Функции ввода/вывода в Си

Библиотека для языка Си, позволяющая использовать потоки ввода/вывода была написана Дэнисом Ричи в 1972 году, за это время она не претерпела практически никаких изменений. При подключении заголовка `<stdio.h>` в начале выполнения программы автоматически открываются три стандартных потока. Это `stdin` (стандартный поток ввода), `stdout` (стандартный поток вывода) и `stderr` (стандартный поток ошибок). Обычно эти потоки направляются в консоль, но в средах, которые поддерживают перенаправление ввода/вывода, они могут быть перенаправлены операционной системой на другое устройство. Кроме того пользователь может самостоятельно открывать и закрывать потоки для общения с файлами.

Создание потоков ввода/вывода в Си для файлов

В Си для создания потока существуют специальные функции. Функция `fopen()` открывает для использования поток, связывает файл с данным потоком и затем возвращает указатель `FILE` на данный поток. Чаще всего файл рассматривается как дисковый файл. Функция `fopen()` имеет следующий прототип:

```
FILE *fopen(const char *name, const char
*mode);
```

Поле `name` содержит Си строку, содержащую путь до файла, поле `mode` содержит Си строку с режимом выполнения этой операции. Режим может быть выставлен для чтения "`r`", записи "`w`" или добавления "`a`".

Параметр `mode` для `fopen` и `freopen` должен быть строковый и начинаться с одной из следующих последовательностей:

Режим			**Описание**	**Начинает с..**
`r`	`rb`		открывает для чтения	начала
`w`	wb		открывает для записи (создаёт файл в случае его отсутствия). Удаляет содержимое и перезаписывает файл.	начала
`a`	`ab`		открывает для добавления (создаёт файл в случае его отсутствия)	конца
`r+`	`rb+`	`r+b`	открывает для чтения и записи	начала
`w+`	`wb+`	`w+b`	открывает для чтения и записи. Удаляет содержимое и перезаписывает файл.	начала
`a+`	`ab+`	`a+b`	открывает для чтения и записи (создаёт файл в случае его отсутствия)	конца

Функции для работы с потоками в Си

Для работы с потоками в языке Си присутствуют большое количество функций.
Например функции и макросы для неформатированного ввода/вывода:
`int fgetc(FILE *stream)`, `getc(FILE *stream)` — Функция `fgetc` считывает один байт из указанного аргументом stream потока данных. Разница между `fgetc` и `getc` в том, что `getc` это макрос, а `fgetc` это функция. От функции можно взять адрес, передать её в другую функцию по указателю, в то время как от макроса нельзя взять адрес, однако макрос работает заметно быстрее функции.

`char * fgets( char *string, int num, FILE *filestream )` — Функция `fgets` считывает символы из потока и сохраняет их в виде строки в параметр `string` до тех пор пока не наступит конец строки или пока не будет достигнут конец файла. Параметр `num` это максимальное количество символов для чтения, включая нулевой символ.

Пример использования `getc()`:

```
FILE *output = fopen("output.txt", "r");

if (output == NULL) {
    printf("Ошибка открытия файла");
    exit(-1);
}

int sym;

for( int i = 0 ; i < 10; i++){
    sym = getc(output);

    //Проверка на конец файла или ошибку чтения
    if (sym == EOF){
        if( feof (output) != 0 ){
            printf ("Чтение файла закончено\n");
            exit(0);
        }else{
            printf ("\nОшибка чтения из файла\n");
            exit(-1);
        }
    }
    printf("%d\n", sym);
}
```

`int fputc(int sym, FILE *stream)`, `putc(int sym, FILE *stream)` — `putc` выводит один символ. Параметр `sym` это код выводимого символа. В случае успешной выдачи байта возвращается код выведенного байта (символа). Если при выводе байта произошла ошибка, то возвращается `EOF`.

`int fputs(const char *str, FILE *stream)` — Функция `fputs` выводит строку, на которую указывает аргумент `str`, в поток данных, на который указывает аргумент `stream`.

`int getchar(void)` — Функция `getchar` считывает символ из стандартного потока ввода. Работа функции эквивалентна работе функций `fgetc` и `getc` при чтении данных из стандартного потока ввода (`fgetc(stdin)`, `getc(stdin)`).

Пример использования fputc():

```
const char *str = "Hello";
FILE *output = fopen("output.txt", "w");

if (output == NULL) {
    printf("Ошибка открытия файла");
    exit(-1);
}

// Код символа s
int sym = 115;
int res;

res = fputc (sym, output);

// Проверка записи
if (res == EOF)
    printf ("ошибка записи\n");
else
    printf ("записан символ '%c'\n", res);

fclose(output);
```

`char *gets (char *s) (until C11)` — Функция `gets` считывает строку из стандартного потока ввода (`stdin`) и помещает её в массив указанный аргументом `s`.

`int putchar(int c)` — Функция `putchar` выводит символ в стандартный поток вывода. Эта функция аналогична функции `putc`, при указании в качестве потока вывода стандартного потока вывода (`stdout`).

`int puts (const char *s)` — Функция `puts` выводит строку в стандартный поток вывода. После вывода строки производится переход на новую строку. Символ конца строки (нулевой символ) не выводится.

`int ungetc(int c, FILE *stream)` — заносит c обратно в stream, преобразует в unsigned char, если это возможно для дальнейших операций чтения.

Функции и макросы для форматированного ввода/вывода:

Группа функций `scanf` считывает вводимую информацию в соответствии с форматом `format` так, как описано ниже. Этот формат может включать в себя определители преобразования; результаты каждого преобразования, если они производились, будут сохраняться с помощью параметров указателя. Функция `scanf` считывает информацию, прибывающую из стандартного потока ввода `stdin`; `fscanf` считывает информацию из потока, на который указывает `stream`, а `sscanf` считывает информацию из символьной строки, на которую указывает `str`.

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

#include <stdarg.h>
int vscanf(const char *format, va_list ap);
int vsscanf(const char *str, const char *format, va_list
ap);
int vfscanf(FILE *stream, const char *format, va_list ap);
```

Возможные форматные символы функции scanf для разных типов данных в целом соответствуют форматным символам для функции printf, но имеют меньше различных модификаторов.

%hhd — Считать число (десятичное) и записать его в переменную типа char (для unsigned char нужно использовать %hhu)
%hd — short int (для unsigned short int нужно использовать %hu)
%d — int (для unsigned int нужно использовать %u)
%ld — long int (для unsigned long int нужно использовать %lu)
%lld — long long int (unsigned long long int нужно использовать %llu)
%f — float
%lf — double
%Lf — long double
%c — char. Считывается один символ, он может быть пробелом или символом конца строки.
%s — Считывается последовательность символов без пробелов (строка), записывается в C-строку (типа char * или char[])

Пример для использования функции fscanf:

```
char str1[10], str2[10], str3[10];
int year;
FILE * output;

output = fopen ("output.txt", "rw");
fputs("We are in 2023", output);

rewind(output);
fscanf(output, "%s %s %s %d", str1, str2, str3,
&year);

printf("Read String1 |%s|\n", str1 );
printf("Read String2 |%s|\n", str2 );
printf("Read String3 |%s|\n", str3 );
printf("Read Integer |%d|\n", year );

fclose(output);
```

Функции семейства `printf` выводят данные в соответствии с параметром `format`, описанным ниже. Функции `printf` и `vprintf` направляют данные в стандартный поток вывода `stdout`; `fprintf` и `vfprintf` направляют данные в заданный поток вывода stream; `sprintf`, `snprintf`, `vsprintf` и `vsnprintf` направляют данные в символьную строку `str`.

```
#include <stdio.h>
int printf(const char *format, …);
int fprintf(FILE *stream, const char *format, …);
int sprintf(char* buff, const char* format, …);
```

```
#include <stdarg.h>
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list
ap);
int vsprintf(char *str, const char *format, va_list
ap);
int vsnprintf(char *str, size_t size, const char
*format, va_list ap);
```

Пример для использования функции sprintf для конкатенации строк:

```
const char *str0 = "string0";
const char *str1 = "string1";

size_t len = strlen(str0) + strlen(str1) + 2;
char buffer[len];
sprintf( buffer, "%s %s", str0, str1 );
printf( "str:'%s' len:%lu\n", buffer,
strlen(buffer) );
```

Функции `asprintf` и `vasprintf` являются аналогами `sprintf` и `vsprintf`, отличаясь только тем, что они выделяют в памяти строку, достаточную для размещения результата, включая конечный `NULL`, и возвращают указатель на эту строку через первый аргумент. Для высвобождения выделенной памяти указатель должен быть передан функции `free`.

```
int asprintf(char **strp, const char
*fmt, ...);
int vasprintf(char **strp, const char *fmt,
va_list ap);
```

Эти функции являются расширениями GNU, и не соответсвуют ни C, ни POSIX.

Пример для использования функции asprintf:

```
const char *string0 = "string0";
int a = 100;
float f = 100.1;

char *buffer;

asprintf( &buffer, "%s %d %f", string0, a, f );
printf( "str:'%s' len:%lu\n", buffer,
strlen(buffer) );

free(buffer);
```

Кроме того существуют функции подписанные буквой «w». Такие функции используются для работы с символами, имеющими размер больше 1 байта. Соответствующий тип называется `wchar_t` и может занимать 2-4 байта.

```
wscanf,fwscanf,swscanf
vwscanf,vfwscanf,vswscanf
wprintf,fwprintf,swprintf
vwprintf,vfwprintf,vswprintf
```

Чтение/запись файлов в бинарном формате в Си

Текстовые файлы позволяют хранить информацию в виде, понятном для человека. Можно, однако, хранить данные непосредственно в бинарном виде. Для этих целей используются бинарные файлы. Для работы с бинарными файлами в языке Си используется стандартная библиотека для работы с файлами, но с небольшими изменениями. Так, например, для чтения бинарного файла применяется параметр «rb», а не обычный «r». Аналогично и с модификаторами записи «w», добавления «a», чтения и записи «r+», «w+», «a+»: «wb», «ab», «rb+», «wb+», «ab+». Как и для обычных файлов наличие буквы «w» обозначает, что файл будет создан в случае отсутствия, а, если файл уже есть, то его содержимое будет удалено и заменено новым. А наличие буквы «a» значит, что файл будет создан в случае отсутствия, но содержимое файла не будет уничтожено.

Значение «b» зарезервировано для двоичного режима Си. Стандарт языка Си описывает два вида файлов — текстовые и двоичные — хотя операционная система не требует их различать. Текстовый файл — файл, содержащий текст, разбитый на строки при помощи некоторого разделяющего символа окончания строки или последовательности (в Unix — одиночный символ перевода строки`\n`; в Microsoft Windows за символом перевода строки следует знак возврата каретки)`\r\n`. При считывании байтов из текстового файла, символы конца строки обычно связываются (заменяются) с переводом строки для упрощения обработки. При записи текстового файла одиночный символ перевода строки перед записью связывается (заменяется) с специфичной для ОС последовательностью символов конца строки. Двоичный файл — файл, из которого байты считываются и выводятся в «сыром» виде без какого-либо связывания (подстановки).

Для считывания двоичных файлов можно использовать например функцию fread.

```
size_t fread ( void * ptr, size_t size, size_t count,
FILE * stream );
```

Функция возвращает число удачно прочитанных элементов, которые помещаются по адресу `ptr`. Всего считывается `count` элементов по `size` байт.

```
char buffer[6];
FILE *input = fopen("output.txt", "rb");

if (input == NULL) {
    printf("Error opening file");
    exit(-1);
}

fread(str, 6, 1, input);
printf("%s\n", buffer);
fclose(input);
```

Иногда для считывания всего файла необходимо подготовить заранее пространство для его хранения. Вычислить размер файла могут помочь функции файлового позиционирования. Например:

```
int rs;
int size;
void *data;
FILE *fd = fopen("output.txt", "rb");
if( fd == NULL )
    exit(-1);
rs = fseek(fd, 0L, SEEK_END);
if(rs != 0){
    fclose(fd);
    exit(-1);
}
size = ftell(fd);
if(size <= 0){
    fclose(fd);
    exit(-1);
}
rs = fseek(fd, 0L, SEEK_SET);
if(rs != 0){
   fclose(fd);
   exit(-1);
}
data = malloc(size + 1);
rs = fread( data, size, 1, fd );
((char*)data)[size] = '\0';
if( rs < 0 ){
    fclose(fd);
    exit(-1);
}
printf("str: '%s' filelen: %u\n", (char*)data, size);
fclose(fd);
free(data);
```

В этом примере задействованы функции `fseek` и `ftell`.

Функция `int fseek(FILE *stream, long int offset, int whence)` находится в заголовочном файле `stdio.h`. В качестве аргументов используются параметры `stream` – указатель на поток данных, `offset` – смещение позиции, `whence` – точка отсчета смещения. Возвращает 0 при успешной установке позиции. В качестве параметра `whence` принимаются параметры `SEEK_SET` – смещение отсчитывается от начала файла, `SEEK_CUR` – смещение отсчитывается от текущей позиции, `SEEK_END` – смещение отсчитывается от конца файла.

Функция `long int ftell (FILE *stream)` находится в заголовочном файле `stdio.h`. Функция принимает указатель на поток данных, возвращает в случае успешного выполнения позицию в потоке данных, в случае ошибки возвращается -1.

Функция `void rewind (FILE *stream)` находится в заголовочном файле `stdio.h`. Функция `rewind` устанавливает текущую позицию для чтения/записи файла, связанного с потоком данных в начало файла и сбрасывает индикатор ошибок потока данных.

Запись в файл осуществляется с помощью функции

```
size_t fwrite ( const void * ptr, size_t size, size_t
count, FILE * stream );
```

Функция `fwrite()` записывает `count` объектов, каждый объект по `size` символов в длину в поток `stream`. Функция возвращает число удачно записанных элементов. В качестве аргументов принимает указатель на массив, размер одного элемента, число элементов и указатель на файловый поток.

```
const char *str = "Hello";
FILE *output = fopen("output.txt", "wb");

if (output == NULL) {
    printf("Error opening file");
    exit(-1);
}

fwrite(str, strlen(str), 1, output);
fclose(output);
```

Потоки ввода/вывода в C++

Поток — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Потоки C++ («stream»), в отличие от функций ввода/вывода в стиле C, обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис. По сути, ввод/вывод в языке C++ реализован с помощью потоков. Со временем поток может производить или потреблять потенциально неограниченные объемы данных. Функционал потоков ввода/вывода не определен как часть языка C++, а предоставляется Стандартной библиотекой C++, следовательно, находится в пространстве имен std.

Поток ввода используется для хранения данных, полученных от источника данных: клавиатуры, файла, сети и т.д. Например, пользователь может нажать клавишу на клавиатуре в то время, когда программа не ожидает ввода. Вместо игнорирования нажатия клавиши, данные помещаются во входной поток, где затем ожидают ответа от программы.

Поток вывода используется для хранения данных, предоставляемых конкретному потребителю данных: монитору, файлу, принтеру и т.д. При записи данных на устройство вывода, это устройство может быть не готовым принять данные немедленно. Например, принтер все еще может прогреваться, когда программа уже записывает данные в выходной поток. Таким образом, данные будут находиться в потоке вывода до тех пор, пока принтер не начнет их использовать. Некоторые устройства, такие как файлы и сети, могут быть источниками как ввода, так и вывода данных.

Чаще всего программисту не нужно знать детали взаимодействия потоков с разными устройствами и источниками данных, необходимо только научиться взаимодействовать с этими потоками для чтения и записи данных.

По виду устройств, с которыми работает поток, можно разделить потоки на стандартные, файловые и строковые.
Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея, файловые потоки — для обмена информацией с файлами на внешних носителях данных (например, на магнитном диске), а строковые потоки — для работы с массивами символов в оперативной памяти.
Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов — `ios` и `streambuf`. Класс `ios` содержит общие для ввода и вывода поля и методы, класс `streambuf` обеспечивает буферизацию потоков и их взаимодействие с физическими устройствами. От этих классов наследуется класс `istream` для входных потоков и `ostream` — для выходных.

Иерархия потоков ввода/вывода

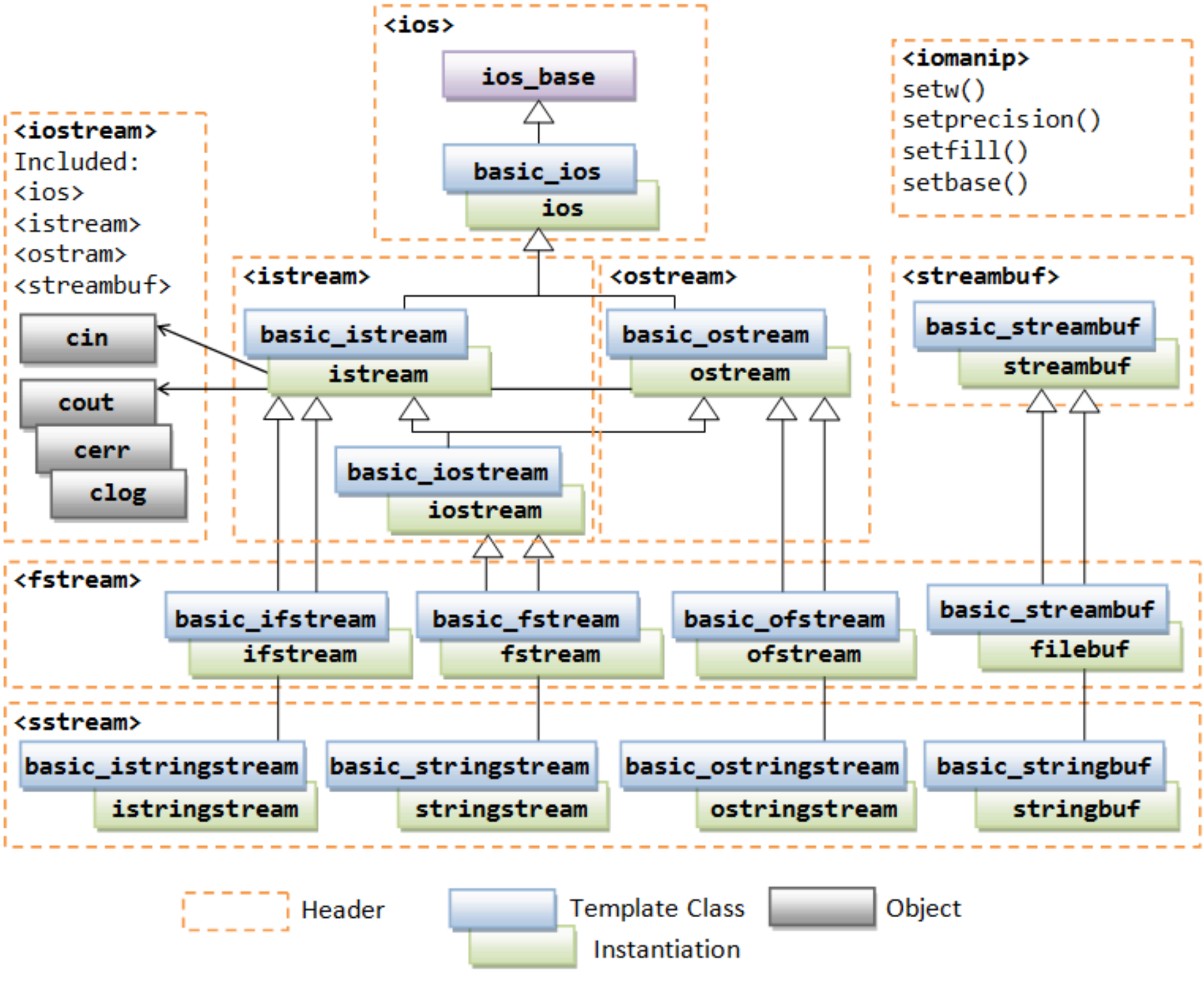
Класс `istream` используется для работы с входными потоками. Оператор извлечения >> используется для извлечения значений из потока. Это имеет смысл: когда пользователь нажимает на клавишу клавиатуры, код этой клавиши помещается во входной поток. Затем программа извлекает это значение из потока и использует его.

Класс `ostream` используется для работы с выходными потоками. Оператор вставки << используется для помещения значений в поток.

Класс `iostream` может обрабатывать как ввод, так и вывод данных, что позволяет ему осуществлять двунаправленный ввод/вывод.

Классы потоков:

`ios` — базовый класс потоков;
`istream` — класс входных потоков;
`ostream` — класс выходных потоков;
`iostream` — класс двунаправленных потоков;
`istringstream` — класс входных строковых потоков;
`ostringrstream` — класс выходных строковых потоков;
`stringstream` — класс двунаправленных строковых потоков;
`ifstream` — класс входных файловых потоков;
`ofstream` — класс выходных файловых потоков;
`fstream` — класс двунаправленных файловых потоков.


<ios>
ios_base
basic_ios
ios
<iomanip>
setw()
setprecision()
setfill()
setbase()
<iostream>
Included:
<ios>
<istream>
<ostram>
<streambuf>
cin
cout
cerr
clog
<istream>
basic_istream
istream
basic_iostream
iostream
<ostream>
basic_ostream
ostream
<streambuf>
basic_streambuf
streambuf
<fstream>
basic_ifstream
ifstream
basic_fstream
fstream
basic_ofstream
ofstream
basic_streambuf
filebuf
<sstream>
basic_istringstream
istringstream
basic_stringstream
stringstream
basic_ostringstream
ostringstream
basic_stringbuf
stringbuf
Header
Template Class
Instantiation
Object

Описания классов находятся в заголовочных файлах:

`<ios>` — базовый класс потоков ввода/вывода
`<iosfwd>` — предварительные объявления средств ввода/вывода;
`<istream>` — шаблон потока ввода;
`<ostream>` — шаблон потока вывода;
`<iostream>` — стандартные объекты и операции с потоками ввода/вывода;
`<fstream>` — потоки ввода/вывода в файлы;
`<sstream>` — потоки ввода/вывода в строки;
`<streambuf>` — буферизация потоков ввода/вывода;
`<iomanip>` — манипуляторы;

Встроенные потоки ввода/вывода

Заголовочный файл `<iostream>` содержит, кроме описания классов для ввода/вывода, четыре предопределенных объекта: `cin`, `cout`, `cerr`, `clog`. Поток `cin` это объект класса `istream`, стандартный входной поток, `cout` это объект класса `ostream`, стандартный выходной поток. Потоки `cerr` и `clog` являются объектами класса `ostream`, они необходимы для вывода информации о ошибках. Разница между ними в том, что `cerr` в отличие от `clog` не буферизирован. Это значит, что любые данные, посланные в поток `cerr` будут немедленно выведены, а при использовании потока `clog` данные будут сначала накопленные в буфере, а затем только выведены. В C++ также предусмотрены многобайтовые версии стандартных потоков, именуемые `wcin`, `wcout`, `wcerr` и `wclog`. Они предназначены для расширенной поддержки языков мира.

В классах `istream` и `ostream` операции извлечения из потока `>>` и помещения в поток `<<` определены путем перегрузки операций сдвига. Операции извлечения и чтения в качестве результата своего выполнения формируют ссылку на объект типа `istream` для извлечения и ссылку на `ostream` — для чтения. Это позволяет формировать цепочки операций. Вывод при этом выполняется слева направо:

```
int i;
cin >> i;
cout << "Вы ввели " << i << endl;
```

Как и для других перегруженных операций, для вставки и извлечения невозможно изменить приоритеты, поэтому в необходимых случаях используются скобки:

```
// Скобки не требуются — приоритет сложения больше, чем << :
cout << 1 + j;

// Скобки необходимы — приоритет операции отношения меньше, чем << :
cout << ( 1 < j );

// Правая операция << означает сдвиг
cout << (1 << j );
```

Операции << и >> перегружены для всех встроенных типов данных, что позволяет автоматически выполнять ввод и вывод в соответствии с типом величин. Это означает, что при вводе последовательность символов преобразуется во внутреннее представление величины, стоящей справа от знака извлечения, а при выводе выполняется обратное преобразование, например:

```
#include <iostream.h>

int main(){
    int i = 0xD;
    double d;
    // Символы из потока ввода преобразуются в double:
    cin >> d;
    // int и double преобразуются в строку символов:
    cout << i << " " << d;
}
```

Функционал istream

Самый основной пример функционала библиотеки istream это считывание информации из входного потока с помощью оператора извлечения <<:

```
#include <iostream>

char buf[12];
std::cin >> buf;
```

Однако такой код может вызвать ошибки потому что при вводе с клавиатуры достаточно большого текста произойдет переполнение. Одним из способов решения этой проблемы является использование манипуляторов. Манипулятор — это объект, который применяется для изменения потока данных с использованием операторов извлечения >> или вставки <<. Часто можно встретить манипулятор `endl`, который одновременно выводит символ новой строки и удаляет текущие данные из буфера. Язык C++ предоставляет еще один манипулятор `setw()` (заголовок `iomanip`), который используется для ограничения количества символов, считываемых из потока. Для использования `setw()` необходимо просто передать в качестве параметра максимальное количество символов для извлечения. Эта программа прочитает только первые 11 символов из входного потока (один символ для нуль-терминатора). Все остальные символы останутся в потоке до следующего извлечения.

```
#include <iostream>
#include <iomanip>

char buf[12];
std::cin >> std::setw(12) >> buf;
```

Оператор извлечения работает с «отформатированными» данными, то есть он игнорирует все пробелы, символы табуляции и символ новой строки. Например следующая программа выведет текст без введенных знаков пробела:

```
#include <iostream>

char ch;
while (std::cin >> ch)
    std::cout << ch;

>> Hello World
<< HelloWorld
```

Часто пользовательский ввод все же нужен со всеми его пробелами. Для этого класс `istream` предоставляет функцию `get()`, которая извлекает символ из входного потока.

```
#include <iostream>

char ch;
while (std::cin.get(ch))
  std::cout << ch;
```

```
>> Hello World
<< Hello World
```

Функция get() также имеет перегрузку для строковой версии, в которой можно указать максимальное количество символов для извлечения.

```
#include <iostream>

char strBuf[10];
std::cin.get(strBuf, 10);
std::cout << strBuf << std::endl;

>> Hello World
<< Hello Wor
```

Функция `get()` не считывает символ новой строки, это может вызвать некоторые проблемы.

```
char strBuf[12];

// первые 11 символов
std::cin.get(strBuf, 12);
std::cout << strBuf << std::endl;

// дополнительно еще 11 символов
std::cin.get(strBuf, 12);
std::cout << strBuf << std::endl;
```

Если пользователь введет:

```
>> Hello
```

То получит:

```
<< Hello
```

И программа сразу же завершит свое выполнение, первый get() считывает символы до символа новой строки, а затем останавливается. Второй get() увидит, что во входном потоке все еще есть данные и попытается их извлечь. Но первый символ, на который он натыкается — символ новой строки, поэтому происходит завершение программы.

Для решения данной проблемы
класс `istream` предоставляет функцию `getline()`, которая работает точно так же, как и функция `get()`, но при этом может считывать символы новой строки. Для того, чтобы узнать количество символов, извлеченных последним `getline()`, можно использовать функцию gcount().

```
char strBuf[12];

// первые 11 символов
std::cin.getline(strBuf, 12);
std::cout << strBuf << std::endl;

// дополнительные 11 символов
std::cin.getline(strBuf, 12);
std::cout << strBuf << std::endl;
std::cout << std::cin.gcount() << " characters were
read" << std::endl;
```

Есть специальная версия функции `getline()`, которая находится вне класса `istream` и используется для считывания переменных типа `std::string`. Она не является членом ни `ostream`, ни `istream`, а подключается заголовочным файлом `string`.

```
#include <iostream>
#include <string>

std::string strBuf;
getline(std::cin, strBuf);
std::cout << strBuf << std::endl;
```

Существует ещё несколько необходимых функций класса istream:

Функция ignore() — игнорирует первый символ из потока;

Функция ignore(int nCount) — игнорирует
первые nCount (количество) символов из потока;

Функция peek() — считывает символ из потока, при этом не удаляя его из потока;

Функция unget() — возвращает последний считанный символ обратно в поток, чтобы его можно было извлечь в следующий раз;

Функция putback(char ch) — помещает выбранный символ обратно в поток, чтобы его можно было извлечь в следующий раз.

Функционал ostream

Оператор вывода << используется для помещения информации в выходной поток.
Классы `istream` и `ostream` являются дочерними классу `ios`, а одной из задач `ios` и `ios_base` является управление параметрами форматирования вывода.
Есть два способа управления параметрами форматирования вывода: флаги — это логические переменные, которые можно включить/выключить и манипуляторы — это объекты, которые помещаются в поток и влияют на способ ввода/вывода данных.
Для включения флага используется функция `setf()` с соответствующим флагом в качестве параметра.

Например, по умолчанию C++ не выводит знак + перед положительными числами. Однако, используя флаг `std::showpos`, можно это изменить:

```
std::cout.setf(std::ios::showpos);  // включение флага std::showpos
std::cout << 30 << std::endl;  // >> +30
```

Также можно включить сразу несколько флагов,
используя побитовый оператор ИЛИ (|):

```
std::cout.setf(std::ios::showpos |
std::ios::uppercase); //включение флагови std::showpos и
std::uppercase
std::cout << 30 << std::endl; // >> +30
```

Чтобы отключить флаг, можно использовать функцию unsetf():

```
std::cout.setf(std::ios::showpos); // включаем флаг std::showpos
std::cout << 30 << std::endl;
std::cout.unsetf(std::ios::showpos); // выключаем флаг std::showpos
std::cout << 31 << std::endl;
```

```
>> +30
>> 31
```

Флаги группы форматирования `basefield` управляют выводом целочисленных значений. По умолчанию установлен флаг `std::dec`, т.е. значения выводятся в десятичной системе счисления.
`oct` (от англ. «octal» = «восьмеричный») — восьмеричная система счисления;
`dec` (от англ. «decimal» = «десятичный») — десятичная система счисления;
`hex` (от англ. «hexadecimal» = «шестнадцатеричный») — шестнадцатеричная система счисления.

```
std::cout.setf(std::ios::hex);  // флаг std::hex
std::cout.unsetf(std::ios::dec); // снять флаг std::dec
std::cout << 30 << std::endl;
```

```
>> 1e
```

Язык C++ также предоставляет еще один способ изменения параметров форматирования: манипуляторы. Манипуляторы способны одновременно включать и выключать соответствующие флаги. Например:

```
std::cout << std::hex << 30 << std::endl; // выводим 30 в шестнадцатеричной системе счисления
std::cout << 31 << std::endl; // все еще шестнадцатеричный вывод
std::cout << std::dec << 32 << std::endl; // обратно в десятичную систему счисления
```

Полезные флаги, манипуляторы и методы

Флаг boolalpha — если включен, то логические значения выводятся как `true`/`false`. Если выключен, то логические значения выводятся как `0`/`1`.

Манипулятор boolalpha — логические значения выводятся как `true`/`false`.

Манипулятор noboolalpha — логические значения выводятся как 0/1.

```
std::cout << true << " " << false << std::endl;
std::cout.setf(std::ios::boolalpha);
std::cout << true << " " << false << std::endl;
```

Флаг showpos — если включен, то перед положительными числами указывается знак +.

Манипулятор showpos — перед положительными числами указывается знак +.

Манипулятор noshowpos — перед положительными числами не указывается знак +.

```
std::cout << 7 << std::endl;
std::cout.setf(std::ios::showpos);
std::cout << 7 << std::endl;
```

```
>>+7
>>7
```

Флаг uppercase — если включен, то используются заглавные буквы.
Манипулятор uppercase — используются заглавные буквы.
Манипулятор nouppercase — используются строчные буквы.

```
std::cout << 12345678.9 << std::endl;
std::cout.setf(std::ios::uppercase);
std::cout << 12345678.9 << std::endl;
std::cout << std::nouppercase << 12345678.9 <<
std::endl;
std::cout << std::uppercase << 12345678.9 <<
std::endl;
```

```
>>1.23457e+07
>>1.23457E+07
>>1.23457e+07
>>1.23457E+07
```

Флаг dec — значения выводятся в десятичной системе счисления;
Флаг hex — значения выводятся в шестнадцатеричной системе счисления;

Флаг oct — значения выводятся в восьмеричной системе счисления.

Манипулятор dec — значения выводятся в десятичной системе счисления;

Манипулятор hex — значения выводятся в шестнадцатеричной системе счисления;

Манипулятор oct — значения выводятся в восьмеричной системе счисления.

Флаг `fixed` — используется десятичная запись чисел типа с плавающей точкой;
Флаг `scientific` — используется экспоненциальная запись чисел типа с плавающей точкой;
Флаг `showpoint` — всегда отображается десятичная точка и конечные нули для чисел типа с плавающей точкой.
Манипулятор `fixed` — используется десятичная запись значений;
Манипулятор `scientific` — используется экспоненциальная запись значений;
Манипулятор `showpoint` — отображается десятичная точка и конечные нули чисел типа с плавающей точкой;
Манипулятор `noshowpoint` — не отображаются десятичная точка и конечные нули чисел типа с плавающей точкой;
Манипулятор `setprecision(int)` — задаем точность для чисел типа с плавающей точкой.
Метод `precision()` — возвращаем текущую точность для чисел типа с плавающей точкой;
Метод `precision(int)` — задаем точность для чисел типа с плавающей точкой.

```
#include <iostream>
#include <iomanip>

std::cout << std::fixed;
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';

std::cout << std::scientific << '\n';
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';

>> 123.456
>> 123.4560
>> 123.45600
>> 123.456000
>> 123.4560000

>> 1.235e+02
>> 1.2346e+02
>> 1.23456e+02
>> 1.234560e+02
>> 1.2345600e+02
```

Если не используются ни десятичная, ни экспоненциальная запись чисел, то точность определяет, сколько значащих цифр будет отображаться.

```
#include <iostream>
#include <iomanip>

std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << ‘\n';

>> 123
>> 123.5
>> 123.46
>> 123.456
>> 123.456
```

Используя манипулятор или флаг showpoint, можно заставить программу выводить десятичную точку и конечные нули.

```
std::cout << std::showpoint;
std::cout << std::setprecision(3) << 123.456 << ‘\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';
```

```
>> 123.
>> 123.5
>> 123.46
>> 123.456
>> 123.4560
```

Обычно числа выводятся без учета пространства вокруг них. Тем не менее, числа можно выравнивать. Чтобы это сделать, нужно сначала определить ширину поля (т.е. количество пространства (пробелов) вокруг значений).

Чтобы использовать любой из перечисленных объектов, нужно сначала установить ширину поля. Это делается с помощью метода `width(int)` или манипулятора `setw()`. По умолчанию при использовании ширины поля значения выравниваются по правому краю.

Флаг `internal` — знак значения выравнивается по левому краю, а само значение — по правому краю;

Флаг `left` — значение и его знак выравниваются по левому краю;

Флаг `right` — значение и его знак выравниваются по правому краю.

Манипулятор `internal` — знак значения выравнивается по левому краю, а само значение — по правому краю;

Манипулятор `left` — значение и его знак выравниваются по левому краю;

Манипулятор `right` — значение и его знак выравниваются по правому краю;

Манипулятор `setfill(char)` — задаем символ-заполнитель;

Манипулятор `setw(int)` — задаем ширину поля.

Метод `fill()` — возвращаем текущий символ-заполнитель;

Метод `fill(char)` — задаем новый символ-заполнитель;

Метод `width()` — возвращаем текущую ширину поля;

Метод `width(int)` — задаем ширину поля.

```
std::cout << -12345 << '\n'; // выводим значение без использования ширины поля
std::cout << std::setw(10) << -12345 << '\n'; // выводим значение с использованием ширины поля
std::cout << std::setw(10) << std::left << -12345 << '\n'; // выравниваем по левому краю
std::cout << std::setw(10) << std::right << -12345 << '\n'; // выравниваем по правому краю
std::cout << std::setw(10) << std::internal << -12345 << '\n'; // знак значения выравнивается по левому краю, а само значение - по правому
```

```
>>-12345
>>    -12345
>>-12345
>>    -12345
>>-    12345
```

Кроме прочего с помощью метода `fill` можно указать свой символ-заполнитель.

```
std::cout.fill('*');
std::cout << -12345 << '\n'; // выводим значение без использования ширины поля
std::cout << std::setw(10) << -12345 << '\n'; // выводим значение с использованием ширины поля
std::cout << std::setw(10) << std::left << -12345 << '\n'; // выравниваем по левому краю
std::cout << std::setw(10) << std::right << -12345 << '\n'; // выравниваем по правому краю
std::cout << std::setw(10) << std::internal << -12345 << '\n'; // знак значения выравнивается по левому краю, а само значение - по правому
```

```
>>-12345
>>****-12345
>>-12345****
>>****-12345
>>-****12345
```

Ввод/вывод текстовых данных в файл

Заголовочный файл `fstream` предоставляет функционал для считывания данных из файла и для записи в файл. В целом он очень похож на `iostream`, который работает с консолью, поскольку консоль это тоже файл. Поэтому все основные операции такие же, за мелкими отличиями.

- Операторы перенаправления ввода и вывода << и >>.
- Методы записи и чтения строк `getline()` и `get()` с `put()`.
- Потоковая запись и чтение методами `write()` и `read()`.
- Методы открытия\создания и закрытия файлов `open()` и `close()`.
- Методы проверки открыт ли файл `is_open()` и достигнут ли конец файла `eof()`.
- Настройка форматированного вывода для >> с помощью `width()` и `precision()`.
- Операции позиционирования `tellg()`, `tellp()` и `seekg()`, `seekp()`.

std::ifstream

Класс ifstream предоставляет возможности для чтения файлов. Открыть файл можно двумя способами: вызвав метод open() или указав путь к нему в конструкторе. Для рассмотренного примера необходим текстовый файл с названием `file.txt`.

```
void ifstream::open( const char *filename,
ios::openmode mode = ios::in);
```

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream file ("file.txt"); // указание файла через конструктор
    file.close();
}
```

Конструкторы с параметрами создают объект соответствующего класса, от крывают файл с указаным именем и связывают файл с объектом:

ifstream(const char *name, int mode = ios::in);
ofstream(const char *name, int mode = ios::out | ios::trunc);
fstream(const char *name, int mode = ios::in | ios::out);

Вторым параметром конструктора является режим открытия файла. Если установленное по умолчанию значение не устраивает программиста, можно указать другое, составив его из битовых масок, определенных в классе ios:

```
enum open_mode{
    in = 0x01,          // Открыть для чтения
    out = 0x02,         // Открыть для записи
    ate = 0x04,         // Установить указатель на конец файла
    app = 0x08,         // Открыть для добавления в конец
    trunc = 0x10,       // Если файл существует, удалить
    nocreate = 0x20,    // Если файл не существует, выдать ошибку
    noreplace = 0x40,   // Если файл существует, выдать ошибку
    binary = 0x80       // Открыть в двоичном режиме
};
```

Файл можно открыть методом open().

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream file;
    file.open("file.txt"); // открытие файла через метод open
}
```

Открытие файла может не состояться в силу разных причин, необходимо проверить. Для проверки можно пойти двумя путями: проверить переменную файла в логическом выражении или использовать метод `is_open()`

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream file ("file.txt");
    if (!file)
    {
        std::cout << "Файл не открыт" << std::endl;
        return -1;
    }
    else
    {
        std::cout << "Файл открыт!" << std::endl;
        return 1;
    }
}
```

Проверка методом `is_open()`. Метод `is_open()` вернет `1`, если файл найден и успешно открыт. Иначе вернет `0` и сработает код прописанный в блоке `else`.

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream file ("file.txt");
    if (file.is_open())
        std::cout << "Открыт!" << std::endl;
    else{
        std::cout << "Не открыт!" << std::endl;
        return -1;
    }
}
```

Так же как и в `iostream` считывание можно организовать оператором >>, который указывает в какую переменную будет произведено считывание. Поток считает вещественное, целое и строку. Считывание строки закончится, если появится пробел или конец строки. Стоит отметить, что оператор >> применяется к текстовым файлам.

...

```
double d;
int i;
string s;
file >> d >> i >> s;
```

Считывание целой строки до перевода каретки производится так же как и в `iostream` методом `getline()`. Причем рекомендуется использовать его переопределеную версию в виде функции, если считывается строка типа `string`:

```
std::string s;
getline(file, s); //считывание строки 0 из текста
std::cout << s << std::endl;
getline(file, s); //считывание строки 1 из текста
std::cout << s << std::endl;
```

Если нужно считать данные в массив символов `char[]`, то стоит использовать либо `get()` либо `getline()`:

```
int n = 10;
char* buffer = new char[n+1]; // буффер для чтения
buffer[n]=0;
file.get( buffer, n ); // чтение n символов
file.getline(buffer, n, ' '); // до первого пробела
cout << buffer; // вывод считанного
delete [] buffer; // освобождение буфера
```

Считывание с помощью метода `read()`. Считается указанное количество символов. Исключение только в том, что нельзя указать разделитель. `read()` применяется для неформатированного ввода, например для считывания бинарных файлов.

```
int n = 10; // считывание из файла n байт
char* buffer = new char[n+1];
buffer[n] = 0;
file.read(buffer, n);
std::cout << buffer << std::endl;
delete [] buffer;
```

Открытый файл, после считывания, должен быть закрыт. Для этого существует метод `close()`.
Метод `eof()` проверяет не достигнут ли конец файла, то есть можно ли из него продолжать чтение.

```
std::string s;
for(file >> s; !file.eof(); file >> s){
      std::cout << s << std::endl;
}
```

Метод `seekg()` производит установку текущей позиции в нужную, указываемую числом. В этот метод так же передается способ позиционирования: `std::ios_base::end` – отсчитать новую позицию с конца файла, `std::ios_base::beg` – Отсчитать новую позицию с начала файла, `std::ios_base::cur` — перескочить на n байт начиная от текущей позиции в файле.

```
file.seekg(0, std::ios_base::end); // переход в конец файла
file.seekg(10, std::ios_base::end); // переход на 10 байтов с конца
file.seekg(30, std::ios_base::beg); //переход на 31-й байт
file.seekg(3, std::ios_base::cur); //перепрыгнуть через 3 байта
file.seekg(3); //перепрыгнуть через 3 байта - аналогично
```

Иногда нужно получать информацию о том, сколько уже прочитано. В этом поможет метод tellg():

```
std::cout << "Считано байт: " << file.tellg() <<
std::endl;
```

Метод возвращает значение типа `int`, которое показывает сколько уже пройдено в байтах. Его можно использовать в паре с методом `seekg()`, чтоб получать размер файла:

```
file.seekg(0, ios_base::end); // в конец файла
cout << "Размер файла (в байтах): " << file.tellg() <<
std::endl; // текущая позиция
file.seekg(0, ios_base::beg); // возврат в начало
```

std::ofstream

Класс ofstream предоставляет функционал для записи файлов.

```
void ofstream::open( const char *filename, ios::openmode
mode = ios::out);
```

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream f;
    f.open("output.txt"); // Использование метода open
    if (!f){
        std::cout << "Невозможно открыть файл для записи";
        return 1;
    }
    f << "Hello" << std::endl;
    f.close();
}
```

Ввод/вывод бинарных данных в файл

В C++ доступно открывать файл в бинарном режиме. По умолчанию все файлы открываются в текстовом режиме, при котором могут происходить преобразования символов (например, последовательность, состоящая из символов возврата каретки и перехода на новую строку, может быть преобразована символ новой строки). При открытии файла в бинарном режиме преобразования символов не выполняется. Для того, чтобы открыть файл в бинарном представлении нужно использовать параметр `mode` как `ios::binary`.

Считывание из бинарного файла производить лучше всего с помощью метода `read()`.

```
basic_istream& read( char_type* s, std::streamsize count );
```

```
#include <fstream.h>

...

char buffer[10];
ifstream in ("data.bin", ios::in | ios::binary);
in.read (buffer, 10);
if (! in) {
      // in.gcount() возвращает количество прочитанных байт
      std::cout << in.gcount() << std::endl;

      // вызов in.clear() очищает внутреннее состояние потока
      in.clear();
}

...

if (!in.read (buffer, 10)) {
      // считаются те же самые данные
}
in.close();
```

Запись в бинарный файл лучше всего производить с помощью метода `write()`.

```
basic_ostream& write( const char_type* s, std::streamsize
count );
```

```
#include <fstream.h>

...

char buffer[10];
ofstream out("out.bin", ios::out | ios::binary | ios::trunc);
if (!out.is_open()) {
    std::cerr << "Error in open file './out.bin'" <<
std::endl;
    exit(-1);
}

int num = 111;
out.write(reinterpret_cast<const char*>(&num), sizeof(num));
out.close();
```

Потоковые классы и строки

В Стандартной библиотеке C++ есть отдельный набор классов, которые позволяют использовать операторы вставки (<<) и извлечения (>>) со строками. Как и istream с ostream, так и потоковые классы для строк предоставляют буфер для хранения данных. Однако в отличие от cin и cout, эти потоковые классы не подключены к каналу ввода/вывода (т.е. к клавиатуре, монитору и т.д.).

Есть 6 потоковых классов, которые используются для чтения и записи строк:

— класс istringstream (является дочерним классу istream);
— класс ostringstream (является дочерним классу ostream);
— класс stringstream (является дочерним классу iostream);
— класс wistringstream;
— класс wostringstream;
— класс wstringstream.

Поток `stringstream`, позволяет связать поток ввода-вывода со строкой в памяти. Всё, что выводится в такой поток, добавляется в конец строки; всё, что считывается из потока — извлекается из начала строки.Чтобы использовать класс `stringstream`, нужно подключить заголовочный файл `sstream`. Чтобы добавить данные в `stringstream`, можно использовать оператор вставки (<<) или метод `str`:

```
#include <iostream>
#include <sstream>  // для stringstream

int main()
{
   std::stringstream myStringStream0;
   std::stringstream myStringStream1;

   myStringStream0 << "Фраза для вставки" << std::endl; // передача "Фраза для вставки" в stringstream

  myStringStream1.str("Фраза для вставки"); // присвоение "Фраза для вставки» stringstream

}
```

Аналогично, чтобы получить данные обратно из `stringstream`, можно использовать функцию `str()`:

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
  std::stringstream myStringStream;
  myStringStream << "Фраза для вставки" <<
std::endl;
  std::cout << myStringStream.str();
}
```

Конвертация числ в строку с помощью `stringstream`:

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
  int nValue = 336000;
  double dValue = 12.14;
  std::stringstream myStringStream;
  myStringStream << nValue << " " << dValue;
  std::cout << myStringStream.str() <<
std::endl;
}
```

Конвертация числовой строки обратно в числа с помощью `stringstream`:

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
  int nValue;
  double dValue;
  std::stringstream myStringStream;
  myStringStream << "336000 12.14";
  myStringStream >> nValue >> dValue;
  std::cout << nValue << " " << dValue <<
std::endl;
}
```

Для очистки `stringstream` можно передать пустую строку в str() или вызвать метод clear():

```
#include <iostream>
#include <sstream> // для stringstream

int main()
{
  std::stringstream myStringStream;
  myStringStream << "Фраза для вставки" <<
std::endl;
  myStringStream.clear(); // myStringStream.str("");
}
```

Перегрузка операторов ввода/вывода

В языке C++ оператор "<<" выводит информацию в поток, а оператор ">>" выводит информацию из потока. Операторы ввода и вывода уже перегружены в iostream — они могут быть использованы с любыми встроенными типами в C++. Также можно определить эти операторы для пользовательских классов.
Например:

```
class Vertex{
public:
    double x, y, z;
    Vertex(double a_x, double a_y, double a_z): x(a_x),
y(a_y), z(a_z){}
};

std::ostream &operator<<( std::ostream &stream, Vertex
&vertex){
    stream << vertex.x << ", ";
    stream << vertex.y << ", ";
    stream << vertex.z << std::endl;
    return stream;
}
```

Теперь вывод данных, содержащихся в объекте класса Vertex можно инициировать стандартным потоком данных.

```
int main(){

    Vertex a( 0.1, 0.45, 1.1);
    Vertex b( 0.1, 0.45, 1.1);
    Vertex c( 0.1, 0.45, 1.1);

    std::cout << a << b << c << std::endl;
}
```

Чтобы не оставлять данные открытыми и не плодить лишние методы геттеры, можно сделать перегрузку оператора вывода дружественной.

```
class Vertex{
    double x, y, z;
public:
   Vertex(double a_x, double a_y, double a_z): x(a_x), y(a_y),
z(a_z){}
    friend std::ostream &operator<<( std::ostream &stream, Vertex
&vertex);
};

std::ostream &operator<<( std::ostream &stream, Vertex &vertex){
    stream << vertex.x << ", ";
    stream << vertex.y << ", ";
    stream << vertex.z << std::endl;
    return stream;
}

int main(){
   Vertex a( 0.1, 0.45, 1.1);
   Vertex b( 0.1, 0.45, 1.1);
   Vertex c( 0.1, 0.45, 1.1);
   std::cout << a << b << c << std::endl;
}
```

Таким же образом можно перегрузить операторы ввода для пользовательского класса.

```
class Vertex{
public:
    double x, y, z;
    Vertex(): x(0), y(0), z(0) {}
    Vertex(double a_x, double a_y, double a_z): x(a_x), y(a_y),
z(a_z){}
friend std::istream &operator>>( std::istream &stream, Vertex
&vertex);
friend std::ostream &operator<<( std::ostream &stream, Vertex
&vertex);
};

std::istream &operator>>( std::istream &stream, Vertex &vertex){
    stream >> vertex.x >> vertex.y >> vertex.y;
    return stream;
}

int main(){
   Vertex a;
   std::cin >> a;
   std::cout << a << std::endl;
}
```