КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 22.1 — Ввод-вывод в C++

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Базовый ввод/вывод....3
Стандартный вывод (cout)....4
Стандартный ввод (cin)....8
getline — Получить строку из потока в строку <string>....13
stringstream....14
Файловый ввод/вывод....16
Открыть файл....17
Закрытие файла....20
Текстовые файлы....21
Проверка флагов состояния....23
Получить и установить позицию в потоке....24
Двоичные (бинарные) файлы....28
Буферизация и синхронизация....30

Базовый ввод/вывод

Стандартная библиотека предоставляет множество дополнительных способов взаимодействия с пользователем с помощью функций ввода/вывода.

C++ использует удобную абстракцию, называемую потоками, для выполнения операций ввода и вывода на последовательных носителях, таких как экран, клавиатура или файл.

Поток — это объект, в который программа может вставлять символы или извлекать из него.

Знать подробности о носителях, связанных с потоком, или о каких-либо его внутренних спецификациях нет необходимости. Все, что нам нужно знать, это то, что потоки являются источником/приемником символов и что эти символы предоставляются/принимаются последовательно (то есть один за другим).

Стандартная библиотека определяет несколько потоковых объектов, которые могут использоваться для доступа к тому, что считается стандартными источниками и приемниками символов в среде, в которой выполняется программа:

Поток	Описание
`cin`	Поток стандартного ввода
`cout`	Поток стандартного вывода
`cerr`	Поток стандартного вывода ошибок
`clog`	Поток стандартного вывода журнала

Мы пока будем рассматривать подробно только **cout** и **cin** (стандартные потоки вывода и ввода). **cerr** и **clog** также являются потоками вывода, поэтому по сути они работают как **cout**, с той лишь разницей, что они идентифицируют потоки для определенных целей — сообщения об ошибках и ведение журнала, которые во многих случаях в большинстве настроек среды фактически делают одно и то же — они печатают на экране, хотя их также можно индивидуально перенаправить.

Стандартный вывод (cout)

В большинстве программных сред стандартным выводом по умолчанию является экран, а объект потока C++, определенный для доступа к нему, — **cout**.

Для операций форматированного вывода **cout** используется вместе с оператором вставки, который записывается как **<<** (выглядит, как оператор сдвига влево).

```
cout << "Текст для вывода"; // печатает на экране «Текст для вывода»
cout << 120;                // печатает на экране число 120
cout << x;                  // печатает на экране значение переменной x
```

Оператор **<<** вставляет данные, следующие за ним, в поток, который ему предшествует.

В приведенных выше примерах он вставлял строку-литерал «**Текст для вывода**», число **120** и значение переменной **x** в стандартный выходной поток **cout**.

Предложение в первом операторе заключено в двойные кавычки (**"**), потому что это строковый литерал, а в последнем **x** — нет.

Два этих случая отличаются наличием кавычек — когда текст заключен между ними, текст печатается буквально, в противном случае текст интерпретируется как идентификатор переменной, и вместо этого печатается ее значение.

Пример

```
cout << "Hello";  // печатает текст «Hello»
cout << Hello;    // печатает содержимое переменной Hello
```

Несколько операций вставки (**<<**) могут быть объединены в один оператор:

```
cout << "Это " << "один оператор C++";
```

Этот оператор будет печатать текст «Это один оператор C++».

Сцепление операторов вставки полезно для смешивания литералов и переменных в одном операторе:

```
cout << "Васе " << age << " лет и его личный номер " << personal_number;
```

Предполагая, что переменная **age** содержит значение **25**, а переменная **personal_number** содержит **123456789012345**, вывод будет следующим:

```
Васе 25 лет и его личный номер 123456789012345
```

Что **cout** не делает автоматически, так это добавления переводов строк в конце, если это не указано явно. Например,

```
cout << "Это предложение.";
cout << "Это еще одно предложение.";
```

Выдаст

```
Это предложение.Это еще одно предложение.
```

Чтобы вставить разрыв строки, необходимо вставить символ новой строки в том месте, где должна быть строка переведена.

В C++ символ новой строки, как и в C, может быть указан как **'\n'**. Например:

```
cout << "Это предложение.\n";
cout << "Это еще одно предложение.\nЕще одно." << '\n' << "И еще одно...\n";
```

Что напечатает

```
Это предложение.
Это еще одно предложение.
Тще одно.
И еще одно...
$
```

В качестве альтернативы для разрыва строк можно использовать манипулятор[1] **endl**.

Например:

```
cout << "Это предложение." << endl;
cout << "Это еще одно предложение. << endl";
```

1 Манипуляторы – это глобальные функции, предназначенные для использования вместе с операторами вставки (<<) и извлечения (>>), выполняемыми для объектов потока iostream. Обычно они изменяют свойства и настройки форматирования потоков.

Манипулятор **endl** создает символ новой строки точно так же, как при вставке **'\n'**, но он также имеет дополнительное поведение — выполняет принудительную запись буферов потока (если они есть) на устройство вывода.

Таке поведение влияет, в основном, на полностью буферизованные потоки, а **cout**, как правило, не является полностью буферизованным. Поэтому, обычно рекомендуется использовать **endl** только тогда, когда необходимо сбросить данные из буфера на устройство, в противном же случае имеет смысл использовать **'\n'**.

Операция освобождения буфера (flush) влечет за собой определенные накладные расходы, и на некоторых устройствах может вызывать заметную задержку.

Стандартный ввод (cin)

В большинстве программных сред стандартным вводом по умолчанию является клавиатура, а объект потока C++, определенный для доступа к нему, — это **cin**.

Для операций форматированного ввода **cin** используется вместе с оператором извлечения, который записывается как **>>** (выглядит, как оператор сдвига вправо). Затем за этим оператором следует переменная, в которую будут помещены извлеченные данные. Например:

```
int age;
cin >> age;
```

Первый оператор объявляет переменную типа **int** с именем **age**, а второй извлекает из **cin** значение, которое будет в нем храниться.

Данная операция заставляет программу ждать ввода из **cin** и, как правило, это означает, что программа будет ждать, пока пользователь введет некоторую последовательность с клавиатуры.

В этом случае (!!!), что символы, введенные с помощью клавиатуры, передаются в программу только при нажатии клавиши Enter (или Return). Как только будет встречен оператор с операцией извлечения из **cin**, программа будет ждать столько времени, сколько потребуется, пока не будет предоставлен какой-либо ввод.

Операция извлечения из **cin** использует тип переменной после оператора **>>**, чтобы определить, как он интерпретирует символы, считанные из ввода. Если это целое число, ожидаемый формат — это серия цифр, если строка — последовательность символов и т. д.

```
// i/o example

#include <iostream>
using namespace std;

int main () {

  int i;
  cout << "Введите целочисленное значение: ";
  cin >> i;
  cout << "Вы ввели значение " << i;
  cout << " его половина равна " << i/2 <<", вернее, "<< i/2.0 << ".\n";
  return 0;
}
```

На консоли имеем

```
Введите целочисленное значение: 702
Вы ввели значение 123; его половина равна 61, вернее, 61.5.
```

Как можно видеть, извлечение из **cin** делает задачу получения входных данных из стандартного ввода довольно простой и понятной.

Но у этого метода есть и большой недостаток.

Что будет, если пользователь вводит что-то такое, что не может быть интерпретировано как целое число? В этом случае операция извлечения не выполняется. И это, по умолчанию, позволяет программе продолжать работу без установки значения для переменной **i**, что дает неопределенные результаты, если значение **i** используется позже.

Это очень плохое поведение программы. Ожидается, что большинство программ будут вести себя ожидаемым образом независимо от типа пользователя, соответствующим образом обрабатывая недопустимые значения. Только очень простые программы должны полагаться на значения, извлеченные непосредственно из **cin** без дополнительной проверки.

Чуть позже мы увидим, как можно использовать строковые потоки, чтобы лучше контролировать ввод данных пользователем.

Извлечения в **cin** также можно связать, чтобы запросить более одного элемента данных в одном операторе:

```
cin >> a >> b;
```

This is equivalent to:

```
cin >> a;
cin >> b;
```

При извлечении из **cin** пробелы (пробелы, табуляции, символ перевода строки …) всегда рассматриваются как завершающие извлекаемое значение, и, таким образом, извлечение строки означает всегда извлекать одно слово, а не фразу или целое предложение.

Чтобы получить всю строку из **cin**, существует функция **getline( )**, которая принимает поток (**cin**) в качестве первого аргумента, а строковую переменную — в качестве второго.

Пример

```
// Ввод из cin в strings
#include <iostream>
#include <string>
using namespace std;

int main () {

  string mystr;

  cout << "С кем имею честь? ";
  getline(cin, mystr);
  cout << "Рады приветствовать Вас, " << mystr << ", в нашем заведении.\n";
  cout << "Что изволите подать? ";
  getline(cin, mystr);
  cout << "О, " << mystr << " -- это прекрасный выбор!\n";
  return 0;
}
```

В консоли будет что-то похожее на

```
Как Вас зовут? Ганс Абрамович
Рады приветствовать Вас, Ганс Абрамович, в нашем заведении.
Что изволите подать? сто грамм и огурчик
О, сто грамм и огурчик -- это прекрасный выбор!
```

В обоих вызовах **getline()** использовался один и тот же идентификатор строки (**mystr**).

Во втором вызове оператор извлечения заменяет предыдущий контент новым, который был ему представлен.

Стандартное поведение, которое большинство пользователей ожидает от консольной программы, заключается в том, что каждый раз, когда программа запрашивает у пользователя ввод, пользователь вводит поле и затем нажимает Enter (или Return). То есть обычно ожидается, что ввод в консольных программах будет происходить в виде строк.

Этого можно достичь, используя **getline()** для получения ввода от пользователя. Поэтому, если нет веских причин этого не делать, для ввода данных в консольных программах вместо извлечения из **cin** нужно всегда использовать **getline()**.

getline — Получить строку из потока в строку <string>

(1)

```
istream& getline(istream&  is, string& str, char delim);
istream& getline(istream&& is, string& str, char delim);
```

(2)

```
istream& getline(istream&  is, string& str);
istream& getline(istream&& is, string& str);
```

Извлекает символы из входного потока **is** и сохраняет их в **str** до тех пор, пока не будет найден разделительный символ **delim** (или символ новой строки **'\n'** для варианта (2)).

Извлечение также останавливается, если достигается конец файла или если во время операции ввода возникает другая ошибка.

Если разделитель найден, он извлекается и отбрасывается (т.е. он не сохраняется, и следующая операция ввода начнется после него).

Следует обратить внимание, что любое содержимое, которое было в **str** перед вызовом, заменяется вновь извлеченной последовательностью.

Каждый извлеченный символ добавляется к строке, как если бы был вызван ее член **push_back()**.

Возвращаемое значение

То же самое, что и параметр **is**.

stringstream

Стандартный заголовок **<sstream>** определяет тип, называемый **stringstream**, который позволяет обрабатывать строку как поток и, таким образом, разрешает операции извлечения или вставки из/в строки так же, как они выполняются для **cin** и **cout**. Эта функция наиболее полезна для преобразования строк в числовые значения и наоборот. Например, чтобы извлечь целое число из строки, мы можем написать:

```
string mystr("1204");
int myint;
stringstream(mystr) >> myint;
```

Объявляются строка типа **string**, которая инициализируется значением «1204», и переменная типа **int**. Затем третья строка использует эту переменную для извлечения из строкового потока, который создан из строки.

Этот фрагмент кода сохраняет числовое значение 1204 в переменной **myint**.

В следующем примере числовые значения извлекаются из стандартного ввода косвенно — вместо извлечения числовых значений непосредственно из **cin**, из него вводятся строки в строковый объект **mystr**, а затем значения из этой строки извлекаются в переменные **price** и **quantity**. Если это числовые значения, с ними можно выполнять арифметические операции, например, умножать их для получения общей цены.

При таком подходе получения целых строк и извлечения их содержимого процесс получения пользовательского ввода отделяется от интерпретации ввода как данных, что позволяет получить больший контроль над преобразованием его содержимого в данные.

```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main () {

  string mystr;
  float price  = 0;
  int quantity = 0;

  cout << "Введите цену: ";
  getline(cin, mystr);
  stringstream(mystr) >> price;
  cout << "Введите количество: ";
  getline(cin, mystr);
  stringstream(mystr) >> quantity;
  cout << "Полная стоимость: " << price * quantity << endl;
  return 0;
}
```

В консоли что-то похожее на

```
Введите цену: 123.45
Введите количество: 3
Полная стоимость: 370.35
```

Файловый ввод/вывод

C++ предоставляет следующие классы для вывода и ввода символов в/из файлов:
ofstream: потоковый класс для записи в файлы
ifstream: потоковый класс для чтения из файлов
fstream: потоковый класс для чтения и записи из/в файлы.

Эти классы прямо или косвенно являются производными от классов **istream** и **ostream**.

Мы уже использовали объекты, типы которых были этими классами: **cin** — это объект класса **istream**, а **cout** — объект класса **ostream**. Поэтому мы уже использовали классы, связанные с нашими файловыми потоками. Фактически, мы можем использовать наши файловые потоки так же, как мы уже привыкли использовать **cin** и **cout**, с той лишь разницей, что мы должны связать эти потоки с физическими файлами.

Пример

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
}
```

Этот код создает файл с именем **example.txt** и вставляет в него предложение так же, как мы привыкли делать с **cout**, но вместо этого использует файловый поток **myfile**.

Открыть файл

Первая операция, обычно выполняемая над объектом одного из классов **ofstream**, **ifstream**, **fstream**, — это связать его с реальным файлом. Эта процедура называется открытием файла.

Открытый файл представлен в программе потоком, т.е. объектом одного из этих классов (например, **ofstream myfile** в предыдущем примере), и любая операция ввода или вывода, выполненная с этим объектом потока, будет применена к физическому файлу, связанному с ним.

Чтобы открыть файл с объектом потока, мы используем его функцию-член **open()**:

```
open(filename, mode);
```

Где **filename** — это строка, представляющая имя файла, который нужно открыть, а **mode** — необязательный параметр с комбинацией следующих флагов:

ios::in открыть для операций ввода;

ios::out открыть для операций вывода;

ios::binary открыть в бинарном режиме;

ios::ate установить начальную позицию в конце файла. Если этот флаг не установлен, начальная позиция — это начало файла.

ios::app Все операции вывода выполняются в конце файла, добавляя содержимое к текущему содержимому файла.

ios::trunc Если файл открывается для операций вывода и он уже существует, его предыдущее содержимое удаляется и заменяется новым.

Все эти флаги можно комбинировать с помощью побитового оператора OR (|). Например, если необходимо открыть файл **example.bin** в двоичном режиме для добавления данных, следует это сделать с помощью следующего вызова функции-члена **open()**:

```
ofstream myfile;
myfile.open("example.bin", ios::out | ios::app | ios::binary);
```

Для сравнения, то же самое в C

```
FILE *myfile = fopen(“example.bin”, “ab”);
```

Каждая из открытых функций-членов классов **ofstream**, **ifstream** и **fstream** имеет режим по умолчанию, который используется, если файл открывается без второго аргумента:

Класс	Режим по умолчанию
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Для классов **ifstream** и **ofstream** автоматически, соответственно, предполагаются **ios::in** и **ios::out**, даже если режим, который их не включает, передается в качестве второго аргумента функции-члену **open()** (флаги объединяются).

Для **fstream** значение по умолчанию применяется только в том случае, если функция вызывается без указания какого-либо значения для параметра режима. Если функция вызывается с каким-либо значением в этом параметре, режим по умолчанию не объединяется, а переопределяется.

Файловые потоки, открытые в двоичном режиме, выполняют операции ввода и вывода независимо от каких-либо форматных соображений.

Небинарные файлы известны как текстовые, и из-за необходимого форматирования некоторых специальных символов (например, символов новой строки и возврата каретки) могут происходить некоторые перемещения.

Поскольку первая задача, которая выполняется для файлового потока, как правило, заключается в открытии файла, эти три класса включают конструктор, который автоматически вызывает функцию-член **open()** и имеет те же параметры, что и этот член. Следовательно, мы могли также объявить предыдущий объект **myfile** и провести ту же операцию открытия в нашем предыдущем примере, написав следующее:

```
if (myfile.is_open()) {
  /* Отлично, продолжаем работу */
}
```

или

```
if (!myfile.is_open()) {
 /* обрабатываем ошибку */
}
/* Отлично, продолжаем работу */
...
```

Закрытие файла

Когда операци ввода и вывода с файлом будут выполенны, его следует закрыть, чтобы операционная система получила соответсвующее уведомление и ресурсы, выделенные для работы с этим файлом, снова стали доступными.

Для этого следует вызвать функцию-член потока **`close()`**.

Эта функция-член очищает связанные с потоком буферы и закрывает файл:

```
myfile.close();
```

После вызова этой функции-члена объект потока можно повторно использовать для открытия другого файла, а предыдущий файл снова становится доступным для открытия другими процессами. В случае, если объект уничтожается, но все еще остается связанным с открытым файлом, деструктор автоматически вызывает функцию-член **`close()`**.

Текстовые файлы

Потоки текстовых файлов — это такие потоки, для которых в режим открытия не был включен флаг **ios::binary**.

Такие файлы предназначены для хранения текста, и поэтому все значения, которые вводятся или выводятся из/в них, могут подвергаться некоторым преобразованиям форматирования, которые не обязательно соответствуют их буквальному двоичному значению.

Операции записи в текстовые файлы выполняются так же, как мы работали с **cout**:

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {

  ofstream myfile("example.txt");           // Пробуем открыть файл
  if (myfile.is_open()) {                   // Если удалось
      myfile << "This is a line.\n";        //   Пишем что-нибудь
      myfile << "This is another line.\n";  //   в файл
      myfile.close();                       //   и закрываем его
  } else                                    // в случае неудачи с открытием
      cout << "Unable to open file";        //   сообщаем об этом
  return 0;                                 // и выходим
}
```

Чтение из файла выполняется аналогично чтению из **cin**:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {

  string line;
  ifstream myfile("example.txt");     // открываем в конструкторе
  if (myfile.is_open()) {             // если удалось
      while (getline(myfile, line)) { //   выводим все строки в цикле
          cout << line << '\n';       //   в консоль
      }
      myfile.close();                 //   и закрываем
  } else                              // если не удалось
      cout << "Unable to open file";  //   сообщаем о проблеме

  return 0;
}
```

В этом примере выполняется чтение текстового файла и вывод его содержимого в консоль.

Цикл **while** считывает файл построчно, используя **getline()**. Значение, возвращаемое функцией **getline()**, является ссылкой на сам объект потока, который при оценке как логическое выражение является истинным, если поток готов для дополнительных операций, и ложным, если был достигнут конец файла или произошла какая-либо другая ошибка.

Проверка флагов состояния

Для проверки определенных состояний потока (все они возвращают значение типа **bool**) существуют следующие функции-члены:

bad() — возвращает **true**, если операция чтения или записи завершилась неудачно. Например, в случае, если мы пытаемся писать в файл, который не открыт для записи, или если на устройстве, на которое мы пытаемся записать, не осталось места.

fail() — возвращает **true** в тех же случаях, что и **bad()**, но также в случае ошибки формата, например, когда извлекается алфавитный символ, в то время как мы пытаемся прочитать целое число.

eof() — возвращает **true** , если для файла, открытого для чтения, достигнут конец.

good() — это наиболее общий флаг состояния — он возвращает **false** в тех же случаях, когда вызов любой из предыдущих функций вернул бы **true**.

Обратите внимание, что **good()** и **bad()** не являются точными противоположностями — **good()** проверяет сразу несколько флагов состояния.

Для сброса флагов состояния можно использовать функцию-член **clear()**.

Получить и установить позицию в потоке

Все объекты потоков ввода-вывода сохраняют внутренне по крайней мере, одну внутреннюю позицию:

ifstream, как и **istream**, хранит элемента позицию внутри потока, который будет считан в следующей операции ввода.

ofstream, как и **ostream**, хранит позицию внутри потока, куда должен быть записан следующий элемент.

fstream, как и **iostream**, сохраняет обе позиции — получения (get) и размещения (put).

Эти позиции внутри потока указывают на места в потоке, где выполняется следующая операция чтения или записи. Эти позиции можно узнать и изменять с помощью следующих функций-членов:

tellg() и **tellp()** — эти две функции-члены без параметров возвращают значение элемента с типом **streampos**, который является типом, представляющим текущую позицию получения (в случае **tellg()**) или позицию размещения (в случае **tellp()**).

seekg() и **seekp()** — эти функции позволяют изменять расположение позиций получения и размещения. Обе функции перегружены двумя разными прототипами. Первая форма:

```
seekg(streampos position);
seekp(streampos position);
```

С помощью этих прототипов, указатели в потоке перемещаются в позицию **position**, считая с начала файла. Тип этого параметра — **streampos**, который является тем же типом, что и возвращаемый функциями **tellg()** и **tellp()**.

Другая форма для этих функций:

```
seekg(streamoff offset, seekdir direction);
seekp(streamoff offset, seekdir direction);
```

С помощью этих прототипов, для позиции получения или размещения устанавливается значение **offset** — смещение относительно некоторой конкретной точки, определяемой параметром **direction**.

offset имеет тип **streamoff**.

direction имеет тип **seekdir**, который является перечислимым типом, определяющим точку, от которой отсчитывается смещение. **direction** может принимать любое из следующих значений:

ios::beg	смещение, отсчитываемое от начала потока
ios::cur	смещение, отсчитываемое от текущего положения
ios::end	смещение, отсчитываемое от конца потока

В следующем примере для получения размера файла используются только что рассмотренные нами функции-члены.

Пример — получение размера файла

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {

  streampos begin, end;
  ifstream myfile("example.bin", ios::binary);
  begin = myfile.tellg();                          // получить позицию записи
  myfile.seekg(0, ios::end);                       // сместиться на конец файла
  end = myfile.tellg();                            // получить позицию записи
  myfile.close();
  cout << "size is: " << (end - begin) << " bytes.\n";
  return 0;
}
```

Обратите внимание на тип, который мы использовали для переменных **begin** и **end**:

```
streampos size;
```

streampos — это особый тип, используемый для позиционирования по буферам и файлам.

Также это тип, возвращаемый **file.tellg()**.

Значения этого типа могут быть безопасно вычитаться из других значений того же типа, а также могут быть преобразованы в целочисленный тип, достаточно большой, чтобы содержать размер файла.

Вышеуказанные функции позиционирования в потоке используют два конкретных типа: **`streampos`** и **`streamoff`**.

Эти типы также определены как типы-члены класса **`stream`**:

Тип	**Тип члена**	**Описание**
`streampos`	`ios::pos_type`	Определен, как `fpos<mbstate_t>`. Он может преобразовываться в/из **`streamoff`**, также можно складывать или вычитать значения этих типов.
`streamoff`	`ios::off_type`	Это псевдоним одного из основных целочисленных типов (например, **`int`** или **`long long`**).

Каждый из перечисленных выше типов членов является псевдонимом своего эквивалента, не являющегося членом (они одного и того же типа).

Неважно, какой из них используется.

Типы членов являются более общими, поскольку они одинаковы для всех объектов потока (даже для потоков, использующих экзотические типы символов).

Типы, не являющиеся членами, широко используются в существующем коде по историческим причинам.

Двоичные (бинарные) файлы

Для двоичных файлов чтение и запись данных с помощью операторов извлечения и вставки (**<<** и **>>**), а также таких функций, как **getline()**, неэффективны и бессмысленны, поскольку нам не нужно форматировать какие-либо данные. Да и скорее всего, данные не отформатированы в виде строк.

Файловые потоки включают две функции-члены, специально разработанные для последовательного чтения и записи двоичных данных — **write()** и **read()**.

write() — это функция-член **ostream** (унаследованной от **ofstream**).

read() — это функция-член **istream** (унаследованная от **ifstream**).

У объектов класса **fstream** есть и та, и другая. Их прототипы:

```
write(char *memory_block, size_t size);
 read(char *memory_block, size_t size);
```

Здесь **memory_block** имеет тип **char *** и представляет адрес массива байтов, в котором сохраняются элементы данных для чтения или откуда берутся элементы данных для записи.

Параметр **size** — это целочисленное значение, указывающее количество символов, которые должны быть прочитаны или записаны из/в блок памяти.

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {

  streampos size;
  char *memblock;

  ifstream file("example.bin", ios::in|ios::binary|ios::ate); // чтение с конца
  if (file.is_open()) {
      size     = file.tellg();    // получаем размер файла
      memblock = new char[size];  // просим выделить массив соотв. размера
      file.seekg(0, ios::beg);    // смещаемся в начало
      file.read(memblock, size);  // читаем весь файл в память
      file.close();               // закрываем файл

      cout << "the entire file content is in memory";

      delete[] memblock;
  } else
      cout << "Unable to open file";
  return 0;
}
```

В этом примере считывается весь файл и сохраняется в блоке памяти.

Файл открывается с флагом **ios::ate**, т.е. указатель получения будет помещен на конец файла. Таким образом, когда мы вызываем член **tellg()**, мы напрямую получаем размер файла.

Буферизация и синхронизация

Когда мы имеем дело с файловыми потоками, они связаны с внутренним буферным объектом типа **`streambuf`**. Этот буферный объект предоставляет блок памяти, который действует как посредник между потоком и физическим файлом. Например, с помощью **`ofstream`** каждый раз, когда вызывается функция-член **`put()`**, записывающая один символ, символ может быть записан в этот буфер вместо записи его непосредственно в физический файл, с которым связан поток.

Операционная система также может определять свои уровни буферизации для файловых операций чтения и записи. Когда буфер выходного потока очищается, все содержащиеся в нем данные записываются на физический носитель. Этот процесс называется *синхронизацией* и происходит при любом из следующих обстоятельств:

Когда файл закрывается — перед закрытием файла все буферы, которые еще не были очищены, синхронизируются, и все ожидающие данные записываются на физический носитель.

Когда буфер заполняется — буферы имеют определенный размер. Когда буфер заполняется, он синхронизируется автоматически.

Явно манипуляторами — когда в потоках используются определенные манипуляторы, происходит явная синхронизация. Это манипуляторы **`flush`** и **`endl`**.

Явно при вызове функции-члена **`sync()`** — вызов этой функции для потока вызывает немедленную синхронизацию.

`sync()` возвращает значение типа **`int`**, равное -1, если поток не имеет связанного с ним буфера или в случае сбоя.

В противном случае (если буфер потока был успешно синхронизирован) возвращается 0.