Языки программирования

Лекция 9

ПМИ 2 курс

Демяненко Я.М.

ЮФУ 2024

Ввод/вывод и работа с файлами

В языки C и C++ функции ввода/вывода **не встроены**.

Первоначально в языке C реализация ввода/вывода была оставлена на усмотрение разработчиков компиляторов. Практически для большинства компиляторов использовался набор функций, разработанный для среды UNIX.
По стандарту ANSI **C** этот набор с заголовочным файлом **cstdio** является обязательным компонентом стандартной библиотеки C.

В **C++** чаще используется ввод/вывод при помощи набора классов, определенных в заголовочных файлах **iostream** и **fstream**.

Ввод/вывод в C

При запуске C-программы операционная система открывает три файла и обеспечивает три файловые ссылки на них. Этими файлами являются: стандартный ввод, стандартный вывод и стандартный файл ошибок.

Соответствующие им указатели называются **stdin**, **stdout** и **stderr**. Они также описаны в **cstdio**.

Файловые указатели **stdin**, **stdout** и **stderr** представляют собой объекты типа **FILE***.

Это константы, а не переменные, следовательно, им нельзя ничего присваивать.

Обычно **stdin** соотнесен с клавиатурой, а **stdout** и **stderr** — с экраном.
Однако их можно связать с файлами.
Часто вывод **в stderr** отправляется на экран, даже если вывод **stdout** перенаправлен в другое место.

Ввод/вывод в C++

В **C++** чаще используется ввод/вывод при помощи набора классов, определенных в заголовочных файлах **iostream** и **fstream**.

Объекты cin и cout

Объекты **cin** и **cout** определены в заголовочном файле **iostream**

cin — объект класса **istream** и соответствует стандартному потоку ввода, связанному по умолчанию с клавиатурой

cout — объект класса **ostream** и соответствует стандартному потоку вывода, который по умолчанию связан с монитором

Объекты cerr и clog

Ещё двумя стандартно определенными потоковыми объектами являются **cerr** и **clog**.

Эти объекты используются для вывода отладочной информации и сообщений об ошибках, по умолчанию они тоже связаны с монитором.

Отличие между ними заключается в том, что поток **cerr** является небуферизованным, поэтому вся направляемая в этот поток информация сохранится в нем, даже если программа завершится аварийно.

Классы istream и ostream

Заголовочный файл **fstream** определяет классы **istream** и **ostream**, позволяющие работать с файлами как с потоками.

Текстовый и двоичный режимы

И средства языка C, и средства языка C++ предоставляют возможность работать с файлами в текстовом и двоичном режимах

С точки зрения операционной системы любой файл — это набор двоичных данных.

Определением, как интерпретировать эти данные, занимается прикладная программа.

Принято различать **два режима** интерпретации — **текстовый и двоичный**.

Текстовый и двоичный режимы. Отличия

Одно из отличий заключается в том, что в **текстовом** режиме **некоторые символы** (их двоичные коды) **обрабатываются особым образом** — это символы конца строки, перехода на новую строку, пробелы, символ табуляции.

В **двоичном** режиме все **символы равнозначны**.

Кроме того ввод/вывод данных в текстовом режиме может сопровождаться операцией преобразования из/в символьное представление.

В двоичном режиме преобразований не происходит.

Работа с текстовыми файлами в стиле C++

Пример. Создать текстовый файл, содержащий ведомость о результатах сдачи студентами трех экзаменов. Вывести содержимое этого файла на экран

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

 ofstream fout(filename);
 fout<<"Students\n";
 char name[20];
 int mark[3];
 fout<<"----------NAME-------|-M-|-A-|-I-|\n";
 for (int i=0; i<n; ++i){
  cin>>name;
  cin>>mark[0]>>mark[1]>>mark[2];
  fout<<setw(20)<<name<<" | "<<mark[0]<<" | "<<mark[1]<<" | "<<mark[2]<<"\n";
 }
 fout.close();
```

```
ifstream fin(filename);
 char ch;
 cout<<"Contents of file \n";
 while(fin.get(ch))
   cout<<ch;
 fin.close();
```

get(char&)

возвращает значение **true**, если считан символ

при достижении конца файла функция возвратит значение **false**

Для записи информации в файл создается объект класса **ofstream**.
Используемый конструктор с параметром (именем файла) создает новый файл с таким именем (или очищает существующий файл) и открывает его в текстовом режиме для записи.
ofstream fout(filename);

Для записи в текстовый файл можно использовать операцию **<<** так же, как и для стандартного потока вывода **cout**.

При этом могут быть использованы средства форматирования, например, манипулятор **setw()** для установления ширины поля вывода.

Манипуляторы описаны в **iomanip**.

```
fout<<"----------NAME-------|-1-|-2-|-3-|\n";
fout<<setw(20)<<name<<" | "<<mark[0] <<" | "<<mark[1]<<" | "<<mark[2]<<"\n";
```

После завершения записи файл должен быть закрыт; это гарантирует, что буфер будет выгружен.
fout.close();

Для того чтобы прочитать содержимое файла, используется объект класса ifstream. Конструктор с одним параметром связывает этот объект с только что созданным файлом и открывает файл для чтения в текстовом режиме.
ifstream fin(filename);

Пример. Написать программу, подсчитывающую общее количество символов в нескольких текстовых файлах. Список имен файлов, для которых нужно выполнить подсчет, задается в списке аргументов командной строки при запуске программы.

G:\count text1.txt text2.txt info.dat

```
int main(int argc, char* argv[]) {
  if (argc==1) {
    cerr<<"Usage: "<<argv[0]<<" filename[,filename[…]]\n";
    exit(1);
  }
  ifstream fin;
  long count;
  long total =0;
  char ch;
  for (int file=1; file<argc; file++) {
    fin.open(argv[file]);
    if (!fin.is_open()) {
      cerr<<"couldn't open file "<<argv[file] << "\n";
      fin.clear(); // сброс failbit
      continue;
    }
```

```
count = 0;
   while (fin.get(ch))
     count++;
   cout<<count<< " in "<<argv[file]<<"\n";
   total += count;
   fin.close();
   fin.clear();
  }
  cout<<total<<" in all files \n";
  return 0;
}
```

Параметр **argc** передает в программу количество аргументов командной строки.
В число аргументов командной строки входит и само имя выполняемой программы.
Параметр **argv[]** представляет собой массив указателей на символьные строки, содержащие аргументы командной строки

Функция **exit(1)** приводит к нормальному завершению программы. Значение параметра функции передается операционной системе. Значение **0** означает успешное завершение программы, а отличное от 0 может интерпретироваться как код ошибки.

Для того чтобы проверить, не было ли задано имя несуществующего файла, и пропустить его при обработке, используется метод **is_open()**

Состояние потока

Метод может завершиться успешно или с ошибкой.

Каждый **объект потокового класса** содержит элемент данных (**битовую маску**), описывающий **состояние потока**.

При возникновении таких ситуаций, как
достижение конца файла,
невозможность прочесть очередной байт (символ),
попытка чтения из недоступного файла,
попытка записи в защищенный файл и пр.,
один из битов состояния потока устанавливается в **1**.

Нормальная работа с потоками возможна только тогда, когда все биты маски равны **0**.

Для проверки состояния потока используются соответствующие методы.

Например, для того чтобы проверить, не было ли задано имя несуществующего файла, и пропустить его при обработке, используется метод **is_open()**, проверяющий состояние потока после открытия.

Чтобы продолжить работу со следующим файлом, нужно сбросить информацию о состоянии потока (обнулить все биты) с помощью метода **clear()**

Применять метод **clear()** нужно всякий раз перед открытием следующего файла, так как достижение конца файла тоже отражается в маске состояния потока, но метод **close()** может не сбрасывать биты в маске состояния потока.

При последовательной обработке файлов использовался **только один объект** типа **ifstream**.
В данном примере это удобно, потому что все файлы обрабатываются в цикле.

На каждый объект файлового типа выделяются ресурсы, которые будут освобождены только тогда, когда завершится время жизни этого объекта.
Поэтому данный прием экономит ресурсы.

Всегда, когда алгоритм позволяет обрабатывать файлы последовательно, рекомендуется использовать один объект файлового типа, по очереди связывая его с файлами.

Пример. Создание копии файла

```
void copyTextFile(char* inName, char* outName) {
  char c;
  ifstream inFile(inName);
  if (!inFile.is_open()) {
    cerr<<"Can't open "<<inName<<"\n";
    return;
  }
  ofstream outFile(outName);
  while (inFile.get(c))
    outFile<<c;
  inFile.close();
  outFile.close();
}
```

Пример. Выдача содержимого файла по словам

```
//пропуск пробелов
void skipBlank(ifstream& inFile) {
  while (inFile.peek() ==' ')
    inFile.ignore(1);
}

//выделяет и распечатывает по очереди слова в строке
void echoWord(ifstream& inFile) {
  char w;
  while (inFile.peek() != '\n') {
    while (inFile.peek() !=' ' && inFile.peek()!='\n') {
      inFile.get(w);
      cout<<w;
    }
    cout<<"\n";
    skipBlank(inFile);
  }
  inFile.ignore(1);
}
```

```
//выделение и печать каждого слова в текстовом файле
void printWords(char* nameFile) {
  ifstream inFile(nameFile);
  if (!inFile.is_open()){
    cerr<<"Can't open "<<nameFile<<"\n";
    return;
  }
  skipBlank(inFile);
  while (inFile.peek() != EOF)
    echoWord(inFile);
  inFile.close();
}
```

Пример. Вводить строки и дописывать их в текстовый файл. Добавляемые строки переформатировать так, что бы их длины не превышали 80 символов.

```
void appendTextFile(char* filename) {
  const int limit = 80;
  const int size = limit + 1;
  ofstream fout(filename, ios::app);
  if (!fout.is_open()) {
     cerr << "Can't open " << filename << "\n";
     exit(1);
  }
```

ofstream fout(filename, **ios::app**);

Режим файла можно задавать в конструкторе вторым параметром. При использовании только одного параметра режим задается по умолчанию.

Для класса **ifstream** по умолчанию используется режим, задаваемый константой **ios::in** (открыть для чтения).

Для класса **ofstream** значение по умолчанию **ios::out|ios::trunc** (открыть для записи и усечь файл). Поразрядный оператор «или» (|) используется для объединения двух режимов.

Режим **ios::app** означает, что файл должен быть открыт для записи с сохранением имеющейся в нем информации и добавлением новых данных в конец файла. При этом если файл не существует, то он создается.

```
  cout << "enter new strings (blank line to exit)\n";
  char line[size];
  cin.getline(line, size);
  while (line[0] != '\0') {
    fout << line << "\n";
    cin.clear();
    cin.getline(line, size);
  }
  fout.close();
}
```

При использовании функции **cin.getline(line,size)** ввод строки в переменную **line** завершается или при вводе символа новой строки, или после ввода количества символов, ограниченного константой **limit=size-1**.

Если прочитан **size-1** символ и при этом не достигнут символ конца строки, то устанавливается признак ошибки **failbit**.

В этом состоянии дальнейшее использование потока **cin** вызывает исключение. Чтобы продолжить ввод, необходимо сбросить **failbit**:
cin.clear();

Пример. Написать программу, позволяющую создать «сжатую» копию текстового файла, удалив из него все пустые строки.

При просмотре текстового файла в текстовом редакторе строка, содержащая только пробельные символы, воспринимается как пустая. Поэтому пустыми строками считать строки содержащие только символ конца строки и, возможно, пробелы.

```
int seekEoln(ifstream &in, char &c){
  c = in.get();
  int k = 0;
  while (!in.eof() && c == ' '){
    k++;
    c = in.get();
  }
  if (in.eof() || c == '\n')
    return -1;
  return k;
}
```

```
int main() {
  char filename[30] = "text.txt";
  char filename2[30] = "text2.txt";
  ifstream inFile(filename);
  if (!inFile.is_open()) {
    cerr << "Can't open " << filename << "\n";
    return -1;
  }
  ofstream outFile(filename2);
  if (!outFile.is_open()) {
      inFile.close();
     cerr << "Can't open " << filename2 << "\n";
     return -1;
  }
```

```
  int k;
  char c;
  while (!inFile.eof()){
    k = seekEoln(inFile, c);
    if (k > -1){
      for (int i = 0; i < k; ++i)
        outFile << ' ';
      while (!inFile.eof() && c != '\n'){
        outFile << c;
        c = inFile.get();
      }
      if (!inFile.eof())
        outFile << '\n';
    }
  }
  outFile.close();
  inFile.close();
  return 0;
}
```