Динамическая память

А. Г. Фенстер, fenster@fenster.name

2 апреля 2010 г.

Конспект семинара №8 по программированию для студентов 1 курса ММФ НГУ. Помните, что чтение конспекта не делает посещение соответствующего семинара необязательным! О найденных ошибках и неточностях, пожалуйста, пишите мне по адресу fenster@fenster.name. Также буду рад получить любые комментарии по поводу этого текста.

На этом семинаре будет рассматриваться работа с динамической памятью — техника, которая используется в тех случаях, когда размер требуемой памяти неизвестен на этапе написания программы и нужно позволить программе запрашивать дополнительную память для хранения данных в процессе работы.

1 Зачем нужна динамическая память

Иногда программист не может заранее предсказать, сколько памяти потребуется программе для работы. Например, простейшая программа, повторяющая введённый пользователем текст дважды, может быть написана следующим образом:

```
#define N 1000
char buffer[N];
int size = 0;
int i;
int c;

while ((c = getchar()) != EOF) /* читаем до конца введённого текста */
{                              /* (Ctrl+D в Linux, Ctrl+Z в Windows) */
    if (size == N)
        break;  /* кончилось место в массиве */
    buffer[size] = c;
    size++;
}
```

```
for (i = 0; i < size; i++) /* печатаем первый раз */
    putchar(buffer[i]);

for (i = 0; i < size; i++) /* печатаем второй раз */
    putchar(buffer[i]);
```

Основная проблема здесь — необходимость ограничения размера массива `buffer` ещё на этапе создания программы. Конечно, если тысячи байт не хватает — можно указать размер и две, и пять тысяч, но проблему это не решит. Нужна возможность задавать (и изменять!) размер выделенной памяти по ходу работы программы.

Ещё одна проблема появляется при попытке сделать функцию, результатом работы которой является массив (или строка: строки в C, как мы помним, являются массивами символов, ограниченными символом с кодом 0). Любой массив, определённый внутри функции, будет автоматически уничтожен при выходе из неё, поэтому функция, возвращающая локальный массив, правильно работать не будет[1].

2 Функции работы с динамической памятью

Так или иначе, мы приходим к тому, что неплохо было бы иметь возможность выполнять следующие операции:

1. Запрашивать в процессе работы программы дополнительную память для хранения данных. Желательно также, чтобы эта память могла существовать и после выхода из текущей функции.

2. Изменять размер памяти, которая ранее была нам выделена (это пригодится всё в той же задаче про сохранение того, что ввёл пользователь).

3. Если выделенная память нам больше не нужна, «отдавать» её обратно системе.

Существуют специальные функции, выполняющие эти действия. Это функции `malloc`, `realloc` и `free`, для использования которых нужно

[1] Исключение — если массив объявлен как *статический* (`static`); подробнее можно прочитать, например, вот в этой лекции для ФИТа.

подключить `stdlib.h`. Все эти функции работают с *нетипизированными указателями* — переменными типа `void *`, про которые известно, что они хранят адрес некоторой ячейки памяти, в которой может находиться значение любого типа (в отличие, например, от `int *` или `char *` — в этих случаях мы знаем, адрес значения какого типа хранит данный указатель).

Также для работы с новыми функциями нам впервые потребуется специальный оператор `sizeof`, возвращающий размер данного типа или переменной в байтах. Его результатом является целое число[2]. Например, на большинстве современных компиляторов следующая строчка напечатает число 4:

```
printf("%d\n", sizeof(int));
```

и это будет означать, что для хранения значения типа `int` необходимо 4 байта памяти.

Для выделения памяти используется функция `malloc`. Параметром она принимает размер памяти, который требуется в данный момент программе, а возвращает указатель типа `void *` на начало выделенной памяти (или нулевой указатель `NULL` в случае ошибки, если память выделена не была). В языке C (не в C++![3]) значение типа `void *` можно присвоить любому другому указателю, поэтому работает, например, такой код:

```
int *p = malloc(sizeof(int));
if (!p) /* p равен нулю */
    ошибка;
/* сейчас можно использовать *p как int, например: */
*p = 7;
```

Обычно, конечно, выделяют память сразу под массив из нескольких значений. Вот так, например, можно создать массив с размером, заданным на этапе выполнения программы:

```
int *p;
int N;

scanf("%d", &N);
p = malloc(N * sizeof(int));
if (!p)
    ошибка;
```

[2]Более строго — значение типа `size_t`.

[3]В C++ требуется явное приведение типа.

Выделенная таким образом память будет считаться используемой, пока не будет вызвана функция `free`, «освобождающая» ранее выделенную память (или до конца работы программы):

```
free(p);
```

Помните, что содержимое памяти, выделенной при помощи `malloc`, является неопределённым. Не забывайте инициализировать (обнулять) такие динамические массивы, если это необходимо!

Освобождать память после работы в небольших программах является правилом хорошего тона, тогда как в больших (долго работающих) программах делать это необходимо, чтобы не возникало ситуации *утечки памяти*, когда программа со временем начинает потреблять всё больше и больше памяти.

Функция `realloc` изменяет размер выделенной памяти и возвращает новый указатель (он может не совпадать со старым, поэтому старым пользоваться будет уже нельзя). Вторым параметром она принимает новый требуемый размер памяти:

```
q = realloc(p, newsize);
if (!q)
{
    ошибка; размер не изменился и p указывает на старую память
}
else
{
    q указывает на память с изменённым размером, старое значение p использовать нельзя
    p = q;
}
```

Само собой, информация, хранившаяся в памяти, будет скопирована в новое место автоматически.

Удобное свойство функции `realloc`: вызов `realloc(NULL, size)` эквивалентен вызову `malloc(size)`.

Чуть подробнее про `malloc`, `realloc` и `free`, а также про `calloc`: информация в отдельном файле.

3 Примеры

Задача. Реализуйте функцию `strdup` (она присутствует практически во всех современных компиляторах), которая принимает параметром

строку и дублирует её — выделяет необходимое количество памяти, копирует туда строку и возвращает полученную копию. Помните, что согласно стандарту C `sizeof(char) == 1`, поэтому необязательно умножать размер строки на размер `char`. Не забывайте также про завершающий ноль.

Задача. Напишите функцию, выполняющую чтение одного слова из файла. Пусть эта функция выделяет сначала несколько байт, затем увеличивает размер выделенной памяти, если его не хватает, чтобы вместить слово. Чтение делайте посимвольным при помощи `getchar`. Если слово не прочитано из-за того, что достигнут конец файла — верните `NULL`, в противном случае верните адрес начала прочитанного слова в памяти. Не забудьте про нулевой символ в конце строки.

Обычно при реализации таких задач создают две целочисленных переменных: `allocated` для запоминания размера выделенной области и `used` для запоминания, сколько байт реально было использовано. Если «свободных» мест в массиве не осталось, необходимо увеличить размер памяти при помощи `realloc`. Крайне неэффективно увеличивать память на 1 байт на каждом шаге, лучше вызывать `realloc` реже, добавляя при этом больше байт. Необходимость в конце добавить к строке завершающий 0 делает код чуть менее понятным, чем он мог бы быть.

В простом варианте (если считать, что слова разделены ровно одним пробелом) функция может выглядеть примерно так:

```
char *readword(void)
{
    char *p = NULL;
    int allocated = 0;
    int used = 1; /* помним о завершающем нуле */
    int c;

    while ((c = getchar()) != EOF)
    {
        if (c == ' ')
            break;
        if (allocated <= used)
        {
            char *q = realloc(p, allocated + CHUNK_SIZE);
            if (!q) /* отсутствует память */
                break;
            p = q;
            allocated += CHUNK_SIZE;
        }
        p[used - 1] = c;
        used++;
```

```
    }

    if (p) /* успели что-то прочитать */
    {
        p[used - 1] = 0;
    }

    return p;
}
```

Константу `CHUNK_SIZE` можно задать равной ожидаемой длине слова.

4 Проверочное задание («пятиминутка»)

Тема: функции, возвращающие строки (см. задание).