GeekBrains

Основы C++

Управление памятью

На этом уроке

1. Узнаем, что представляет из себя процесс управления памятью, как выделять и освобождать динамическую память
2. Научимся не допускать утечек памяти и изучим операторы new и delete
3. Рассмотрим работу с файловой системой и потоками ввода-вывода

Оглавление

[На этом уроке](#)

[Динамическая память](#)

- [Области памяти](#)
- [malloc, calloc, realloc, free](#)
- [Пример использования](#)
- [Операторы new и delete](#)

[Особенности использования](#)

- [Ввод-вывод данных в языке Си.](#)
- [Работа с файлами.](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Динамическая память

Вопросы управления памятью мы затрагивали в той или иной мере, когда обсуждали на предыдущих занятиях объявление и инициализацию переменных, а также передачу аргументов в функции и области видимости имен, меток и переменных. Но мы не касались вопросов распределения реальной (физической) памяти компьютера для размещения переменных и иных объектов языка Си.

Среди программистов бытует два полярно противоположных мнения по вопросу управления аппаратными ресурсами, в том числе – физической памятью. Приверженцы первого из них считают, что управлением физической памятью на вычислительной платформе должны заниматься исключительно компилятор и операционная система (ОС), и не должен программист отвлекаться на такие мелочи. Приверженцы второго направления убеждены как раз в противоположном – все должно быть под контролем программиста, а аппаратные ресурсы – в первую очередь.

Интересно проследить взаимосвязь этих мнений и концепций построения современных ОС. Все поколения ОС Windows нацелены на то, чтобы самостоятельно разбираться с аппаратными ресурсами (т.н. технологии “Plug&Play”, или, выражаясь по русски, любая (уборщица, кухарка, …) должна программировать под Windows, особо не разбираясь ни в распределении памяти, ни в аппаратном обеспечении вообще. Все под контролем ОС, или стремится быть таковым). В многочисленных на сегодня дистрибутивах и сборках Linux и других UNIX-подобных систем все как раз наоборот – там изначально и по сегодняшний день существует жесткое понятие системного администратора (как правило, он же и системный программист), то есть, до определенного уровня там

на сегодня тоже “Plug&Play”, но все аппаратные ресурсы, в конечном итоге, локально или удаленно обязан контролировать администратор (т.е. человек).
Каждый волен выбирать, какого из этих направлений придерживаться. Заметим лишь, что живем мы не в идеальном мире, аппаратные ресурсы на современных компьютерах не бесконечны, и современные компиляторы не настолько умны, чтобы самостоятельно грамотно распорядиться ограниченными ресурсами памяти. Современные компиляторы способны лишь правильно выделить ячейки памяти под понятные им, компиляторам, типы переменных. Поэтому в языках C/C++, на котором, кстати, написаны и Unix, и Linux, и Windows, заложены сразу два основных механизма управления распределением физической памяти.
Первый состоит в использовании переменных. Область памяти для них выделяется компилятором автоматически во время компиляции программы, и не может быть изменена до завершения программы. Затратно, но просто, понятно и надежно.
Вторым способом является **динамическое распределение** выделяемой под объекты программы (в том числе и под переменные) памяти.

Области памяти

В самых общих чертах, без учета сегментации, компиляторы разных версий C/C++ распределяют оперативную память под компоненты программы следующим образом:

Область стека (занимается сверху вниз)	Старшие адреса памяти
Область динамической памяти	
Область статических данных	
Область команд	Младшие адреса памяти

Область динамической памяти (программисты называют ее “**кучей**” (“heap”), серьезные теоретики языка - пулом свободной памяти) является адресуемым пространством и располагается между областью глобальных (статических) данных и стеком. Управление динамической памятью обеспечивает возможность занимать и освобождать блоки из этой области памяти **в процессе исполнения программы**.
Благодаря управлению динамической памятью отпадает необходимость заранее резервировать всю память, используемую программой, что оказывается полезным в ситуациях, когда требуемый объем памяти заранее неизвестен или зависит от конкретного применения программы. А все переменные, как правило, никогда и не нужны модулям программы одновременно. Ключевое слово здесь - одновременно. Управление динамической памятью позволяет программе определять доступную память для конкретного компьютера и данного приложения, а затем занимать в ней область требуемого размера.

`malloc`, `calloc`, `realloc`, `free`

Во всех на сегодня реализациях языка Си стандартные библиотеки содержат функции `malloc`, `calloc`, `realloc`, `free`, которые позволяют занимать свободные области динамической памяти, менять размер ранее занятых областей, и освобождать ранее занятые области. Обратите внимание на то, что в языке Си эти операции выполняются **функциями**, а в языке C++, как мы покажем далее, и функциями, и **операторами** языка C++.
Приведем описание наиболее употребимых **функций** управления памятью (на самом деле в разных компиляторах их может быть гораздо больше):

```
void *malloc(unsigned int nbytes);
void *calloc(unsigned int nitem, unsigned int size);
```

```
void *realloc(void optr, unsigned int size);
void free(void *ptr);
```

Приведём пример, а затем дадим к каждой строке подробные комментарии. Из приведенных комментариев к примерам должно стать понятно назначение и работа стандартных функций динамического управления памятью. Эти функции определены также и в библиотеках диалекта языка C++. Более того, там они могут быть переопределены программистом по собственному усмотрению. Но переопределение этих функций и соответствующих операторов весьма опасное занятие, и нужно иметь достаточно высокую квалификацию, а главное – веские причины, чтобы этим заниматься.

```
#include <stdlib.h>
char *char_ptr;
int *int_ptr1;
int *int_ptr2;

char_ptr = (char*) malloc(200);
free(char_ptr);
int_ptr = (int*) malloc(20*sizeof(int));
free(int_ptr);
int_ptr1 = (int*) calloc(100, sizeof(int));
int_ptr2 = (int*) realloc(int_ptr1, 200* sizeof(int));
free(int_ptr2);
```

Директива `#include<stdlib.h>` Подключает стандартную библиотеку, в которой определены функции `malloc`, `calloc`, `realloc`, `free`.
Строка `char_ptr = (char*) malloc(200);` - это запрос на выделение двухсот байт из кучи, преобразование типа указателя на выделенный блок памяти из неопределённого `void*` в `char*` и присвоение полученного указателя в качестве значения переменной `char_ptr`, если память успешно выделена. Если же память, по каким то причинам, не выделилась, значением переменной будет `NULL`.
`free(char_ptr);` освободит область памяти, на которую указывает `char_ptr` и “вернёт” её в кучу.
Строка `int_ptr1 = (int*) calloc(100, sizeof(int));` - это также запрос на выделение из кучи памяти, на этот раз сто элементов, каждый размером `sizeof(int)`, здесь также происходит преобразование типов указателей, и также будет присвоен `NULL` в случае, если память не будет выделена. Существенным отличием будет то, что при использовании `calloc` выделенная память будет принудительно очищена и содержать нулевые значения.
Наконец `int_ptr2 = (int*) realloc(int_ptr1, 200* sizeof(int));` - это самый неоднозначный оператор из всех рассматриваемых, поскольку не только реализуется разными версиями компиляторов по разному, но и работающего не всегда так, как ожидает программист: на этой строке выполняется запрос на увеличение в два раза размера ранее выделенного из кучи блока памяти, также преобразование типов, и если такой запрос был не успешен, то по старому указателю будет находиться не старая область памяти, а `NULL`.
Резюмируем сведения о функциях `malloc`, `calloc`, `realloc`, `free`:

- Функция `malloc` (memory allocate - занять память) занимает в пуле свободной памяти блок размером не менее `nbytes` и возвращает указатель на первый байт этого блока. Содержимое занятого блока остается **неопределенным (т.е. не обнуляется)**. Если необходимо, конец блока динамической памяти расширяется с помощью функций управления памятью нижнего уровня (уровня операционной системы), но это выходит за рамки нашего курса.
- Функция `calloc` (clear allocate - занять и очистить) занимает блок памяти для массива из `nitem` элементов, размер каждого из которых составляет `size` байтов. Эта функция, в отличие от `malloc`, **заполняет занятый блок нулевыми байтами**.

- Функция `realloc` (reallocate - перезанять) может быть использована для изменения размера **ранее занятого блока памяти**. При этом **адрес начала блока может измениться и скорее всего изменится**.
- Функция `free` (освободить) возвращает блок памяти в пул свободной памяти. Ее аргументом **должен быть указатель, возвращенный** при предшествующем вызове функций `malloc` или `calloc`.

Функции `malloc`, `calloc`, `realloc` возвращают указатель типа `void`, который в Си означает тип указатель. Перед сохранением в переменной возвращенный `void` указатель обязательно должен быть **явно преобразован** в тип указателя на те данные, которые предполагается хранить в этой области памяти. Некоторые, на сегодня очень редкие компиляторы, не поддерживают тип указателей `void`. Для функций динамического управления памятью они возвращают тип `char*`. Этот тип также должен быть явно преобразован в необходимый тип указателя, все действия аналогичны показанным.

Пример использования

Следующая программа принимает количество имен сотрудников, для хранения которых динамически распределяет память. Количество имен задается в командной строке как аргумент при вызове программы (на прошлых занятиях мы говорили, что вызов функции main() может быть параметризованным). Программа запрашивает в цикле имена сотрудников в количестве, полученном из командной строки. Имена сотрудников (для примера, не более 30 символов на каждое имя) запоминаются, а по завершении ввода считываются обратно из динамической памяти через массив указателей на них. Причем массив указателей также хранится в динамически выделенной памяти.

```
#include <stdlib.h>
int main(int argc, char *argv[]) {
   char **emp_ptr;
   char *name;
   int num_of_emp, i;
   if (argc != 2) {
       printf("Корректный вызов программы: %s количество_сотрудников: \n",
argv[0]);
       return 0xDEAD;
   }
   num_of_emp = atoi(argv[1]);
   if ((emp_ptr = (char**) calloc(num_of_emp, sizeof(char*))) == nullptr) {
       printf("Недостаточно памяти.\n");
       exit(0xBAD);
   }
   if ((name = (char*) calloc(30, sizeof(char))) == nullptr) {
       printf("Недостаточно памяти.\n");
       exit(0xDECADE);
   }
   for (i = 0; i < num_of_emp; ++i) {
       printf("Введите имя сотрудника: ");
       gets(name);
       if ((emp_ptr[i] = (char*) calloc(strlen(name) + 1, sizeof(char))) ==
nullptr) {
           printf("Недостаточно памяти.\n");
           exit(0xBEEF);
       }
       strcpy(emp_ptr[i], name);
   }
   free(name);
   printf("--------------------------------\n");
```

```
    for (i = 0; i < num_of_emp; ++i)
        printf("Программист #%d: %s\n", i, emp_ptr[i]);
    free(emp_ptr);
    return 0;
}
```

Компилируем и запускаем его на исполнение. Результат работы программы должен быть таким:

```
> staff_office
Корректный вызов программы: staff_office количество_сотрудников

> staff_office 3
Введите имя сотрудника: Николай Нидвораев
Введите имя сотрудника: Василий Алибабаев
Введите имя сотрудника: Степан Тяпляпов
 --------------------------------
Программист #1: Николай Нидвораев
Программист #2: Василий Алибабаев
Программист #3: Степан Тяпляпов
```

Заголовочный файл `<stdlib.h>` включается в программу в самом начале и делает доступными для использования стандартные функции языка `malloc`, `calloc`, `free`, которые мы только что обсудили, функцию преобразования из символьной строки в целое число `atoi`, при помощи которой мы получаем из символьной строки вызова программы второй аргумент – количество имен сотрудников (`num_of_emp`), и функции `printf` и `gets`, обеспечивающие вывод на терминал в соответствии с форматной строкой и ввод с клавиатуры.

Аргументами функции `main` являются `int argc и char *argv[]`. Через первый из них в функцию передается количество параметров, передаваемых в `main`, через второй – указатель на массив передаваемых параметров.

Если в командной строке вызова программы на исполнение указаны только имя программы и через пробел `количество_сотрудников`, то через `argc` в `main` будет передано число два, а в массиве `argv[]` будет два начальных элемента. При любом другом варианте командной строки `argc` будет не равен двум, и программа уйдет по ветке `else` оператора `if`/`else` на печать сообщения о неправильном вызове и завершение выполнения программы операцией `return`.

Через `argv[0]` из командной строки вызова программы в `main` в любом случае будет передано имя файла исполняемой программы. Это имя и выводится в строке о неправильном вызове. А в случае корректного вызова программа получает из `argv[1]` количество вводимых имен сотрудников, преобразует его в целое число при помощи функции `atoi` (код ASCII в целое), и помещает это целое в переменную `num_of_emp`.

Затем выделяется память для массива указателей на имена сотрудников, адрес которой записывается в переменную `emp_ptr`, тип которой объявлен как **указатель на указатель на char** или указатель на строковый литерал. Функция `calloc(num_of_emp, sizeof(char*));` запрашивает выделение памяти в размере, достаточном для хранения `num_of_emp` элементов размером указателя на `char`, и в случае неравенства нулю сохраняет адрес этой памяти в переменной `emp_ptr`.

Далее аналогичным образом выделяется фрагмент памяти, необходимый для временного хранения 30 байт информации вводимых имен сотрудников.

Потом в цикле for, параметр которого равен числу сотрудников, хранимому в `num_of_emp`, выводится подсказка, и с помощью функции `gets(name);` в память временного хранения вводится имя очередного сотрудника.

Затем снова при помощи функции `calloc(strlen(name) + 1, sizeof(char))` запрашивается блок памяти для хранения только что введенного имени, на один символ большего размера (для хранения нулевого конечного байта), чем только что введенное имя. В случае успешного выделения этого блока его адрес заносится в `i`-й элемент динамического массива указателей `emp_ptr[num_of_emp]` через операцию присваивания.

Затем в только что выделенный блок памяти при помощи функции `strcpy(emp_ptr[i], name);` копируется имя очередного сотрудника из памяти временного хранения.
После завершения цикла `for` ставшая ненужной память временного хранения возвращается в кучу функцией `free(name);` распечатывается строка с дефисами и в последующем цикле `for` распечатывается список сотрудников. После чего функцией `free(emp_ptr);` освобождается динамическая память, выделенная под массив указателей, и программа завершается.
В принципе большинство современных компиляторов автоматически освобождают динамически выделенную программе память при завершении работы программы, можно было бы понадеяться на это и не вызывать функции `free` внутри данного примера. Но в реальной жизни лучше перестраховаться, потому что без явных вызовов `free` можно нарваться на трудноуловимую ошибку, когда после нескольких успешных запусков этого примера вдруг посыпятся ошибки с сообщениями: “Недостаточно памяти”. Такая ситуация называется "**утечкой памяти**".
Функции `exit` аварийно завершают выполнение программы, передавая операционной системе свои коды завершения (аналогично ей можно просто завершить выполнение программы выйдя из функции `main` при помощи оператора `return`).

Операторы `new` и `delete`

В диалекте C++ языка Си остаются доступными для использования функций `malloc`, `calloc`, `realloc`, `free`, находящиеся там в библиотеке `<cstdlib>`. Дополнительно напомним разницу в правилах именования и использования стандартных библиотек C и C++:

- у стандартных библиотек в C++ остаются те же имена, что и в C, только спереди к этим именам добавляется буква `c`. В нашем случае это: `<cstdlib>` вместо `<stdlib>`;
- формат строки подключения стандартной библиотеки к программе: `#include <cstdlib>` вместо `#include <stdlib.h>`, обратите внимание на отсутствие расширения имени файла `.h`).

Однако в последнее время программисты C/C++ практически полностью перешли от использования этих стандартных функций к более безопасному и удобному способу выделения и освобождения динамической памяти. Поэтому, хотя наш курс и делает упор на “традиционный” стиль программирования C, чтобы показать фундамент языка, рассказать об операторах `new` и `delete` мы сочли уместным именно здесь, тем более что никаких не рассмотренных нами ранее сложных понятий и особенностей языка C++, которые выгодно отличают его от традиционного ANSI C, для понимания приводимых далее примеров не потребуется.
Итак, память из кучи можно выделять с помощью оператора `new`, а освобождать (возвращать обратно в пул свободной памяти) оператором `delete`. Вот основная форма (синтаксис) использования этих операторов:

```
pvar = new type;
delete pvar;
```

Здесь `type` – это спецификатор типа объекта, для которого Вы запрашиваете выделение памяти, а `pvar` – указатель на этот тип. В него оператор new возвращает указатель на динамически выделенную память, достаточную для хранения объекта типа `type` в случае успешного завершения выделения памяти.
В первом стандарте языка C++ оператор `new`, при невозможности удовлетворить запрос на выделение памяти, возвращал нулевой указатель подобно функциям `malloc` и `calloc`. Но в дальнейшем ситуация изменилась, и в подавляющем большинстве современных компиляторов при неудачной попытке выделения памяти оператор `new` генерирует исключительную ситуацию по умолчанию, а возвращение нулевого указателя остается в качестве возможной опции, о наличии которой сообщается в документации на компилятор.
Сейчас мы не будем подробно рассказывать о понятии исключительной ситуации, а коротко говоря – это динамическая ошибка, которую можно обработать определенным образом. Понятие исключительной ситуации вошло в стандарты языка C++, о которых мы говорили на самом первом занятии, и теперь любой компилятор, если о нем заявлено, что он соответствует Standard C++, обязан генерировать исключительную ситуацию, если оператор `new` не в состоянии удовлетворить запрос на

динамическое выделение памяти. Если Ваша программа не обрабатывает эту исключительную ситуацию, то выполнение программы завершается. Итог вроде бы плачевный для начинающих программистов, но в C++ не сложно переопределить оператор `new` так, чтобы он не генерировал исключительных ситуаций, а всегда возвращал нулевой указатель в случае ошибки.
Оператор `delete` освобождает ранее выделенную память, когда в ней пропадает необходимость. В случае вызова оператора `delete` с неправильным указателем чаще всего происходит разрушение системы динамического распределения памяти и крах программы.
В чем же преимущество операторов `new` и `delete` перед функциям `malloc` и `calloc`? В принципе, мы о них уже поговорили, теперь подытожим:

- оператор new автоматически выделяет требуемое для хранения объекта заданного типа количество памяти, т.е. отпадает необходимость использовать `sizeof`, как было в примерах, показанных ранее;
- оператор `new` автоматически возвращает указатель на заданный тип данных, т.е. не нужно выполнять явное приведение типов, которое было обязательным в примерах, показанных выше;
- оба оператора, и `new` и `delete`, можно перегружать, что намного облегчает реализацию Вашей собственной модели распределения динамической памяти;
- при выделении памяти при помощи `new` допускается инициализация объекта, для которого выделяется память, вот так: `pvar = new type(начальное_значение);`
- отпадает необходимость включать в Вашу программу заголовок `<cstdlib>`.

Теперь покажем примеры использования операторов `new` и `delete`, например выделение памяти для хранения целого и инициализации этой переменной

```
#include <iostream>
using namespace std;
int main() {
  int *p;
  p = new int(9); // начальное значение переменной равно 9
  if (!p) {
    cout << "Ошибка выделения памяти.\n";
    return 1;
  }
  cout << "Это целое, на которое указывает p: " << *p << "\n";
  delete p;   // Освобождение памяти
  return 0;
}
```

А также пример выделения памяти для хранения массива целых

```
#include <iostream>
using namespace std;
int main() {
  int *p;
  p = new int[5]; // Выделение памяти под массив из 5 целых
  if (!p) {
    cout << "Ошибка выделения памяти.\n";
    return 1;
  }
  for (int i = 0; i < 5; i++) p[i] = i;  // Заполнение массива
  for (int i = 0; i < 5; i++)   { // Распечатка содержимого массива
    cout << "Это целое, на которое указывает p[" << i << "] : ";
    cout << p[i] << "\n";
  }
  delete [] p;  // Обратите внимание на синтаксис оператора освобождения
```

```
памяти, выделенной под массив
 return 0;
}
```

В последнем примере можно было бы создать деструктор для удаления каждого из элементов массива при освобождении памяти, но оставим это до соответствующего занятия в будущем, где рассмотрим конструкторы и деструкторы объектов C++.

Особенности использования

Адрес - это номер ячейки оперативной памяти, содержащей данные, присвоенный в соответствии с некоторыми известными правилами. правила присвоения адресов определены архитектурой машины. Операция над адресами реализуют машинные команды и функции стандартной библиотеки.

Ссылка - имя, сопоставленное адресу, по которому уже хранятся данные, имеющие тип. Адрес, доступный по ссылке, строго постоянен для конкретного имени

Операция над ссылками реализует стандартная библиотека. С точки зрения языка они не отличаются от операций над переменными.

Указатель - созданная операцией разыменования переменная, имя или аргумент функции, хранящий адрес для доступа к некоторым данным

- предполагаемый тип указателя задается при его объявлении
- указатель с предполагаемым типом void предоставляет возможность унифицированного хранения адресов данных любых типов
- тип данных, сопоставляемый самому указателю, реализует то же множество операций, что и беззнаковое целое число unsigned short
- при доступе к данным через указатель не проверяется соответствие пространств имен (адрес, хранимый указателем, не является именем)
- адрес прямого доступа, полученный операцией взятия ссылки, записанной как rvalue, может быть присвоен указателю напрямую, без подстановки

Пустой указатель - указатель, который не хранит действительного адреса данных в оперативной памяти

- автоматическая проверка на действительность средствами стандартной библиотеки невозможна.
- попытка доступа к данным по недействительному указателю вызывает ошибку сегментирования
- для ручного контроля действительности указателя введены служебные константы NULL и nullptr

Статическая память - память, выделяемая с помощью специальных процедур, прописываемых транслятором в машинном коде программы

- адреса, выделяемые при загрузке программы, не меняются в процессе ее работы (статические адреса данных)
- под данные резервируется объем памяти, совпадающий с их размером хранения

Динамическая память - память вычислительной машины, использование которой определяется потребностями работающих с ней программ

- распределяется между программами, выполняемыми на машине, динамически
- требует реализации (на уровне стандартной библиотеки или операционной системы) механизма запроса и освобождения известного объема хранения

Указатель на функцию - адрес функции, записанный в переменную специального вида, которой сопоставлены:

- обобщенная операция вызова с известными типами аргументов

- предполагаемый тип возвращаемого значения

Особенности:

- присваивание указателям на функции идет аналогично присваиванию переменных. Требуется совпадение операций вызова и возврата
- указатель на функцию позволяет реализовать управляемую перегрузку имен, когда совпадают аргументы операций вызова, но не имена функций
- контроль действительности указателя на функцию возможен только средствами программы
- вызов функции по указателю идет без применения операции разыменования

Ввод-вывод данных в языке Си.

В любом языке ввод-вывод данных - это возможность читать их из файлов, получать их с устройств ввода (клавиатуры, мыши), и записывать данные в файлы, выводить их на различные устройства, например на принтер и, как минимум, на дисплей. Функции ввода/вывода в языке Си делятся на три класса:

- ввод/вывод верхнего уровня, с использованием понятия “поток”;
- ввод/вывод консольного терминала, путем непосредственного обращения к нему;
- ввод/вывод нижнего уровня, с использованием понятия “дескриптор”.

Мы рассмотрим здесь только верхний уровень, потому что это курс основ. Итак, понятие “поток”. Файл-заголовок в стандартной библиотеке ввода-вывода <stdio.h>, формат строки-включения в программу пользователя:

```
#include <stdio.h>
// или для C++
#include <iostream>
```

Поток можно представить себе как некоторый файл, который уже открыт, и имеет ассоциированные с ним буферы. При выполнении любой С-программы любая ОС (начиная с Unix) автоматически открывает для использования программой три потока:
stdin – стандартный поток ввода;
stdout – стандартный поток вывода, используется для вывода обычных результатов работы программы;
stderr – стандартный поток протокола, используется для вывода сообщений об ошибках, генерируемых системой.
Все три потока по умолчанию связаны с терминалом и клавиатурой, но с каждым из них может быть ассоциирован любой файл. Вот основные стандартные библиотечные функции для работы с потоками:

- scanf - ввод данных с форматированием из стандартного потока ввода;
- getchar - ввод символа из стандартного потока ввода;
- getstr - ввод строки из стандартного потока ввода;
- printf - вывод данных с форматированием в стандартный поток вывода;
- putchar - вывод символа в стандартный поток вывода;
- puts - вывод строки в стандартный поток вывода.

На предыдущих занятиях мы уже использовали некоторые из этих функций для ввода с клавиатуры и вывода на экран. Прототипы этих функций смотри в <stdio.h>.

Работа с файлами.

Файлы-заголовки в стандартной библиотеке ввода-вывода <stdio.h>, <io.h>. Для включения в программу пользователя необходимо выполнить директиву:

```
#include <stdio.h>
#include <io.h>
// для C++
#include <fstream>
```

```
// внутри которого содержатся ifstream, ofstream - потоки ввода-вывода
```

В отличие от стандартных потоков ввода/вывода, которые уже открыты, когда программа начинает выполняться, файлы требуют некоторых подготовительных действий и допускают не только последовательный доступ к содержащимся в них данным, но и доступ в произвольном порядке к данным в разных частях файла. Кроме того, по завершении работы с ним, файл должен быть закрыт.
Библиотека stdio содержит структуру типа данных `FILE` (объявленную с помощью typedef). Там же определена и константа `EOF`, которая возвращается некоторыми из функций как указание о достижении конца файла. Если рассматривать работу с файлами при помощи стандартной библиотеки языка С, то станет очевидно удобство такой работы, все функции для работы с файлами называются точно также, как мы привыкли но с префиксом `f` (`fprintf`, `fscanf`). Исключение составляет функция `fopen`, открывающая файл по его относительному пути. В С++ же для работы с файлами открываются дополнительные потоки, например `ifstream` и далее используются привычные угловые скобки для записи в поток `fin << "Hello";`

Практическое задание

1. Написать программу, которая создаст два текстовых файла, примерно по 50-100 символов в каждом (особого значения не имеет);
2. Написать функцию, «склеивающую» эти файлы, предварительно буферизуя их содержимое в динамически выделенный сегмент памяти нужного размера.
3. * Написать программу, которая проверяет присутствует ли указанное пользователем при запуске программы слово в указанном пользователем файле (для простоты работаем только с латиницей).

Используемые источники

1. *Брайан Керниган, Деннис Ритчи.* Язык программирования С. — Москва: Вильямс, 2015. — 304 с. — ISBN 978-5-8459-1975-5.
2. *Stroustrup, Bjarne* The C++ Programming Language (Fourth Edition)