического распределения памяти

В заголовочном файле <stdlib.h> описаны еще две функции динамического распределения памяти: calloc() и realloc(). Для сравнения приведем все функции в общей таблице:

<table>
<tr><th rowspan="2">Функция</th><th colspan="2">Действие в случае …</th></tr>
<tr><th>успеха</th><th>неудачи</th></tr>
<tr><td>malloc(size)</td><td>Выделяет блок памяти размером size байт и возвращает адрес этого блока</td><td>Возвращает NULL</td></tr>
<tr><td>calloc(n,size)</td><td>Выделяет блок памяти из n элементов размером size байт (т.е. всего n*size байт), заполняет все его биты нулями и возвращает адрес этого блока</td><td>Возвращает NULL</td></tr>
<tr><td rowspan="2">realloc(p,size)</td><td>Изменяет размер блока памяти, на который указывает p до size байт и возвращает адрес измененного блока. Если размер блока увеличивается, то содержимое памяти не изменяется; если размер уменьшается, то содержимое памяти усекается.</td><td>Возвращает NULL и оставляет исходный блок памяти неизменным.</td></tr>
<tr><td colspan="2">realloc(NULL,size) аналогично malloc(size), realloc(p,0) аналогично free(p).</td></tr>
<tr><td>free(p)</td><td>Освобождает блок памяти, на который указывает p. Этот блок памяти предварительно должен быть выделен одной из трех предыдущих функций.</td><td>–</td></tr>
</table>

Первые три функции возвращают **общий указатель** (указатель на void).

Функция `calloc()`

Функция calloc() заполняет все **биты** выделенной области нулями, но это ***не гарантирует, что в ячейках окажутся нулевые значения***.

Если элементы массива имеют тип int, то заполнение всех битов нулями будет означать, что значения всех элементов равны 0.

Но, например, если элементы массива имеют тип с плавающей точкой double, то следующий код не гарантирует, что значения всех элементов массива будут равны 0.0:

```
double *p;
p = (double*) calloc(10,sizeof(double));
```

В таких случаях для выделения памяти рекомендуется использовать функцию malloc(), а инициализацию элементов массива проводить в цикле:

```
double *p;
p = (double*) malloc(10*sizeof(double));
for (i=0; i<10; ++i)
     p[i] = 0.0;
```

Функция `realloc()`

Функция `realloc(p,size)` получает указатель `p` на блок памяти, размер которого необходимо изменить, а возвращает один из следующих указателей:

1) указатель на тот же самый блок, если его размер удалось изменить (например, при уменьшении размера блока, он остается на месте, а отсекаемая от него часть памяти освобождается);
2) указатель на новый блок памяти, если исходный блок не удалось изменить (при этом значения элементов исходного блока копируются в новый, а выделенная для исходного блока память освобождается, причем исходный указатель `p` переходит в неопределенное состояние);
3) нулевой указатель `NULL` в случае, если размер памяти требуется увеличить, но в ней отсутствует свободный непрерывный блок необходимого размера (при этом исходный блок памяти останется неизменным).

Все эти случаи рассмотрены в таблице для следующего примера:

```
#define N □□□□
#define M □□□□
int *p, *q;
p = (int*) malloc(N*sizeof(int));
q = (int*) realloc(p, M*sizeof(int));
```

Пусть здесь после выполнения функции `malloc()` значение адреса, присвоенное указателю `p`, стало равным `4BF2:3D5A`, тогда:

№	N	M	q	p	Комментарий
1)	100	80	4BF2:3D5A	4BF2:3D5A	Уменьшение блока
	100	120	4BF2:3D5A	4BF2:3D5A	Успешное увеличение блока (без перемещения его в памяти)
2)	100	500	87C4:AF92	**?**	Успешное увеличение блока (с перемещением его в памяти)
3)	100	10000	NULL	4BF2:3D5A	Неудачное увеличение блока

В таблице «**?**» означает, что указатель `p` находится **в неопределенном состоянии**. Чаше всего он сохраняет свое предыдущее значение (`4BF2:3D5A`), но память по этому адресу уже освобождена и не может больше использоваться.

Пример увеличения размера массива в ходе выполнения программы:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int  i,
        *p, /* p – динамический массив.*/
        *q; /* q – вспомогательный указатель.*/
    p=(int*)calloc(5,sizeof(int)); /*Выделение 5-и ячеек памяти.*/
    if (p==NULL) { /*Проверка успешности выделения памяти.*/
        printf("Не удалось выделить память.");
        exit(EXIT_FAILURE);
    }
    for (i=0; i<5; ++i)
        p[i]=i;
```

```
    q=(int*)realloc(p,10*sizeof(int));/*Увеличение до 10-и ячеек.*/
    if (q==NULL) { /*Проверка успешности увеличения памяти.*/
        printf("Не удалось увеличить память.");
    }
    else {
        p=q;
        for (i=5; i<10; ++i)
            p[i]=i;
    }
    free(p); /*Освобождение памяти.*/
    return 0;
}
```

Типичная ошибка при использовании функции `realloc()`

При использовании функции `realloc()` необходимо проводить проверку возвращаемого ею значения, а в случае неудачи нельзя потерять указатель на исходный блок памяти. Для этого необходимо использовать вспомогательный указатель (`q` в приведенном примере).

Например, следующий код представляет собой типичную ошибку:

```
int *p;
p = (int*) malloc(10,sizeof(int));
...
p = (int*) realloc(p, 1000*sizeof(int)); /* ОШИБКА */
...
```

В случае неудачного перераспределения памяти функция `realloc()` вернет `NULL` и указатель на исходный блок памяти будет потерян.

Передача динамического массива в функцию

Передача содержимого массива по адресу

Напомним, что передача обычного (статического или автоматического) массива в функцию происходит **по адресу**, благодаря чему мы можем изменять значения элементов передаваемого массива. Например:

```
void func(int *a, int n) { /* a – указатель, n – кол-во элементов*/
    a[n-1] = 123; /* изменяем значение последнего элемента массива m*/
}
int main(void) {
    int m[10];  /* объявление массива m */
    ...
    func(m,10);  /* m – массив */
    ...
}
```

При таком способе передачи в функцию передается **адрес первого элемента массива**: фактическим параметром при вызове функции `func(m,10)` служит имя массива `m`, а формальным параметром в заготовке функции `func()` является указатель `int *a`.

При вызове функции значение адреса первого элемента массива `m` **копируется** в ячейку памяти, выделенную для хранения значения указателя `a`. То есть содержимое массива передается **по адресу**, а сам этот адрес (указатель на массив) передается **по значению**.

Внутри функции мы можем изменить значение указателя `a`, но это изменение не отразится на значении адреса передаваемого массива `m`. Мы всего лишь потеряем связь с массивом.

Ошибки при передаче динамического массива в функцию

При использовании обычного статического (автоматического) массива не может возникнуть задачи изменения его адреса, поскольку память для него выделяется в момент объявления массива (входа в блок) и освобождается в момент завершения работы программы (выхода из блока). Необходимость в изменении адреса массива может возникнуть только для динамического массива в случае изменения его размера (при вызове функции `realloc()`).

Заменим в предыдущем примере обычный массив `m` на динамический `pm`:

```
void func(int *a, int n) { /* a – указатель, n – кол-во элементов */
     a[n-1] = 123;
}
int main(void) {
     int *pm;  /* pm – указатель на динамический массив */
     pm = (int*) malloc(10*sizeof(int));
     ...
     func(pm,10);
     ...
     free(pm);
}
```

Функция `func()` при этом не изменяется. При ее вызове значение адреса массива `pm` также **копируется** в указатель `a`, то есть сам адрес передается **по значению**.

Обратите внимание, что указатель `a` является автоматической переменной – память для него выделяется в момент вызова функции и освобождается в момент ее завершения.

Теперь попытаемся внутри функции `func()` увеличить размер динамического массива, используя `realloc()`:

```
/* Некорректный пример перераспределения памяти. */
void func(int *a, int n) { /* a – указатель, n – кол-во элементов */
     int *q; /* q – вспомогательный указатель */
     q = (int*) realloc(a, 1000*sizeof(int));
     if (q==NULL) { /* проверка успешности увеличения памяти */
          printf("Не удалось увеличить память.");
     }
     else {
          a = q;
          a[999] = 123;
          ...
     }
}
```

Если увеличение памяти с помощью `realloc()` прошло успешно, то адрес нового блока сохраняется во вспомогательном указателе `q`, а затем копируется в указатель `a`. Изменение значения указателя `a` не влияет на указатель `pm` – адрес нового блока памяти не будет известен в вызывающей функции `main()`. Но в результате работы `realloc()` старый блок памяти (по указателю `pm`) освобождается. Таким образом, указатель `pm` ссылается на освобожденную память, а значит, находится в неопределенном состоянии.

Более того, сразу после окончания работы функции `func()` память, выделенная для ее автоматических переменных (формальных параметров и локальных данных), освобождается. В том числе освобождается память переменных `q` и `a`, в которых хранится адрес нового блока для исходного массива – доступ к новому блоку теряется.

В результате работы такой некорректной функции `func()` мы можем потерять весь динамический массив и всю выделенную для него память.

Передача указателя на массив по адресу

Выход из описанного положения следующий – передавать указатель на динамический массив не по значению, а по адресу. То есть передавать адрес по адресу, а значит – передавать **указатель на указатель**.

Вот корректная реализация рассматриваемого примера:

```
void func(int **a, int n) { /* a – указатель на указатель */
     int *q;
     q = (int*) realloc(*a, 1000*sizeof(int));
     if (q==NULL) {
          printf("Не удалось увеличить память.");
     }
     else {
          *a = q;
          (*a)[999] = 123;  /* или   *((*a)+999) = 123;   */
          ...
     }
}
int main(void) {
     int *pm;
     pm = (int*) malloc(10*sizeof(int));
     ...
     func(&pm,10); /* в функцию передается адрес указателя pm */
     ...
     free(pm);
}
```

Здесь `int **a` – это объявление указателя с именем `a` на указатель типа `int*`, который в свою очередь указывает на переменную типа `int`.

Теперь в ячейке памяти указателя `a` хранится адрес другой ячейки, в которой содержится адрес первого элемента динамического массива. Если элементы массива имеют тип `int`, то адрес массива – тип `int*`, а указатель `a` – тип `int**`.

В функции `main()` переменная-указатель на динамический массив имеет имя `pm` и хранит адрес первого элемента массива. Внутри функции `func()` не имеется отдельной переменной-указателя, которая бы ссылалась на этот массив, но доступ к адресу первого элемента массива можно получить разыменовав

указатель a. Значение *a является адресом первого элемента массива, то есть значением переменной-указателя pm.

В данном примере при выполнении операции присваивания *a=q; адрес нового блока памяти q копируется в ячейку, адрес которой передан в функцию func(), то есть в ячейку, выделенную для указателя pm при его объявлении. В результате этого адрес нового блока памяти станет известен в вызывающей функции main() как значение указателя pm.

Доступ к элементам массива по указателю на указатель

Из функции func() доступ ко всем элементам динамического массива может быть получен с использованием **двойного разыменования** указателя a. При этом для указания индексов элементов массива можно использовать квадратные скобки ([□]), помня, что их приоритет выше, чем у операции разыменования (*□).

Например:

Двойное разыменование	Индексы в квадратных скобках	Описание
**a или *(*a)	(*a)[0]	первый элемент массива
*((*a)+i) или *(*a+i)	(*a)[i]	i-й элемент массива (объявлено int i;)

Здесь круглые скобки обязательны при использовании индексов элементов в квадратных скобках.