Лекция VII

11 февраля 2017

Файлы. Потоковый ввод/вывод

Стандартные функции для работы с файлами определены в заголовочном файле <**fstream**>. Основными являются три класса (типа данных):

- **ifstream** - класс для *чтения* информации из файла.
- **ofstream** - класс для *записи* информации из файла.
- **fstream** - класс для одновременных операций как *чтения* информации, так и *записи* её в файл.

Файлы. Потоковый ввод/вывод

Объявление переменных для файлового потока.

1. Конструктор без параметров: *ifstream*{}

```
1 ifstream config_file1;
```

2. Путь к файлу (строка в стиле "C") и режим работы с ним: *ifstream*{*char* ∗ *str*, *file_mode*}

```
1 ifstream config_file2{"config.dat"};
```

3. Путь к файлу (строка типа **string**) и режим работы с ним: *ifstream*{*string* *str*, *file_mode*}

```
1 string file_name = "C:\Documents\user\↩
    my_config.txt";
2 ifstream config_file3{file_name};
```

file_mode - установка специальных флагов, управляющих режимом открытия файла.

Установка отложенной связи с файлом - функция **open**. Аргументы - аналогичны конструкторам с параметрами: строка, указывающая путь к файлу, и комбинация специальных флагов.

```
1 ifstream config_file1;
2
3 //  ...
4
5 config_file1.open("config.dat");
```

Режим открытия файла.

Флаг	Для чего нужен
std::ios_base::ate	При открытии текущая позиция потока устанавливается в конец файла
std::ios_base::app	Операции вывода начинаются с конца файла, то есть происходит дозапись
std::ios_base::binary	Операции ввода/вывода происходят в двоичном режиме
std::ios_base::out	Открыть поток на запись
std::ios_base::in	Открыть поток на чтение
std::ios_base::trunc	При открытиии файла удалить всё его содержимое

Режим открытия файла.

```
1 #include <fstream>
2 using namespace std;
3
4 // Открытие на запись
5 // в конец файла в бинарном режиме
6 ofstream out_file{"my_file.txt",
7           ios_base::app | ios_base::binary};
```

Проверка готовности файлак операциям ввода/вывода.

1. Метод **is_open**

   `bool stream_var.is_open()`

```
1 ifstream input_file{"data_file.txt"};
2 if ( input_file.is_open() ) {
3   // Ошибок нет
4   // Совершаем операции с файлом
5 }
```

2. Непосредственное использование переменной потока в условном выражении

```
1 ifstream input_file{"data_file.txt"};
2 if ( input_file ) {
3   // Ошибок нет
4   // Совершаем операции с файлом
5 }
```

Файлы. Чтение данных известного формата

```
istream& stream_var.operator>>(<тип_данных>& value)
```

```
1 ifstream input_file{"data_file.txt"};
2 if ( input_file.is_open() ) {
3   double num;
4   input_file >> num;
5   cout << "Прочитано значение: " << num;
6 }
```

Файлы. Чтение данных известного формата

Способы проверки на отсутствие ошибок при вводе

- Проверка достижения конца файла

  `bool stream_var.eof()`
- Проверка успешности операций ввода/вывода

  `bool stream_var.bad()`
- Проверка успешности ввода данных

  `bool stream_var.fail()`

Файлы. Чтение данных известного формата

Пусть дан файл

```
45.657 6.88 10.56
5.456 8.9905 6.7 7.8 14.5
5.616 8.8888
```

```
1  ifstream input_file{"data_file.txt"};
2
3  if ( input_file.is_open() ) {
4    double num;
5
6    // Прочитать все числа из файла
7    // и напечатать их значения в консоли
8    while ( !input_file.eof() ) {
9      input_file >> num;
10     cout << '\n' << num;
11   }
12 }
```

Файлы. Чтение данных известного формата

Добавляем проверку на ошибки

```
1  ifstream input_file{"data_file.txt"};
2  if ( input_file.is_open() ) {
3    double num;
4
5    while ( !input_file.eof() ) {
6      input_file >> num;
7      cout << '\n' << num;
8    }
9
10   if ( input_file.bad() ) {
11     cout << "Ошибка операций ввода/вывода";
12   } else if ( input_file.fail() ) {
13     cout << "В файле содержатся нечисловые ↩
           данные";
14   }
15 }
```

Файлы. Чтение данных без форматирования

```
(1) int stream_var.get()
(2) istream& stream_var.get(char& symbol)
(3) istream& stream_var.get(char *str,
                            unsigned count)
(4) istream& stream_var.get(char *str,
                            unsigned count, char delim)
```

1. Читаем один символ из потока и возвращаем его код
2. Читаем один символ из потока и помещаем его в переменную **symbol**
3. Читаем как максимум **count - 1** символов и записываем их в переменную **str**. В **str** также добавляется символ окончания строки
4. Аналогично **(3)** с возможностью указать собственный разделитель с помощью переменной **delim**

Важно: символ-разделитель остаётся в потоке для следующих операций чтения данных.

Файлы. Чтение данных без форматирования

В файле:

```
Very important data
```

```
1  ifstream input_file{"data_file.txt"};
2  if ( input_file.is_open() ) {
3    char sym1, sym2;
4
5    sym1 = input_file.get();
6    input_file.get(sym2);
7
8    cout << sym2 << ' ' << sym1 << '\n';
9
10   char str[7];
11   input_file.get(str, 7);
12   // Печатает "ry imp"
13   cout << str << '\n';
14 }
```

Файлы. Чтение данных без форматирования

(1) istream& stream_var.unget()

1. Сделать последний прочитанный символ вновь доступным для извлечения из потока

```
1  ifstream input_file{"data_file.txt"};
2  if ( input_file.is_open() ) {
3    char sym1, sym2;
4
5    input_file.get(sym1);
6    input_file.unget();
7    input_file.get(sym2);
8
9    cout << (sym1 == sym2) << '\n';
10 }
```

Файлы. Чтение данных без форматирования

(1) `int stream_var.peek()`

1. Получить значение следующего символа, не извлекая его из потока

```
1  #include <cctype>
2
3  ifstream input_file{"data_file.txt"};
4  if ( input_file.is_open() ) {
5    int sym_code;
6
7    sym_code = input_file.peek();
8
9    if ( isdigit(sym_code) ) {
10     double num;
11     input_file >> num;
12     // ...
13   }
```

Файлы. Чтение данных без форматирования

```
(1) istream& stream_var.getline(char *str,
                unsigned count)
(2) istream& stream_var.getline(char *str,
                unsigned count, char delim)
```

1. Читаем как максимум **count - 1** символов и записываем их в переменную **str**. В **str** также добавляется символ окончания строки '\0'
2. Аналогично **(1)** с возможностью указать собственный разделитель с помощью переменной **delim**

Важно: символ-разделитель извлекается из потока и не учавствует в последующих операциях чтения данных.

Файлы. Чтение данных без форматирования

В файле:

```
Very important data
```

```
1  ifstream input_file{"data_file.txt"};
2  if ( input_file.is_open() ) {
3    char str[7];
4    input_file.getline(str, 7, ' ');
5    // Печатает "Very"
6    cout << str << '\n';
7
8    char sym;
9    input_file.get(sym);
10   // Печатает "i"
11   cout << sym << '\n';
12 }
```

Файлы. Чтение данных без форматирования

```
(1) istream& stream_var.read(char *str,
                             unsigned count)
```

1. Считывает **count** символов в переменную **str**. В отличие от методов **get** и **getline** символ окончания строки не добавляется.

```
1 ifstream input_file{"data_file.txt"};
2 if ( input_file.is_open() ) {
3   char buffer[31];
4   input_file.read(buffer, 30);
5
6   // Если нужна валидная С-строка, символ её
7   // окончания нужно добавлять самим
8   buffer[30] = '\0';
9 }
```

Файлы. Чтение данных

```
istream& stream_var.ignore(size_t count,
                           char delim)
```

- Метод **ignore** пропускает заданное количество символов (байт) из файла и оставляет их необработанными (то есть не происходит сохранение или преобразование извлечённых символов). Пропуск прекращается или по достижении считывания **count** символов, или при встрече символа-разделителя **delim**.
- Оба параметра метода - **count** и **delim** имеют значения по умолчанию: **count** равен единице, а разделитель **delim** специальному символу, означающему конец файла
- Если пропуск символов прекращается при нахождении разделителя, то он тоже извлекается из потока и не учавствует в дальнейших операциях чтения информации

Когда может быть полезен метод **ignore**?
Во многих программах для ввода начальных параметров более уместно использовать *конфигурационные файлы*, вместо ввода значений через консоль. Особенно это относится к вычислительным задачам: граничные условия при расчёте задач по вычислению различных интегралов или систем уравнений; количество частиц и параметры вроде температуры для задач термодинамики; размеры матриц в каких-нибудь вычислениях.
Конфигурационные файлы предпочтительней хотя бы тем, что при изменениях параметров быстрее и надёжнее поменять их в текстовом файле, чем каждый раз сосредотачиваться на консольном вводе.

Файлы. Чтение данных

Пример конфигурационного файла некой абстрактной вычислительной задачи:

```
Максимальное число итераций: 26
Количество строк матриц: 15
Количество столбцов матриц: 25
Количество слоёв: 5
Сила трения между слоями: -7.8
```

Файлы. Чтение данных

Что можно выделить из описания файла на предыдущем слайде?

- Есть повторяющаяся структура: описание параметра - двоеточие - значение
- Программе нужны значения комментарии-описания нужны для человека
- Комментарии нужны для человека

Файлы. Чтение данных

Для написания универсального разбора и пригодится метод **ignore**

```
1  const size_t PASS_COUNT = 500;
2  int     max_iter, cols, rows, layers_count;
3  double fric_force;
4
5  ifstream config_file{"config_file.dat"};
6  if ( config_file.is_open() ) {
7    // пропускаем символы до двоеточия
8    config_file.ignore(PASS_COUNT, ':');
9    // безопасно считываем первое значение
10   config_file >> max_iter;
11
12   config_file.ignore(PASS_COUNT, ':');
13   config_file >> cols;
14   // ... Остальные переменные — аналогично
15 }
```

Файлы. Чтение данных

Код со слайда **23** разбирает файл со слайда **21** со следующими особенностями:

- Разумно предположить, что более 500 символов в качестве описания параметра человеку будет просто лень набирать
- Перед вводом *каждого* числового значения ищется символ двоеточия
- После нахождения - считываем числовое значение в нужную переменную

Файлы. Чтение данных

```
void stream_var.clear()
```

- Метод **clear** позволяет вернуть поток в состояние, пригодное для новых попыток считывания данных.

На девятом слайде приводились три метода, которые проверяют, не случилось ли каких либо ошибок между объектом потока и файлом - **eof()**, **bad() fail()**. Если хотя бы один из этих методов возвращает **true**, то дальнейшие операции получения данных невозможны. Метод **clear** возвращает поток в такое состояние, что все методы проверки состояния начинают возвращать значение **false**. Это позволяет попробывать считать какие-нибудь данные снова. Далее приводится пример, когда **clear** может быть полезен.

Файлы. Чтение данных

Пусть дан файл, нужно считать все числа. Каждое число отделено пробелом, но могут попадаться и наборы нечисловых символов

```
45.657 6.88 6.7 7.856 14.5 asd 5.616fs 8.88 sdsf
```

```
1  const size_t IGNORE_COUNT = 1000;
2  ifstream input_file{"data_file.txt"};
3  if ( input_file.is_open() ) {
4    double num;
5    while ( !input_file.eof() ) {
6      input_file >> num;
7      if ( input_file.fail() ) {
8        input_file.clear();
9        input_file.ignore(IGNORE_COUNT, ' ');
10     }
11   }
12 }
```

Файлы. Чтение данных

При каждой операции чтения, неважно - форматный ввод или нет - поток запоминает, сколько символов было прочитано и с какого места будет происходить следующая операция извлечения данных. Другими словами, поток содержит внутри себя **специальный индикатор**, который сохраняет текущую позицию в байтах относительно начала файла. При открытии файла значение индикатора равно нулю, затем, по мере чтения, оно меняется на количество прочитанных символов (байт). Узнать текущее значение индикатора можно с помощью метода **tellg**:

```
long int stream_var.tellg()
```

- Как правило, данный метод возращает значение, совместимое с длинным целым
- Если поток находится в "аварийном"состоянии, то возращается отрицательное значение

Файлы. Чтение данных

Дан файл с текстом:

```
Hello smart students!
```

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4
5 string str;
6 istream file{"input_file.txt"};
7
8 // Считываем слово "Hello"
9 file >> str;
10 // Печатаем значение 5,
11 // соответствующее первому пробелу
12 cout << "Значение индикатора: " << file.↩
     tellg();
```

Файлы. Чтение данных

Замечательной возможностью операций работы с файлами является то, что индикатор позиции потока можно менять принудительно. Для этого используется метод **seekg** в 2-х формах:

```
(1) istream& stream_var.seekg(long int pos)
(2) istream& stream_var.seekg(long int off,
                              start_point)
```

1. Устанавливает индикатор на **pos** байт относительно начала файла
2. Устанавливает индикатор путём сдвига **off** относительно некоторой начальной точки **start_point**. **off** - может быть как положительным, так и отрицательным целым значением, а **start_point** может быть одной из трёх следующих констант, определённых в классе **ios_base**.

Файлы. Чтение данных

```
istream& stream_var.seekg(long int off,
                          start_point)
```

Константы для **start_point**

Константа	Что означает
std::ios_base::beg	Начало файла
std::ios_base::cur	Текущая позиция индикатора в файле (изменяется при операциях чтения)
std::ios_base::end	Конец файла

Файлы. Чтение данных

Пример: получение размера файла (подключение нужных библиотек - пропущено)

```
1  // Для того, чтобы узнать размер файла
2  // его нужно открывать в двоичном режиме
3  istream in_file{"some_file.txt", ios_base::↩
       binary};
4
5  if ( in_file.is_open() ) {
6    // Переводим индикатор в конец файла:
7    in_file.seekg(0, ios_base::end);
8    // Считываем количество байт:
9    long int byte_count = in_file.tellg();
10   // Везращаем индикатор в начало файла:
11   in_file.seekg(0);
12   cout << "Размер файла: " << byte_count << ↩
         " байт";
13 }
```

Файлы. Чтение данных

```
void stream_var.close()
```

- Метод **close** принудительно разрывает связь потокового объекта с реальным файлом
- Однако явный вызов данного метода не всегда необходим, потому что он вызывается неявно в тот момент, как только переменная выходит из области видимости

```
1 {
2   istream in_file{"some_file.txt"};
3   // операции чтения...
4   // in_file.close() // необязательно
5 }
6 // здесь переменная in_file уже недоступна
7 // и её связь с файлом some_file.txt
8 // уже прервана
```

Файлы. Запись данных

- Запись данных в файл осуществляется с помощью класса **ofstream** (потоковый класс, осуществляющий только операции вывода). Объекты этого класса связываются с файлом либо через конструктор, либо через функцию **open** (см. слайды 3-6).
- Аналогично классу **ifstream**, объекты **ofstream** имеют методы **is_open()**, **bad()** и **fail()**.
- Закрытие связи с выходным файлом осуществляется также, как указано на предыдущем слайде
- По умолчанию файл всегда создаётся заново, то есть если он существовал, то содержимое будет стёрто. Для предотвращения нужно пользоваться флагом **ios_base::app** (см. пятый слайд)

Файлы. Запись данных с форматированием

Аналогично операциям с консольным выводом через **cout** и происходит с помощью оператора:

```
ostream& stream_var.operator<<(<тип_данных>& value)
```

```
1 ofstream out_file{"data_file.txt"};
2
3 if ( out_file.is_open() ) {
4   // Вывод квадратов чисел от 1 до 100
5   for (unsigned i = 1; i <= 100; ++i) {
6     out_file << i << " * " << i << " = " << ↩
          i * i << '\n';
7   }
8 }
```

Файлы. Запись данных с форматированием

Больше примеров по форматированию вывода можно найти здесь:

`https://github.com/posgen/OmsuMaterials/wiki/Format-output`

В указанной справке все примеры приведены для консольного вывода с использованием **cout**, однако вывод в файл отформатированных данных полностью аналогичен с точностью до названия объекта вывода.

Файлы. Запись неформатированных данных

Запись данных без осуществления форматирования осуществляется с использованием типа **char**. Поскольку в C++ размер типа **char** равен 1 байту, фактически запись неформатированных данных означает побайтовый вывод в файл.

```
ostream& stream_var.put(char symbol)
```

- Метод **put** выводит символ **symbol** в файл
- Стоит отметить, что данный метод посимвольного вывода определён только для типа **char**. Не существует его перегрузки, например, для типа **unsigned char**

Файлы. Запись неформатированных данных

Пример: посимвольный вывод всех строчных букв английского алфавита в файл

```
1  ofstream out_file{"alphabet.txt"};
2
3  if ( out_file.is_open() ) {
4    for (char sym = 'a'; sym <= 'z'; ++sym) {
5      // Метод put возвращает ссылку на тот ↩
           объект,
6      // для которого он был вызван, что ↩
           позволяет
7      // строить цепочки, подобные следующей
8      out_file.put(sym).put('\n');
9    }
10 }
```

Файлы. Запись неформатированных данных

```
ostream& stream_var.write(char *str, size_t count)
```

- Метод **str** выводит **count** символов из строки **str** в файл

```
1  ofstream out_file{"strings.txt"};
2
3  if ( out_file.is_open() ) {
4    out_file.write("Some english string", 8);
5    out_file.put('\n');
6
7    char str[] = "Строка на русском языке";
8    // Вывод указанной выше строки в файл
9    out_file.write(str, sizeof(str));
10 }
```

Файлы. Запись неформатированных данных

Относительно необычный вывод всех байтов переменной типа **char** с помощью **write**. В текстовом файле - число в явном виде не появится!

```
1  ofstream out_file{"numbers.txt"};
2
3  if ( out_file.is_open() ) {
4    double num = 578.83445;
5    // Адрес переменной num — это указатель на↩
         double.
6    // Чтобы записать число через метод write
7    // его адрес надо привести к типу ↩
         указатель на char
8    // и записать все байты переменной num.
9    out_file.write( (char *)(&num), sizeof(num↩
         ));
10 }
```

Файлы. Запись данных

Аналогично классу **ifstream**, класс **ofstream** включает индикатор позиции в выходном файле. И им тоже можно управлять. Названия методов похожи, с точностью до последней буквы:

```
(1) ostream& stream_var.tellp()
(2) ostream& stream_var.seekp(long int pos)
(3) ostream& stream_var.seekp(long int off,
                              start_point)
```

1. Метод **tellp** фактически возвращает число записанных байт в файл
2. Данная форма метода **seekp** позволяет перемещать индикатор в произвольную позицию относительно начала файла
3. Перемещение индикатора относительно начала файла, конца файла или текущей позиции

Файлы. Запись данных

Как пример перемещения индикатора позиции при записи в файл - замена строчных букв на прописные

```
1  ofstream out_file{"text.txt"};
2
3  if ( out_file.is_open() ) {
4    out_file << "hello, my friends";
5    cout << "Записано байт: " << out_file.↩
       tellp();
6    out_file.seekp(0);
7    out_file << 'H';
8    out_file.seekp(10);
9    out_file << 'F';
10   out_file.seekp(0, ios_base::end);
11   out_file << '!';
12 }
13 // В файле текст: "Hello, my Friends!"
```

Файлы. Запись данных с форматированием

Справка по файловому вводу-выводу также доступна здесь:

https://github.com/posgen/OmsuMaterials/wiki/File-input-output