Хранение данных

А. Г. Фенстер, fenster@fenster.name

27 марта 2010 г.

На этом занятии мы поговорим о способах хранения данных в ваших программах. По идее, ничего из сказанного не должно оказаться для вас незнакомым, но всегда лучше повторить известные факты, чем пропустить что-то важное.

1 Массивы

В «классическом» C (стандарт C89) массивы обязаны иметь длину, определённую на этапе компиляции. Однако, в стандарте C99 появилась возможность создавать массивы, длина которых определяется на этапе выполнения программы. Проще говоря, работает такой код:

```
int n = 20;
int A[n]; /* C99 */
```

Как мы помним, стандарт C99 не поддерживается компилятором из Microsoft Visual Studio, а также некоторыми другими компиляторами. Соответственно, для написания переносимых программ этот способ хранения данных не подходит: нужно ограничиваться массивами определённой на этапе компиляции длины.

```
#define SIZE 20
int A[SIZE]; /* массив постоянной длины */
```

2 Динамические массивы

Если всё же необходимо иметь возможность переносимым способом создавать массивы произвольного размера, приходится использовать динамическую память и размещать эти массивы в куче, выделяя память

при помощи `malloc` / `calloc` / `realloc`. Дополнительно мы получаем возможность изменять размер выделенной памяти, но теперь нам нужно самим отслеживать, нужна ли нам выделенная память, и не забывать об её освобождении.

```
int n = 20;
int *p1 = malloc(n * sizeof(int)); /* содержимое не инициализировано */
int *p2 = calloc(n, sizeof(int));  /* содержимое заполнено нулями */
...
free(p1);  /* не забываем освободить память */
free(p2);
```

С использованием функции `realloc` реализуется достаточно стандартный способ создания массива для хранения данных, размер которых неизвестен при создании программы — например, для чтения содержимого файла в память. Напомню, что `realloc` изменяет количество выделенной памяти (его можно как увеличить, так и уменьшить).

Обычно **не следует** вызывать `realloc` для каждого нового прочитанного байта (студенты очень любят так делать). Намного эффективнее выделять память небольшими кусками, увеличивая размер массива по мере необходимости:

```
#define CHUNK_SIZE 1000

char *p = NULL;
size_t allocated = 0;
size_t used = 0;
int c;
FILE *f = ...; /* fopen и проверка */

while ((c = fgetc(f)) != EOF)
{
    if (allocated == used)
    {
        char *q = realloc(p, allocated + CHUNK_SIZE);
        if (!q)
        {
            perror("realloc");
            break;
        }
        p = q;
        allocated += CHUNK_SIZE;
    }
    p[used++] = (char)c;
}
```

В этом фрагменте кода используется удобное (и описанное в стандарте) свойство функции `realloc`: если передать ей `NULL` в качестве адреса, она отработает как `malloc`.

Конечно, следующим шагом по оптимизации будет отказ от посимвольного чтения в пользу чтения буферизованного (`fread`), с его использованием чтение будет выполнено намного быстрее:

```
size_t allocated = 0;
size_t used = 0;
char *p = NULL;
size_t numread = 0;

FILE *f = ...; /* fopen и проверка */

do
{
    char *q = realloc(p, allocated + CHUNK_SIZE);
    if (!q)
    {
        perror("realloc");
        break;
    }
    p = q;
    allocated += CHUNK_SIZE;
    numread = fread(p + used, 1, CHUNK_SIZE, f);
    used += numread;
}
while (numread == CHUNK_SIZE);
```

Ещё более быстрый (но непереносимый, зависящий от ОС) способ чтения файла — отображение файла в память при помощи системного вызова `mmap` (стандарт POSIX) и подобных.

Динамические массивы (как и любые другие массивы) всегда размещаются в памяти единым куском, поэтому вполне возможна ситуация, при которой для очень большого запрошенного объёма памяти единого куска не найдётся и `realloc` не отработает (вернёт `NULL`), но при этом суммарный объём свободной памяти превышает запрошенный (т. е. наблюдается фрагментация памяти).

Ещё один недостаток динамических массивов — необходимость выполнения сдвига элементов за время $O(N)$ для вставки или удаления элемента в произвольное место массива.

3 Списки

Динамические списки (они же «связанные списки», linked lists) являются одной из базовых структур данных для хранения информации неизвестного на этапе компиляции объёма, при этом данные располагаются в памяти произвольным образом (не обязательно подряд).

Базовые сведения о списках представлены в отдельном файле.

Списки — это очень важно, работать со списками должен уметь каждый на потоке!

На всякий случай перечислю отдельно основные правила работы со списками.

1. Выделение памяти (`malloc`) должно происходить только при создании нового элемента списка. Частая ошибка: непонимание, что такое выделение памяти, в итоге `malloc` применяется чуть ли не каждой переменной типа `struct item *`. Естественно, это ошибка. Рисуете новый прямоугольничек — делаете `malloc`.

2. То же самое с освобождением памяти. Стёрли прямоугольничек — вызвали `free`.

3. В односвязный список вставить элемент можно только **после** известного, удалить можно только элемент **после** известного.

4. Разыменовывать можно только гарантированно не равный нулю указатель. Конструкций типа `p->next->data`, `p->next->next` лучше избегать совсем.

5. Всегда помните про крайние случаи: пустой список, список из одного элемента, обработка первого и последнего элемента списков.

4 Абстрактные структуры данных

Списки и массивы можно использовать для реализации структур данных более высокого уровня (работа которых практически не зависит от того, как именно они реализованы): стека, очереди, дека, кучи.

Подробная информация представлена в отдельном файле.